Notes on the threat model of *cross-origin isolation*

aaj@, December 2020

If you're only interested in a high-level list of takeaways from this document, see the tl;dr section.

Background

The cross-origin isolated mode (COI) can be enabled by a document by setting Cross-Origin Opener Policy (COOP) and Cross-Origin Embedder Policy (COEP). At a high level, it's meant to enforce a set of security restrictions which allow the browser to guarantee that the document can't load or embed authenticated cross-origin data without an explicit opt-in. In exchange, the browser can give the document access to APIs which would otherwise be unsafe to expose on the web because they would enable the document to leak cross-origin data.

The full details are beyond the scope of this doc (COOP+COEP explained and Safely reviving shared memory attempt to explain the model and the rationale behind it; web.dev/coop-coep and web.dev/why-coop-coep have developer guidance), but, in a nutshell:

- COOP ensures the document cannot share a browsing context group with top-level documents outside of its own origin, which allows the browser to separate it from non-cooperating documents by putting it in its own renderer process.

- COEP prevents the document from loading authenticated cross-origin resources or frames unless they opt in, and applies transitively to all iframes -- an iframe can only be loaded by a COEP document if it itself opts into COEP.

In practice, security decisions made by developers in conjunction with the cross-origin isolated mode will revolve almost exclusively around COEP and the question of when, and how, to opt resources into being loaded by COI documents.

This document attempts to clarify the consequences of allowing resources to enter COI mode and guide decisions about future platform features that should be gated on cross-origin isolation.

Cross-origin isolation is messy

Cross-origin isolation uses web-facing features to weave a fabric with which we're hoping to cover an emperor who is quite clearly naked. More specifically, in the face of attacks like Spectre, the web platform finds itself in the unadmirable position of needing to mitigate CPU-level vulnerabilities, which should ideally be addressed directly by CPU vendors. As a result, the practical security model of COI is more complex than what we're used to dealing with in the web platform; it is dependent on browser implementation details which would otherwise be transparent to the platform, as well on other security features seemingly unrelated to COI itself.

Each of the following features has significant consequences for the threat model of COI:

1. **The browser's process model, including support for out-of-process iframes** (OOPIFs): Since transient execution attacks can generally read the full memory contents of the browser's renderer process, data in a COI context should be treated as readable by anyone with scripting capability within that renderer. In particular, any cross-origin data which under web rules shouldn't be readable (e.g. iframes or any windows from other browsing context groups), but which is present in the renderer's OS-level process, is at risk of exfiltration. Browser process allocation strategies can have a significant impact on the amount of in-process cross-origin data.

2. **Cross-Origin Read Blocking** (CORB / CORB++): The presence of any default logic which disallows the loading of cross-origin no-cors resources by a COI document reduces the amount of potentially authenticated data which can be exfiltrated by the renderer. This mitigates the risk of misconfigurations, for example when an application mistakenly sets an overly broad CORP policy.

3. **Implementation of networking logic,** including the partitioning of network state and out-of-renderer handling of security-relevant mechanisms, e.g. OOR-CORS. Sharing of network state between non-COI and COI contexts can result in COI documents loading resources which they couldn't directly request. Similarly, any in-renderer implementations of web security logic, such as CORS checks, can put sensitive data not meant to be accessible by a COI document inside a process whose memory can be read by such a document.

4. **Restrictions on loading resources from local networks** (CORS-RFC1918): Under the proposed credentialless mode, resources requested without authentication may be loaded by COI documents. If the browser doesn't prevent requests from crossing the internet↔local network boundary, resources from local devices which use the requester's network location for authentication could leak to COI documents.

As a result, it's not simple to define *the* threat model of COI: the security consequences of a resource being loaded into a COI document will vary between browsers, and can differ even for the same browser on different platforms (e.g. Chrome Desktop vs. Android). Evaluating the security model thus requires keeping in mind the behaviors of major browsers, understanding their future plans (for example, ongoing changes to the process model, including the implementations of Site Isolation or Project Fission), and looking at the behavior from the abstract perspective of the web platform.

However, web developers making decisions for their applications will likely not be able to -- nor should they be required to -- understand the intricacies of each browser's support matrix for the mechanisms which affect the security model of cross-origin isolation. Instead, it seems useful to outline a simpler model that will help developers reason about the consequences of the existence of cross-origin isolation for their applications. This may require accepting temporary insecurity in some configurations under the assumption that in the long term browsers will converge towards a more secure state which restricts the capabilities of COI documents, in a way transparent to web developers.

Cross-origin isolation from the application's point of view

In practice, the major question which developers need to answer is whether they want their resources to be available to cross-origin documents in COI mode. There are currently three mechanisms which may allow a resource to be exposed in this context:

- Cross-Origin Resource Sharing (**CORS**)
- Cross-Origin Resource Policy (**CORP**)
- [Proposed] Credentialless mode

The security models of CORS and the credentialless mode are relatively simple. CORS provides an allowlisted requester full access to the data of the resource (assuming a server-side opt-in which sets `Access-Control-Allow-Credentials: true` and echoes the requesting origin in `Access-Control-Allow-Origin`); this makes COI considerations mostly moot since the full data of the response is exposed to the requester via standard web APIs. In turn, credentialless mode attempts to ensure that the data returned by the server is not sensitive by removing authentication information from the request; responses to requests in credentialless mode generally shouldn't contain authenticated data, making them safe to deliver to a COI document[1].

In contrast, the meaning of CORP isn't as clear-cut. It seems important to review the consequences of setting the header to better understand what it means for developers.

What CORP *does*

The initial proposal for CORP as a resurrection of the From-Origin concept was a relatively simple equivalent of `X-Frame-Options` applied to no-cors subresources. Just like `X-Frame-Options` allows developers to limit iframing to only same-origin documents, CORP lets developers restrict the loading of their resources (e.g. images) to only same-origin or same-site requesters.

However, as a result of the discussions which led to the creation of the Cross-Origin Embedder Policy, CORP acquired additional capabilities. The cross-origin value of CORP could now be used as an opt-in for a resource to be loaded by a cross-origin document which sets a COEP of `require-corp`, which has the effect of allowing the resource to be delivered to a COI context. The semantics of CORP also changed to cover iframes: an iframe can only be embedded by a COI document if it sets both CORP and COEP.

These seemingly small changes complicate the security model of CORP in the following ways:

[1] There are situations where this assumption doesn't hold, for example when the server uses something other than credentials attached to the request to authenticate the user (for example, the user's IP address). These considerations are discussed in more detail here.

- **CORP can both restrict and relax built-in same-origin policy rules.** A CORP of `same-origin` will prevent a resource from being loaded by other websites, whereas a CORP of `cross-origin` acts as explicit permission for the resource to be loaded by a COI document, where it can be efficiently read by the attacker from their renderer's address space. When set to `same-site`, CORP *both* restricts and relaxes SOP rules: it prevents the resource from being loaded cross-site, but at the same time it gives same-site COI documents permission to load the resource and read the resource from their memory.
- **CORP can apply to both in-process and out-of-process data.** While `no-cors` subresources are treated consistently and put in-process in all browsers, CORP (if set in combination with COEP) can be used by iframes as an opt-in for being embedded by a COI document. Since browser process models differ, with some browsers shipping out-of-process iframes and others not, the security consequences of setting CORP+COEP vary significantly. In a browser with OOPIFs delivering an iframe to a cross-site COI document doesn't expose its contents to the embedder; in a browser without OOPIFs, the iframe and its contents are placed in an attacker-controlled renderer and can leak.

As a result, reasoning about the security effects of setting CORP can be challenging for developers. Complicating the matter further, there is no single agreed-upon definition of the meaning of CORP which would help shape developers' mental models and help guide their decisions.

What CORP *means*

There are two alternative (and, arguably, conflicting) views on the semantics of CORP:

1. It allows **embedding only**: The requester can load the resource in no-cors mode, but this permission applies solely to embedding; the requester cannot learn more about the contents of the resource.
2. It allows **legibility**: The requester, having been granted explicit permission to load the resource in their COI document, can read the data and metadata of the resource.

Both of these interpretations pose their own problems.

The embedding-only view is belied by reality. We know, with certainty, that a resource which opts itself into being loaded by a COI document *can* be read by that document from its renderer's memory on many configurations. While this is not a universal truth (some users could have CPUs which don't speculate, or operating systems and browsers which mitigate transient execution attacks), in practice, developers who set CORP for their resources need to consider the fact that a substantial number of their users will have such vulnerable environments. A developer making a decision about setting CORP on a resource cannot simply ignore the fact that this resource could be leaked for a substantial fraction of their users, even if that fraction decreases over time, and even if the attack doesn't operate fully at the web layer and requires vulnerabilities to exist at a lower level.

Since CORP and COI were designed specifically to align the web's security boundaries to a world where attacks on lower layers of the computing stack are possible, we cannot ignore the practical consequences of such attacks.

The legibility view runs into different concerns. First, it needs to grapple with the issue of CORP `same-site`: developers who set this value to prevent cross-site embedding are unwittingly exposing their resource to potentially untrusted same-site COI documents. (Note that `same-site` is the only value of CORP that causes problems here; resources are by definition fully legible to any same-origin requesters, and CORP cross-origin acts as an explicit opt-in for arbitrary embedding, so there's no risk that a developer would set this value for a security reason.)

Second, if CORP implies legibility it becomes a more "lightweight", and consequently easier to misuse, equivalent to CORS. This introduces the risk that developers who mistakenly set CORP `cross-origin` on their authenticated resources will expose them to cross-origin requesters.

Arguably, out of these two models, the latter one where CORP confers legibility is the only one that is defensible from a security standpoint. The embedding-only view will inevitably fall prey to demonstrations of how the contents of resources which only purport to allow embedding can be revealed to COI documents via improved techniques to exploit lower-level vulnerabilities. Fortunately, the problems with the legibility interpretation are solvable and introduce less inconsistency into the platform than it may otherwise seem.

Why CORP implying legibility is acceptable[2]

It is a surprising claim that a mechanism which can be viewed as only allowing embedding can be (re-)interpreted as implying legibility without substantial loss of security. However, in this case the design of CORP provides important, if implicit, protections that reduce the impact of CORP misconfigurations.

Subresources

An important facet of CORP is that it applies only to subresources requested in no-cors mode. This limits any CORP-based opt-in that could expose the resource to COI documents to a small number of resource types meant for cross-origin embedding and not blocked by CORB/CORB++; this means resources loaded as `<img>`, `<script>`, `<link rel="stylesheet">`, `<audio>`, `<video>` and, in some cases, `<object>`/`<embed>`.

While these resource types can contain authenticated data, there are practical reasons for why it's rare for them to do so. Specifically:

[2] Note that this document doesn't suggest creating new web-level APIs to grant the embedder the ability to read CORP-enabled resources; it merely argues that this would be consistent with the security model of COI.

- Information present in JavaScript resources loadable cross-origin leaks to the loading document via side effects of script execution, leading to issues such as XSSI / JSON hijacking. As a result, the presence of authenticated data in script-like responses is generally a vulnerability in the application, and developers have to avoid this pattern.
- Stylesheets similarly leak their contents to the loading document; the document can use Window.getComputedStyle to read the calculated values for a given element which will contain any CSS rules defined in a cross-origin stylesheet.
- As media formats, `<img>`, `<audio>` and `<video>` resources are not frequently dynamically composed of authenticated data. Interestingly, situations where these resources do contain user data already result in minor information leaks due to the cross-origin readability of media metadata. This includes values which are exposed by default such as <img>'s `height`, `width`, `naturalHeight`, `naturalWidth`, <video>'s `videoHeight`, `videoWidth`, `duration`, and <audio>'s `duration`, `buffered` or `seekable` (details).

As a result, unlike other content types, such as HTML, JSON, XML, plaintext, `application/protobuf` or other API responses, resources meant for loading in no-cors mode are less likely to contain sensitive secrets that can leak if the resource mistakenly sets CORP and ends up in a COI context.

The most prominent counterexample to this are static-but-authenticated media resources, such as image thumbnails of documents or the user's private videos; in these cases the contents of the resource can be sensitive and their exposure result in a high-risk information leak. However, for the vulnerability to occur, the application must meet the following conditions:

1. Mistakenly set CORP cross-origin on the resource.
2. Explicitly disable any built-in protections for cookies / storage partitioning, e.g. set its authentication cookies as `SameSite=None`.
3. Return authenticated data in a no-cors resource which:
    a. Has a MIME type not protected by CORB/CORB++.
    b. Doesn't enable any protections against cross-site loads (e.g. CSRF tokens or restrictions using Fetch Metadata).
    c. Is accessible at a non-secret URL known to the attacker.

While there are certainly situations where these conditions can be met, introducing a vulnerability requires both the explicit misconfiguration of CORP and the presence of a fairly uncommon application-level pattern. It's also worth noting that any such instance has already been exploitable irrespective of CORP in browsers which don't gate the use of `SharedArrayBuffer` on COI, such as Chrome Desktop. The lack of examples of successful attacks on such resources serves as an (arguably quite weak) signal that this pattern isn't frequent enough to lead to broad problems.

Finally, we need to grapple with the fact that no matter how we choose to *interpret* CORP, the practical consequences of setting the header result in a situation where the resource *is* loadable by a COI document and its contents are readable cross-origin.

If CORP `cross-origin` implies legibility, haven't we created a backdoor CORS under a new name?

The legibility argument equates the security consequences of setting CORP cross-origin to opting the resource into credentialed CORS (`ACAO: <Origin> + ACAC: true`). It's therefore reasonable to ask whether we shouldn't just get rid of CORP as a COI opt-in and require developers to use CORS instead; after all, the latter is a more established mechanism and makes the security consequences clearer to the developer.

However, CORP has important advantages over CORS in this context:

1. **Security**:
    - ❏ **Reduced scope**: As CORP applies only to no-cors resources, an overly broad CORP setting can at most expose resource types meant to load cross-origin. In comparison, misconfigured CORS logic can reveal all of an application's data to other origins.
    - ❏ **Protections for iframes**: Using CORP+COEP as an opt-in for having an iframe embedded in a COI context prevents the embedder from reading the contents of the frame in browsers which ship OOPIFs and CORB (because the frame will be placed in a separate renderer process and the HTML of the frame cannot be loaded as a subresource by cross-origin documents). Using CORS would expose the markup of the frame and any data it contains.
    - ❏ **No direct exposure of data at the web level**: An opt-in via CORS makes a subresource explicitly readable to the requester as `response.text()` and related Fetch APIs. With CORP, even if we treat the data as potentially legible to a COI context, in practice an attack may have preconditions that could be difficult to meet for an attacker: it can require a substantial amount of time, it may become less reliable due to unrelated activity on the user's system, or may be prevented on future CPUs which patch transient execution bugs. Avoiding the direct exposure of data seems preferable in this case.

2. **Simplicity for developers:** Setting CORP as an application-wide HTTP header is easier for developers than configuring CORS. It also doesn't require changes to the sites which load the resource (e.g. adding the `crossorigin` attribute).
    - ❏ The extra controls in CORS (preflights, `Access-Control-Allow-*` headers, etc.) are certainly warranted for a general mechanism which gives the caller power to craft arbitrary requests and read responses. But they're of less value in the case of no-cors requests, which can't be modified, and where the mechanism doesn't provide an explicit API to access the contents of the response.

Because of this, CORP cross-origin seems like a better fit for no-cors resources and iframes which want to be loadable by COI contexts.

Iframes

While subresources are handled consistently by browsers and always reach the renderer process of their loading document, in the case of iframes differences in process allocation strategies cause the security models to diverge.

In browsers with OOPIFs, a cross-site iframe is separated into its own renderer, protecting it and its resources from attacks by a COI embedder. In browsers without OOPIFs, opting an iframe into COI embeddability by setting COEP+CORP exposes the contents of the frame and any same-origin resources it loads to the embedder.

As a result, in non-OOPIF browsers the security consequences of opting a document into COI embeddability are much more significant than for no-cors subresources. They apply to resource types that tend to contain more sensitive data, such as HTML, and apply transitively to any same-origin resources loaded by the COI-embeddable document (COEP doesn't require same-origin resources to opt in via CORP/CORS).

This poses a problem for developers of iframe-based widgets: they cannot permit their widgets to be embedded on browsers with OOPIFs without at the same time exposing them to attacks in non-OOPIF browsers. These developers will likely have to make one of the following choices:

1. **Forego COI altogether**, preventing the widget from being used by cross-origin isolated documents. This may be a blocker for the adoption of COI by the users of such widgets.
2. **Gate sensitive functionality on OOPIF support**. In some cases, widget authors can apply user-agent sniffing and treat requests from non-OOPIF browsers as unauthenticated (for example, display an embedded video without the option to save it to the user's playlist).
3. **Allow their resources to be embedded by a COI document in any browser**. In some cases, even an iframe containing authenticated data may find it acceptable to expose its contents to the embedder if the data isn't particularly sensitive. For example, ad networks or CAPTCHA widgets seem likely to consider opting into COI to be an acceptable trade-off.

A similar concern exists in the opposite direction: COI documents expose their contents to their COI-enabled iframes in non-OOPIF browsers. However, this problem seems less severe because iframing a cross-site document is an intentional decision on part of the application and often indicates a certain level of trust in the embedded content. In situations where applications embed fully untrusted context, they can use the iframe sandbox to limit the frame's capabilities.

There are two considerations that ameliorate these problems:

- In the medium term, most browsers which support cross-origin isolation will ship OOPIFs. This will reduce the number of users in whose browsers COI-enabled iframes could leak.

- The use of authenticated iframe-based widgets is rarer than that of cross-origin no-cors subresources. The number of developers that will have to address this problem is relatively low, increasing the likelihood that we can guide them towards sensible implementations.

Importantly, the risks of COI in the context of iframes apply only when a response sets *both* CORP and COEP, as both are required as an opt-in for embeddability. This means that sites can safely set CORP for most of their resources without worrying about non-OOPIF browsers. Only when they set out to enable COEP on their documents do they have to pay attention to their configuration of CORP and `X-Frame-Options` / `frame-ancestors` to prevent undesired embedding.

The trickier issues

If resources use CORP to opt into being loaded by a context in which they're fully legible, can we treat CORP as giving the requester/embedder the ability to read the resource?

As becomes apparent from the discussion above, even when we interpret CORP as conferring legibility, it is not an *unqualified* legibility. This is easiest to see in the example of iframes: allowing embedding into a COI context in an OOPIF browser doesn't reveal any information about a cross-site iframe to the embedder.

A consistent way to treat CORP is to see it as permission to expose *at most* as much data as the browser's process model reveals to the embedder as a result of loading the resource. Under this view, opting into sharing a resource via CORP doesn't guarantee that the embedder will always be able to read its contents. For example, in a hypothetical scenario of a future browser with out-of-process image rendering the embedder would no longer be able to access the contents of an image in its memory, although it could likely still learn some metadata about the image, such as its dimensions. In this case, we should no longer treat the presence of CORP as if it granted the embedder legibility of the resource because that would unnecessarily expose additional information.

This, of course, poses problems from the web platform's point of view because it changes what CORP means in different browsers and will change in time as browsers' process models evolve.

A potential compromise that reduces the platform's dependence on browser implementation differences is to treat CORP cross-origin as an opt-in to reveal some resource metadata (such as dimensions or size), but not resource contents, at the web level. This isn't strictly necessary -- the platform could decide to limit the role of CORP to control embedding and COI-embedding without using it to reveal any additional information. However, no-cors resource metadata is both readable by COI contexts today (under the legibility argument) and difficult to conceal from embedders in the long term, even in the face of browser process model improvements; overloading CORP to expose it at the web level doesn't reveal more information than is already available to embedders.

This is simultaneously an argument against providing web-level APIs that reveal the full contents of a CORP cross-origin resource to its requester. Since browsers may change how they process subresources (hypothetically, implement "out-of-process images"), we don't want to commit to disclosing resource data when it would otherwise be protected by the browser's process model.

It's important for any information about CORP-enabled resources that we decide to reveal to the web to not cause a security regression by providing more information to the embedder than is readable today. Specifically, since some browsers ship OOPIFs today and setting CORP for iframes doesn't reveal any information about their contents, we shouldn't interpret CORP as an opt-in to give the embedder of an iframe any additional information at the web level.

However, for CORP cross-origin subresources, all of which are legible by COI contexts today, there is no loss of security if the browser was to expose their metadata to the embedder. Since many use cases for revealing metadata are focused on subresources rather than iframes, treating CORP as an opt-in only in the case of subresources seems like a reasonable possible compromise.

Solving the CORP same-site problem

Recall that while CORP same-origin and cross-origin are unambiguous (they respectively restrict and relax how a resource can be loaded), CORP same-site is problematic: it appears to be a security feature to restrict embedding, but also opens up a resource to same-site COI embeddability, leaking it to potentially less secure origins within the site boundary.

To address this, we need to decide whether CORP same-site should be safe by default and restrict embedding without acting as an opt-in for COI, or whether it should follow the "CORP controls COI embeddability" model and allow same-site COI documents to load the resource. Then, we should create a separate switch that allows CORP same-site resources to enable the other, non-default behavior, depending on what the default is.

This is a fairly clear trade-off between security and developer convenience.

On the one hand, the non-secure CORP same-site behavior is likely not a substantial practical concern; it's aligned with browsers' process model (today, no browser ships origin-level process isolation, allowing renderer compromises of less sensitive origins within the site boundary to read data on more sensitive origins). It's also somewhat unlikely that regular web bugs such as XSS in same-site origins would allow attacks on resources with CORP unless the vulnerable endpoint legitimately enables COI (an attacker with an XSS cannot put the attacked document in COI mode without the ability to set HTTP headers). Finally, this option avoids a backwards-incompatible change for developers which already use CORP same-site as an opt-in for resources meant to be loaded in COI documents.

On the other hand, the non-secure behavior risks introducing unexpected vulnerabilities by making sensitive data susceptible to same-site exfiltration. In some ecosystems, where less secure origins share the site boundary with sensitive applications, this can be a substantial concern, especially because developers currently cannot limit the list of same-site origins which can embed a given resource. This approach also introduces a new site-based security boundary, which is undesirable.

Whatever we decide to do by default, it seems important to give developers the ability to control this behavior. This could be done either by adding another CORP value (e.g. `same-site-allow-loading-into-coi` or `same-site-disallow-loading-into-coi`, depending on what we decide for the default behavior), or by adding an optional boolean to the same-site value (`same-site; allow-loading-into-coi`).

Shouldn't we just split out CORP cross-origin into a different header?

The permissive function of CORP cross-origin as a COI opt-in is different in kind from the restrictive function of CORP same-origin and same-site; this will be even more pronounced if we decide to change the behavior of CORP same-site to not serve as an opt-in for COI. As such, it seems reasonable to consider splitting the cross-origin isolation opt-in bit into a separate header ("`X-Bikeshed-Allow-Loading-Into-COI`", "`X-Bikeshed-Not-Personalized`", or similar), and keeping CORP as a purely restrictive mechanism.

While this may make the function of CORP itself simpler, it would carry the cost of increasing complexity for developers who would need to understand additional concepts beyond "is this resource meant to be loaded only by my own application", e.g. understand the model behind COI or the notion of resource personalization.

Arguably, the main function of CORP -- as a mechanism to control the loading of no-cors resources -- is a fairly close match for the purpose of a COI opt-in served by CORP cross-origin today. That's because, in practice, the intent to explicitly declare a resource to be loadable by requesters outside of the application's own origin ("*any origin (both same-site and cross-site) can read the resource*") largely overlaps with the intent to declare a resource as unpersonalized/non-sensitive.

A split into separate headers also seems less valuable in the specific context of no-cors resources because, as discussed above, these resources are already relatively less likely to contain sensitive data. If, for the majority of resources allowing cross-origin loads implies that a resource isn't highly sensitive (as seems to be the case for predominantly static media resources, scripts or stylesheets), it appears less useful to move the opt-in to a separate header.

As a practical matter, while we can certainly rename the COI opt-in and detach it from CORP, it doesn't seem like it would have significant benefits.

The future

It seems somewhat gratuitous to devote so much attention to the semantics of a set of fairly infrequently used HTTP headers; does this really require as much consideration?

The main reason it's important to iron out the behavior and principles behind cross-origin isolation is that the restrictive mode provided by COI introduces a large shift in the mental model of resource loading on the web which will likely affect platform design decisions in the future. It gives us a path to offer developers features which would otherwise be untenable for security reasons, and to adjust platform defaults to reduce the risks of cross-origin information leaks. In the long term, it can help the platform retain composability without compromising security at a, hopefully, low compatibility cost for developers.

Before we get there, we'll likely need to consider the following issues:

Gating web APIs on COI

In addition to allowing a resource to be loaded by a document with access to features which can be used as high-resolution timers, COI can also be used to gate other features which could otherwise expose sensitive information about cross-origin resources.

Natural candidates for such features include APIs which reveal aggregate information about the page's use of resources such as performance.measureMemory, the amount of generated network traffic, implementation-dependent metadata about the state of the JIT, or detailed information about the document's compliance with its privacy- or other budgets. Under the "CORP implies legibility" argument we could also create COI-gated APIs which explicitly provide additional information about no-cors resources loaded by the page, e.g. include external stylesheet information in `document.styleSheets`, or give developers information about the post-compilation state of their scripts.

Ideally, additional capabilities would be accompanied by corresponding reductions in the resolution of data available to non-COI pages. For example, rather than give COI pages more accurate timing APIs, we could reduce the granularity of timing APIs in non-COI documents. This could also be a promising way to start removing existing information leaks in no-cors resources (e.g. the dimensions of an image or video); in time, access to such properties of cross-origin resources could require a COI context.

Exploiting Spectre without COI

It's worth noting that the use of high-resolution timers to read the renderer's memory is a sufficient but not necessary condition for real attacks. There is strong evidence that small refinements to known attacks will enable the exploitation of Spectre and related issues also with lower resolution timers accessible to pages in non-COI contexts.

Due to this, it seems particularly important to move to a world where isolation is enabled by default.

COI as a future default

In a fortunate turn of events, the number of providers of no-cors resources used cross-origin on the web is significantly lower than the consumers of such resources. Importantly, these entities (CDNs, identity- and widget providers, ad networks) generally have a strong incentive to allow their resources to be broadly loaded, and these resources often don't require authorization.

It seems within reach to work with resource providers to make their resources compatible with the notion of cross-origin isolation, and loadable in COI contexts. In the medium term, this will likely result in a substantial fraction of no-cors resources to be "COI-enabled"; this would allow us to move towards a world with isolation-by-default and slowly increase the pressure on developers whose resources or sites are not COI-enabled.

If we can use COI as a forcing function to make this happen, the labeling of resources meant to be loaded cross-origin can have important positive effects for the ecosystem in the long term.

tl;dr

The main takeaways from this doc, with a mixture of facts and opinion:

1. **The threat model of cross-origin isolation is complex** and depends on browser support for many security features, including its process isolation model (e.g. OOPIFs) and CORB.

   ➢ We have to expect that these implementation considerations, some of which weren't previously exposed to the web, will influence our decisions related to COI. At the same time, we should simplify the story for developers to help them make security decisions about their resources (in particular, their CORP configurations).

2. As a result, **we should treat CORP cross-origin as a permission for a resource to be legible** by cross-origin requesters **within the bounds of the browser's process model**. While this seems dangerous, CORB-exempt no-cors resources are generally relatively unlikely to carry sensitive data which could leak in the event of a CORP misconfiguration.

   ➢ Alternatives such as requiring CORS or creating a separate header for COI opt-in are worse from either a security or compatibility standpoint, or both.

   ➢ A specific practical consequence of this approach is that we may want to treat CORP `cross-origin` on subresources (but not iframes) as an opt-in to allow revealing the resource's metadata, such as size. This is consistent with the security model and simplifies reasoning about the web-observable consequences of COI (it allows us to mostly ignore browsers' process allocation strategies).

3. **The main problematic consequence of the legibility model is with CORP same-site** due to its dual use as both a restrictive and permissive mechanism.

- ➢ We should give developers ways to disambiguate the security function of CORP from its function as a COI opt-in by creating a new CORP value or optional flag.

4. **Cross-origin isolation is important for the web** both due to its possible use as a mechanism to gate new low-level APIs, and potential to address long-standing web information leaks if we can move towards isolation by default.

   - ➢ The security benefit is particularly important in the presence of attacks to exploit CPU-level bugs to read memory without high-res timers, from non-COI contexts.