November 26, 2024 ~9 min read

SECURITY ASP.NET CORE CORS

Cross-Origin-Resource-Policy: preventing hotlinking and XSSI attacks

Understanding cross-origin security headers - Part 2

This is the second post in the series: Understanding cross-origin security headers.

> Part 1 - Cross-Origin-Opener-Policy: preventing attacks from popups
>
> Part 2 - Cross-Origin-Resource-Policy: preventing hotlinking and XSSI attacks (this post)

In this post I describe how some cross-origin resources can be loaded without restriction, despite the same-origin policy. This can open your site up to vulnerabilities that leak sensitive data, as well as enable hotlinking to your resources. I show how the `Cross-Origin-Resource-Policy` can block these cross-origin requests, how each of the available header values behave, and how the header differs from CORS headers.

- Why do we need the `Cross-Origin-Resource-Policy` header?
- Locking down cross-origin requests with `Cross-Origin-Resource-Policy`
- Understanding the difference between same-origin and same-site
- Testing out `Cross-Origin-Resource-Policy`
- CORP vs CORS: why do we need both?
- The relationship with `Cross-Origin-Embedder-Policy`

Why do we need the `Cross-Origin-Resource-Policy` header?

One of the first things I thought when first reading about the `Cross-Origin-Resource-Policy` was: why? We *already* have the same-origin policy that restricts cross-origin network access, right? Why do we need another header?

Well, it turns out, that although the same-origin policy applies some pretty strict restrictions when it comes to JavaScript APIs and requests using `fetch()`, the restrictions are much looser when it comes to *embedded* resources.

Embedded resources include things like scripts referenced in a `<script>` element and images linked in an `<img>` tag. MDN provides a long list of resources which *can* be accessed cross-origin. In each of these cases, the same-origin policy provides no protection; the browser will happily load them from a cross-origin URL by default. That includes:

- Images linked in `<img>` elements.
- Media linked in `<video>` and `<audio>` elements.
- JavaScript files referenced in `<script src="…"></script>`.
- CSS files referenced in `<link rel="stylesheet" href="…">` (as long as the stylesheet has the correct MIME type).

In each of these cases, the same-origin policy doesn't protect you by default on *either* side. That means if an attacker manages to add an `<img>` or `<script>` to your site, they can load links from any origin. The

`Cross-Origin-Resource-Policy` doesn't directly help you in this scenario; you need a `Content-Security-Policy` or `Cross-Origin-Embedder-Policy` for that.

From the other direction, there's nothing in the same-origin policy to stop people linking to images hosted on *your* site and displaying them in *their* site. This is often called hotlinking and is problematic as it makes it easier to plagiarise content, but also places the egress and server costs on *your* site, as the host, instead of the embedding site.

As well as hotlinking, there's a class of v
Script Inclusion (XSSI), in which a script
inside an attackers site, which can leak
attacker. That's on top of speculative side-channel attacks like Spectre which rely on cross-origin access.

The `Cross-Origin-Resource-Policy` header provides a way to protect you from these attacks and vulnerabilities.

Locking down cross-origin requests with `Cross-Origin-Resource-Policy`

The `Cross-Origin-Resource-Policy` (CORP) header is a security header you can return in your responses that signals to the browser whether or not the resource is allowed to be embedded in a document.

> **Note that you might be wondering why you need CORP when we already have `Cross-Origin-Resource-Sharing` (CORS). I go into more detail later, but the short-answer is that CORP only applies to `no-cors` scenarios i.e. when CORS *doesn't* apply.**

There are currently three possible values for the `Cross-Origin-Resource-Policy` header:

- `same-origin`—the browser should **reject** cross-origin responses that are returned with a header with this value.
- `same-site`—the browser should **reject** cross-site responses that are

`same-site` — the browser should **reject** cross-site responses that are returned with a header with this value. This is less restrictive than `same-origin`; I describe the difference between same-site and same-origin in the following section.

- `cross-origin` —the browser should **allow** cross-origin responses. By default, this behaves similarly to when no CORP header is returned, but it also interacts with the `Cross-Origin-Embedder-Policy` header, as I describe later.

> Note that [there's a bug in Chrome](#) wh
> `Resource-Policy` can break PDF ren
> to read past the first page of some P

I'll walk through some examples of these shortly, but first I'll clarify the difference between `same-origin` and `same-site`.

Understanding the difference between same-origin and same-site

The CORP header can be set to either `same-site` or `same-origin`. Site and origin are two similar concepts, but it's important to understand the differences. Two URLs are considered to be "same-**site**" if they:

- Have the same *scheme* i.e. `http` or `https`
- Have the same *domain* i.e. `example.com`, `andrewlock.net` or `microsoft.com`

They *don't* need to have the same `port` or `subdomain`.

Two URLs are considered to be "same-**origin**" if they

- Have the same *scheme* i.e. `http` or `https`
- Have the same *domain* i.e. `example.com`, `andrewlock.net` or `microsoft.com`

- Have the same *subdomain* i.e. `www.`
- Have the same *port* (which may be implicit) i.e. port `80` for `http` and `443` for `https`

So `same-origin` is more restrictive than `same-site`. Note that neither concept cares about the path, querystring, or fragment of the URL.

If we use the URL `http://www.example.org` as an example and compare against variations, you can see the difference more clearly.

URL	Description	same-site	same-origin
`http://www.example.org`	Identical U		
`http://www.example.org:80`	Identical U port)		
`http://www.example.org:5000`	Different port	✅	❌
`http://example.org`	Different subdomain	✅	❌
`http://sub.example.org`	Different subdomain	✅	❌
`https://www.example.org`	Different scheme	❌	❌
`http://www.example.evil`	Different TLD	❌	❌

This becomes important when thinking about the `Cross-Origin-Resource-Policy` header. If you serve resources from a dedicated subdomain, `cdn.example.org` for example, and you want them to be embedded on `example.org` or `www.example.org`, then you will need to make sure the resources are returned with a `same-site` value. If, however, you only embed resources from the exact same origin, then using `same-origin` provides greater security.

Testing out `Cross-Origin-Resource-Policy`

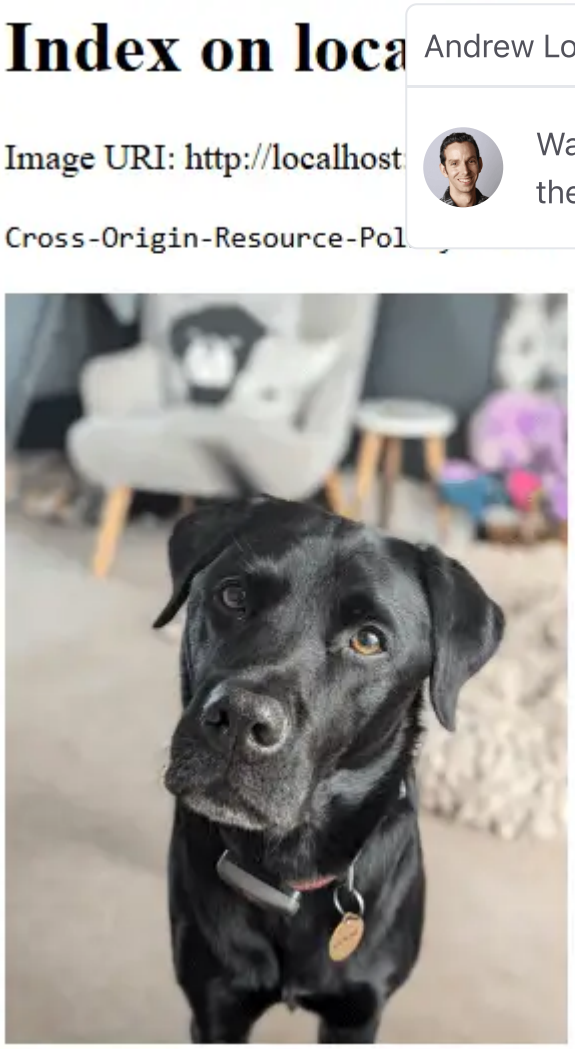
I've described the various possible values for `Cross-Origin-Resource-Policy`, but I find it helps to see the header in action.

To test out the different policy values, I created two ASP.NET Core apps, and used my *NetEscapades.AspNetCore.SecurityHeaders* NuGet package to configure the `Cross-Origin-Resource-Policy` for each. The

Index page loads an image using an `<img>` tag:

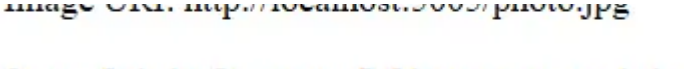
```
<img src="http://localhost:5005/photo.jpg"
    alt="Unable to load image"/>
```

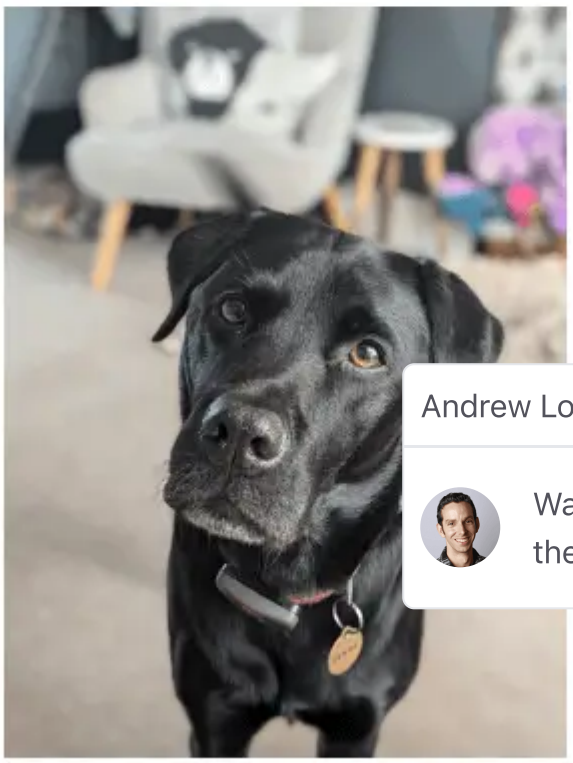
By default, without any `Cross-Origin-Resource-Policy` value, the browser loads the image, even in cross-origin scenarios:



> This is a cross-origin scenario because the root document is hosted at http://localhost:5011, while the embedded image is hosted on a site at a different port, http://localhost:5005.

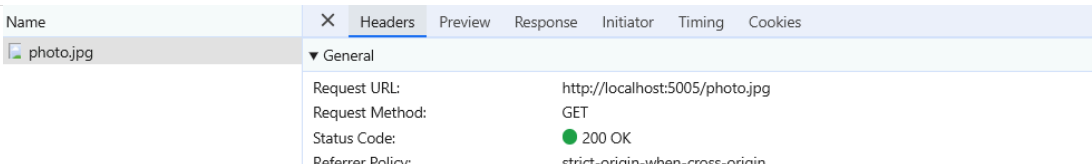
Similarly, if you use the `cross-origin` header value, the image still loads:

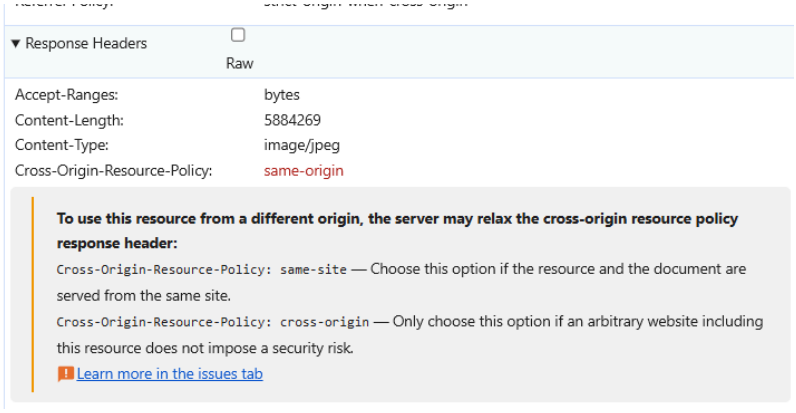




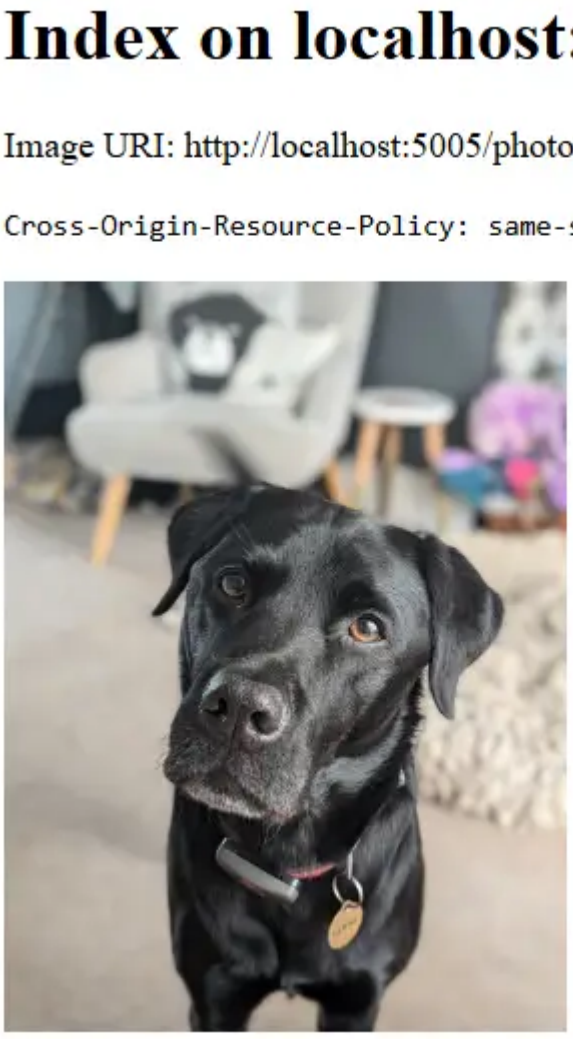
If, however, the image returns a `Cross-Origin-Resource-Policy` header with the value `same-origin`, then the image is blocked:

If we take a look at the dev tools, we can see that the request was made successfully, and returned a `200 OK` response, but the browser rejected the content, because it was a cross-origin request with a `same-origin` CORP header:





The dev tools explanation of why the rec
explaining why the request failed, and po
we can switch the resource to use `same-`
differ in the port number, which is not pa
that change, the image can be embedde



The possible values of the `Cross-Origin-Resource-Policy` header are pretty simple to follow, but it took me a long time to understand why we need the header at all when there are many other security headers that

seem to do similar things.

CORP vs CORS: why do we need both?

One of the common questions about the `Cross-Origin-Resource-Policy` header is: why do we need it, when we already have `Cross-Origin-Resource-Sharing` (CORS)?

CORS is used to *allow* cross-origin reque
blocked by the existing same-origin poli
termed `cors` mode requests. Adding C
security by *allowing* more requests.

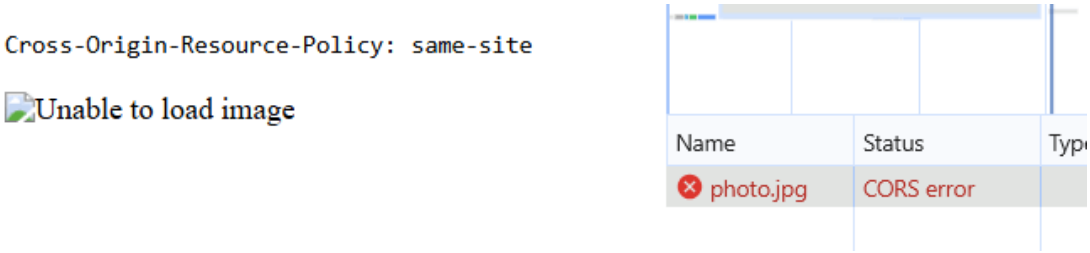
In contrast, the CORP header *increases* security by *blocking* requests that would otherwise have been allowed *despite* the same-origin policy protections. It applies to a different set of requests than CORS, in that it only applies to `no-cors` mode requests.

One difficulty is understanding *exactly* what constitutes a `no-cors` or `cors` request can be tricky. For example, in this post I've shown that resources loaded by `<img>` are typically `no-cors` requests. CORP applies to these requests, hence the behaviour I showed.

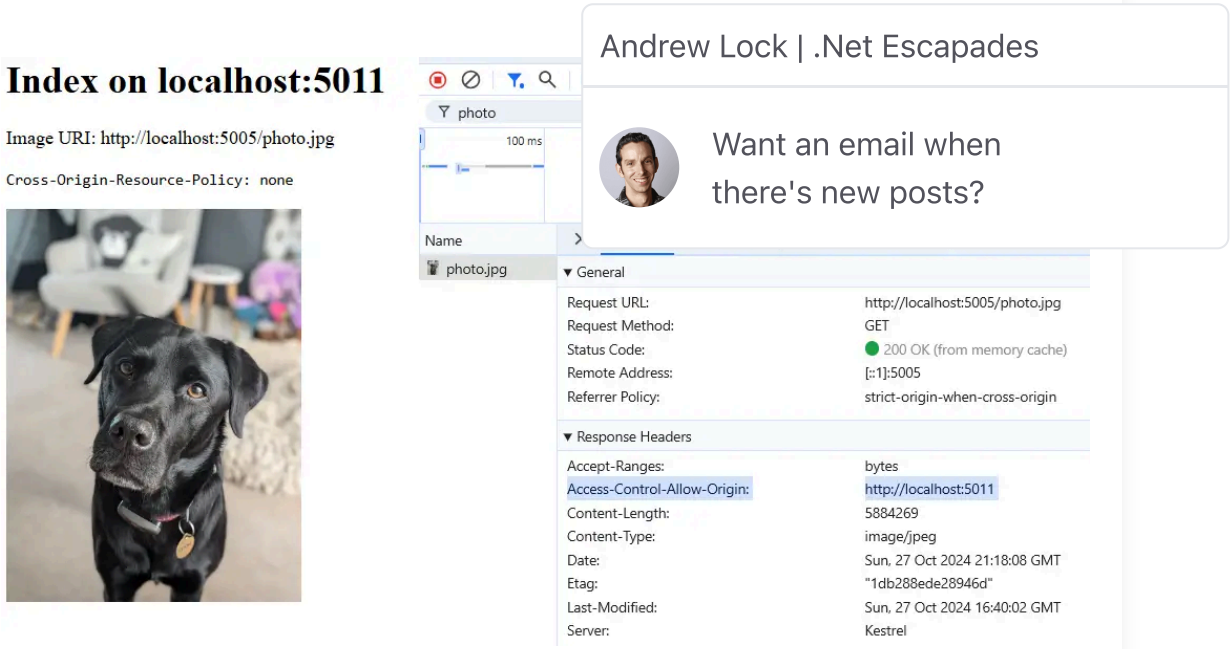
However, you can turn an `<img>` link into a `cors` request by adding the `crossorigin` attribute to an `<img>` tag:

```
<img src="http://localhost:5005/photo.jpg"
    alt="Unable to load image" crossorigin/>
```

With this small edition, the CORP header no longer applies. Even if the CORP header would have allowed the request, the addition of `crossorigin` turns the request into a `cors` request, which will be blocked by default:



To allow the request, you must add CORS headers, returning `Access-Control-Allow-Origin` of either `*` or the embedding origin (`http://localhost:5011`), as in the following image:



As you can see in the above image, in this scenario the CORS header is all that's required; no CORP header is applied and it does not affect the loading of the image.

The relationship with `Cross-Origin-Embedder-Policy`

Many security headers have a "default" value that is equivalent to not applying the header at all. In many cases the `cross-origin` value serves this purpose for the `Cross-Origin-Resource-Policy` header, in that it doesn't block cross-origin requests, which is the same behaviour your see if no header had been applied.

The difference is that the `cross-origin` CORP value *explicitly* allows cross-origin requests (for `no-cors` mode requests). This is important when it comes to the `Cross-Origin-Embedder-Policy` (COEP) which

differentiates between responses which have a CORP header and those that don't. The COEP header can be configured to *require* that a resource returns a CORP header. In the next post we'll look in more detail at the COEP header and how it interacts with both CORP and COEP.

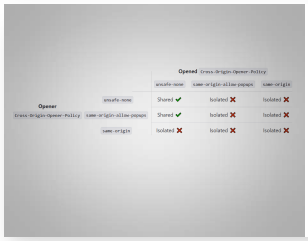
Summary

In this post I described some scenarios where cross-origin resources can be loaded without restrictions, such as w
embedded in a document using `<img>`
your site to cross-site script inclusion (X
others to hotlink to your resources.

The `Cross-Origin-Resource-Policy` (CORP) header enables you to block the browser from loading these resources in cross-origin or cross-site scenarios. For maximum protection, apply a CORP header with the value `same-origin`. If you expect your resources to be loaded from a different origin, use the `same-site` value if possible.

THIS SERIES

Understanding cross-origin security headers



PREVIOUS

Cross-Origin-Opener-Policy: preventing attacks from popups: Understanding cross-origin security headers - Part 1

2 comments – *powered by giscus*

Oldest Newest