

[Technologies](#) ▼[References & Guides](#) ▼[Feedback](#) ▼[Sign in](#) 

Same-origin policy

The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. It is a critical security mechanism for isolating potentially malicious documents.

Definition of an origin

Two pages have the same origin if the protocol, port (if one is specified), and host are the same for both pages. You'll see this referred to as the "scheme/host/port tuple" at times (where a "tuple" is a set of three components that together comprise a whole).

The following table gives examples of origin comparisons to the URL


`http://store.company.com/dir/page.html`:

URL	Outcome	Reason
<code>http://store.company.com/dir2/other.html</code>	Success	
<code>http://store.company.com/dir/inner/another.html</code>	Success	
<code>https://store.company.com/secure.html</code>	Failure	Different protocol
<code>http://store.company.com:81/dir/etc.html</code>	Failure	Different port
<code>http://news.company.com/dir/other.html</code>	Failure	Different host

See also origin definition for `file:` URLs.

Inherited origins

Scripts executed from pages with an `about:blank` or `javascript:` URL inherit the origin of the document that opened that URL, since these types of URLs do not explicitly contain information about the server of origin. For example, `about:blank` is often used as a URL of a new, empty popup window into which the parent script writes content (e.g. via the `Window.open()` mechanism). If this popup window were to also contain code, that code would inherit the same origin as the script that created it. `data:` URLs get a new, empty, security context.

 **Note:** Prior to Gecko 6.0, data URLs inherited the security context of the page currently in the browser window if the user enters a data URL into the location bar.

IE Exceptions

Internet Explorer has two major exceptions when it comes to same origin policy

- Trust Zones: if both domains are in highly trusted zone e.g, corporate domains, then the same origin limitations are not applied
- Port: IE doesn't include port into Same Origin components, therefore `http://company.com:81/index.html` and `http://company.com/index.html` are considered from the same origin and no restrictions are applied.

These exceptions are non-standard and not supported in any other browser but would be helpful if developing an app for Windows RT (or) IE based web application.


Changing origin

A page may change its own origin with some limitations. A script can set the value of `document.domain` to its current domain or a superdomain of its current domain. If it sets it to a superdomain of its current domain, the shorter domain is used for subsequent origin checks. For example, assume a script in the document at `http://store.company.com/dir/other.html` executes the following statement:

```
1 | document.domain = "company.com";
```


After that statement executes, the page can pass the origin check with `http://company.com/dir/page.html` (assuming `http://company.com/dir/page.html` sets its `document.domain` to `"company.com"` to indicate that it wishes to allow that - see `document.domain` for more). However, `company.com` could **not** set `document.domain` to `othercompany.com` since that is not a superdomain of `company.com`.

The port number is kept separately by the browser. Any call to the setter, including `document.domain = document.domain` causes the port number to be overwritten with `null`. Therefore one **cannot** make `company.com:8080` talk to `company.com` by only setting `document.domain = "company.com"` in the first. It has to be set in both so that port numbers are both `null`.

 **Note:** When using `document.domain` to allow a subdomain to access its parent securely, you need to set `document.domain` to the same value in both the parent domain and the subdomain. This is necessary even if doing so is simply setting the parent domain back to its original value. Failure to do this may result in permission errors.

Cross-origin network access

The same-origin policy controls interactions between two different origins, such as when you use `XMLHttpRequest` or an `` element. These interactions are typically placed into three categories:

- Cross-origin *writes* are typically allowed. Examples are links, redirects and form submissions. Certain rarely used HTTP requests require `preflight`.
- Cross-origin *embedding* is typically allowed. Examples are listed below.
- Cross-origin *reads* are typically not allowed, but read access is often leaked by embedding. For example, you can read the width and height of an embedded image, the actions of an embedded script, or the  availability of an embedded resource.

Here are some examples of resources which may be embedded cross-origin:

- JavaScript with `<script src="..."></script>`. Error messages for syntax errors are only available for same-origin scripts.
- CSS with `<link rel="stylesheet" href="...">`. Due to the [relaxed syntax rules](#) of CSS, cross-origin CSS requires a correct Content-Type header. Restrictions vary by browser: [IE](#), [Firefox](#), [Chrome](#), [Safari](#) (scroll down to CVE-2010-0051) and [Opera](#).
- Images with ``. Supported image formats include PNG, JPEG, GIF, BMP, SVG, ...
- Media files with `<video>` and `<audio>`.
- Plug-ins with `<object>`, `<embed>` and `<applet>`.
- Fonts with `@font-face`. Some browsers allow cross-origin fonts, others require same-origin fonts.
- Anything with `<frame>` and `<iframe>`. A site can use the X-Frame-Options header to prevent this form of cross-origin interaction.

How to allow cross-origin access

Use [CORS](#) to allow cross-origin access.

How to block cross-origin access

- To prevent cross-origin writes, check for an unguessable token in the request, known as a [Cross-Site Request Forgery \(CSRF\)](#) token. You must prevent cross-origin reads of pages that know this token.
- To prevent cross-origin reads of a resource, ensure that it is not embeddable. It is often necessary to prevent embedding because embedding a resource always leaks some information about it.
- To prevent cross-origin embedding, ensure that your resource cannot be interpreted as one of the embeddable formats listed above. The browser does not respect the Content-Type in most cases. For example, if you point a `<script>` tag at an HTML document, the browser will try to parse the HTML as JavaScript. When your resource is not an entry point to your site, you can also use a CSRF token to prevent embedding.

Cross-origin script API access

JavaScript APIs such as `iframe.contentWindow`, `window.parent`, `window.open` and `window.opener` allow documents to directly reference each other. When the two

documents do not have the same origin, these references provide very limited access to `Window` and `Location` objects, as described in the next two sections.

To communicate further between documents from different origins, use `window.postMessage`.

Specification: <https://html.spec.whatwg.org/multipage/browsers.html#cross-origin-objects>.

Window

The following cross-origin access to `Window` properties is allowed:

Methods

<code>window.blur</code>
<code>window.close</code>
<code>window.focus</code>
<code>window.postMessage</code>

Attributes

<code>window.closed</code>	Read only.
<code>window.frames</code>	Read only.
<code>window.length</code>	Read only.
<code>window.location</code>	Read/write.
<code>window.opener</code>	Read only.
<code>window.parent</code>	Read only.
<code>window.self</code>	Read only.
<code>window.top</code>	Read only.
<code>window.window</code>	Read only.

Some browsers allow access to more properties than the specification allows.

Location

The following cross-origin access to `Location` properties is allowed:

Methods

`location.replace`

Attributes

`URLUtils.href`

Write only.

Some browsers allow access to more properties than the specification allows.

Cross-origin data storage access

Access to data stored in the browser such as `localStorage` and `IndexedDB` are separated by origin. Each origin gets its own separate storage, and JavaScript in one origin cannot read from or write to the storage belonging to another origin.

Cookies use a separate definition of origins. A page can set a cookie for its own domain or any parent domain, as long as the parent domain is not a public suffix. Firefox and Chrome use the [Public Suffix List](#) to determine if a domain is a public suffix. Internet Explorer uses its own internal method to determine if a domain is a public suffix. The browser will make a cookie available to the given domain including any sub-domains, no matter which protocol (HTTP/HTTPS) or port is used. When you set a cookie, you can limit its availability using the Domain, Path, Secure and Http-Only flags. When you read a cookie, you cannot see from where it was set. Even if you use only secure https connections, any cookie you see may have been set using an insecure connection.

See also

- Same-origin policy for file: URIs
 - [Same-Origin Policy at W3C](#)
-

- Author(s): Jesse Ruderman