COOP and COEP explained

Artur Janc, Charlie Reis, Anne van Kesteren *2020-01-03*

Introduction

A major consequence of transient execution attacks for the web is the risk of exposing private authenticated data to untrusted sites via microarchitectural side channels. In the words of the Post-Spectre Threat Model Re-Think:

"We now assume any active code can read any data in the same address space. The plan going forward must be to keep sensitive cross-origin data out of address spaces that run untrustworthy code, rather than relying on in-process checks."

This document outlines how the web can safely re-enable access to APIs which could otherwise be used as high-resolution timers and help mount transient execution attacks, undermining same-origin policy protections. Specifically, it focuses on two features that developers will need to adopt in their applications to make threaded use of `SharedArrayBuffer`: the Cross-Origin Opener Policy (COOP) and Cross-Origin Embedder Policy (COEP). In the future, similar restrictions may be required to unlock other timing APIs and new low-level mechanisms such as fine-grained memory measurements via the `performance.memory` API.

> **Note**: Various aspects of this area have been extensively studied and described in detail in more exhaustive documents, including Long-Term Web Browser Mitigations for Spectre, A Spectre-shaped Web, Post-Spectre Threat Model Re-Think, Isolating Application-Defined Principals, the COEP Introduction, Spilling the beans across origins and the Cross-Origin Opener Policy explainer[1]. This document focuses on the specific combination of COOP and COEP which emerges as a promising solution to this problem.

Summary

Allowing a document to safely use web APIs that allow for side-channel attacks requires the browser to prevent the document from pulling non-cooperating cross-origin data into its own process. This requires:

- Restrictions at the **window** level to ensure that the document cannot share a browsing context group with top-level documents outside of its own origin. This allows the browser to separate it from non-cooperating documents by putting it in its own process.
- Restrictions on loading **resources** to prevent the document from making arbitrary cross-origin requests and having the responses reach the document, unless the requested resource explicitly opts in.

This can be achieved by a combination of COOP and COEP, enabled as follows:

```
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

[1] *They were a result of excellent work by Charlie Reis, Anne van Kesteren, Chris Palmer, Mike West, Nika Layzell, Łukasz Anforowicz, Ryosuke Niwa, and other contributors at Mozilla, Apple, Microsoft and Google.*

This document explains how these mechanisms work and why they provide the security properties that allow a browser to safely enable certain APIs, including high-resolution timers.

Cross-Origin Opener Policy (definition; explainer)

See also the more detailed developer guidance for adopting COOP in Appendix A.

COOP allows a top-level document to break the association between its window and any others in the browsing context group (e.g., between a popup and its opener), preventing any direct DOM access between them. It does so by moving the document to a new browsing context group. The security of this approach stems from the fact that browsers generally create browsing context groups by spawning a new process, ensuring that the document will not be put in memory to which an attacker-controlled document has access.

The browser enforces COOP by consulting the `Cross-Origin-Opener-Policy` header on every top-level navigation, comparing its value (generally, `same-origin`) between the document which initiated the navigation (by following a link or opening a popup) and the target of the navigation. If the COOP values are the same, and the origins of the documents match, the browser will not create a new browsing context group, allowing the documents to interact with each other. Otherwise, if at least one of the documents sets COOP, the browser will create a new browsing context group, severing the link between the documents.

Examples

Origin 1	COOP 1	Origin 2	COOP 2	New browsing context group?	Why?
a.example.org	[none]	b.example.org	[none]	No	No COOP
a.example.org	[none]	b.example.org	same-origin	**Yes**	COOP values don't match and origins don’t match
a.example.org	same-origin	b.example.org	same-origin	**Yes**	Origins don’t match
a.example.org	same-origin	a.example.org	same-origin-allow-popups	**Yes**	COOP values don't match
a.example.org	[none]	a.example.org	same-origin	**Yes**	COOP values don’t match
a.example.org	same-origin	a.example.org	same-origin	No	COOP values match and origins match

Additionally, the browser compares the values of the COEP header (see below) if COOP is same-origin, ensuring that same-origin documents with a COOP of `same-origin` must also have the same COEP in order to be present in the same browsing context group.

Because COOP is defined in terms of browsing context groups, it doesn't apply to iframes; the browser ignores COOP on documents which aren't top-level, and allows iframes to access their cross-origin ancestors even if these ancestors set COOP.

Sites can use `same-origin-allow-popups` to allow popups they open to be in their browsing context group (unless the popup's own COOP prevents this). This value cannot be used in documents which desire threaded access to `SharedArrayBuffer`. To receive such access, the only allowed configuration is **`Cross-Origin-Opener-Policy: same-origin`.** With this COOP the browser can ensure that, with the exception of frames, no cross-origin documents are present in the same browsing context group / process as the current document.

Cross-Origin Embedder Policy (spec)

Enabling COEP prevents a document from loading any non-same-origin resources which don't explicitly grant the document permission to be loaded. The only allowed value for COEP is `require-corp`.

Cross-origin resources can opt in via CORP (with a `Cross-Origin-Resource-Policy` header with a value which includes the requester, e.g., `same-site`, or `cross-origin` for public resources) or with CORS, by responding with appropriate `Access-Control-Allow-*` headers for requests in CORS mode.

COEP also prevents the document from loading any framed documents or workers which don't opt-in by setting:

`Cross-Origin-Embedder-Policy: require-corp`

This protects documents which don't restrict framing — unless a document sets COEP, it cannot be framed by another doc with COEP. It also means that all descendents of a document with COEP will enforce the same restrictions.

The COOP algorithm takes into account the presence of COEP when making a decision about whether to put a top-level window into a new browsing context group upon navigation. This ensures that a document with COEP cannot share a browsing context group with a document that does not opt itself into COEP, even if that document is same-origin.

Allowing COOP + COEP to unlock SharedArrayBuffer

COOP and COEP are independent mechanisms, and can be tested and deployed separately.

COEP ensures that a document can only load resources which explicitly allow themselves to be loaded by setting the right CORP or CORS headers; requiring COEP to be set on iframes recursively propagates resource loading restrictions to all descendent contexts.

COOP applies to top-level windows and forces the creation of a new browsing context group upon a navigation to a different security principal. Setting COOP to `same-origin` ensures that

a document cannot share a browsing context group with any cross-origin windows other than its own descendents.

When set together on a top-level document, COEP and COOP ensure that neither the document nor any of its descendents can load resources which haven't opted in, and that navigations to any top-level document that doesn't both come from the same origin *and* set COEP will be performed in a new browsing context group (and consequently, different process).

This combination of restrictions guarantees that only cooperating resources and same-origin windows with the same constraints can be present in the browsing context group of a top-level document with COOP: `same-origin` and COEP: `require-corp`. Browsers (even those without out-of-process iframes) can put this browsing context group in a separate process, ensuring resources which don't explicitly opt in are not present in the same address space and subject to transient execution attacks. This lets browsers expose high-resolution timers such as `SharedArrayBuffer with postMessage()` and other APIs to COOP+COEP documents without the risk of them being able to leak information about non-cooperating cross-origin resources.

Developers can detect whether the headers are set correctly and they are in a COOP plus COEP environment through the `self.crossOriginIsolated` API.

Appendix A: Developer guidance for adopting COOP

The main behavior change caused by COOP is that the browser moves a top-level window to a new browsing context group, as if the user closed the tab and opened the URL in a new tab. This has no observable effects when the user's interaction with the application is limited to a single window: in the absence of other windows, a top-level navigation from `coop.example.org` to `example2.org` will behave the same regardless of whether the browser creates a new browsing context group. (When COOP takes effect the browser preserves history so that back/forward navigations work as expected in the new browsing context group.)

However, enabling COOP may result in behavior changes whenever the application requires two or more top-level windows to interact because windows separated into different browsing context groups cannot obtain a direct reference to each other. This affects code which interacts with another top-level window by sending it a message via `window.postMessage()`, navigating it (e.g. `window.opener.location.assign()`) traversing its frames (`newWin = window.open(...); newWin.frames[2].name`), or via any other direct DOM reference.

This may happen, e.g., in an integration with an external chat client (opened in a popup which is closed by a click in the main window), or in applications which use OAuth popups for federated sign-in. In these cases applications can either relax their COOP configurations, or refactor their code to remove the need for direct window interactions.

This document lists common scenarios and outlines corresponding suitable COOP values.

Common COOP configurations

1. No top-level interactions with other windows
If an application doesn't depend on direct DOM interactions between top-level windows, it can enable a safe, locked-down COOP of `same-origin`:

```
Cross-Origin-Opener-Policy: same-origin
```

Many self-contained applications fall into this category and can safely enable a strict COOP.

2. Top-level interactions only with *same-origin* windows
If an application interacts only with other top-level windows hosted in its own origin, it can also enable a strict COOP of `same-origin`:

```
Cross-Origin-Opener-Policy: same-origin
```

In this case it's important for the entire application to start serving the COOP header at the same time. If two windows don't set the same COOP value (e.g., if one doesn't serve the header) the browser will put them in different browsing context groups, preventing them from interacting. See also the note about atomicity of COOP configuration updates below.

3. Top-level interactions with *cross-origin* windows
Applications which rely on interacting with cross-origin windows which they create themselves (e.g., OAuth popups) need to use `same-origin-allow-popups`:

```
Cross-Origin-Opener-Policy: same-origin-allow-popups
```

In this configuration, new windows and navigations initiated by the application will be opened in the same browsing context group (unless the new window sets its own COOP to prevent this). Applications which rely on this but want to protect themselves from navigations to untrusted sites, e.g., if they support user-controlled links in sanitized HTML, can set the `rel=noopener` attribute on their outgoing links. They should similarly review their cross-origin popups to make sure that their outgoing links are trusted or also set `rel=noopener`.

This approach can be made more secure by using `same-origin-allow-popups` only for the parts of the site which open cross-origin windows (e.g., a login page with OAuth popups). This requires these endpoints to not depend on top-level interactions with the rest of the application.

4. Hosting popups which need to interact with cross-origin documents
In some cases, applications rely on their documents being opened in a new cross-origin window. Such documents can explicitly opt out of COOP:

```
Cross-Origin-Opener-Policy: unsafe-none
```

These popups can only communicate with a cross-origin opener if the opener does not itself use COOP or sets a COOP of `same-origin-allow-popups`. Such documents should generally avoid accessing any sensitive authenticated data. `unsafe-none` has not much meaning today, but in the future browsers might want to change the default so this is a recommended value if you need the current default for your popup application.

Refactoring patterns affected by COOP
In cases where it's preferable to enable a strict COOP of `same-origin` — e.g., in applications which desire threaded access to threaded `SharedArrayBuffer` — developers may want to rewrite code to not rely on direct interactions between top-level windows. For example, applications which respond to messages from their openers can implement equivalent functionality with the `BroadcastChannel` API.

Enabling COOP in report-only mode (needs design work)

Before enforcing COOP restrictions in production code, it's recommended to enable reporting:

```
Cross-Origin-Opener-Policy[2]: same-origin report-only=[reporting_api_id]
```

This will not cause the browser to create any new browsing context groups upon navigation, but will generate reports upon cross-window interactions which will be blocked if COOP is enforced. Reporting is an important feature because it identifies any application functionality that may break in the process of adopting COOP. It also helps determine which COOP configurations can be enabled without requiring any application-level changes, and surfaces code that may require refactoring in order for the application to be compatible with a `same-origin` COOP.

[2] The exact header name and syntax are under discussion in https://github.com/whatwg/html/issues/4622.

Atomicity of COOP configuration updates

When an application enables COOP or changes its configuration (e.g. from `same-origin-allow-popups` to `same-origin`) it introduces the possibility of responding to two navigation requests from the same user with different COOP values, putting the windows in different browsing context groups and resulting in a temporary failure if the windows expect to interact.

Specifically, two windows from the same origin of which one sets COOP `same-origin-allow-popups` and the other `same-origin` will be separated by the browser into different browsing context groups.

To reduce the compatibility risk, developers may want to first use the report-only mode (if available) to identify any behavior that can be affected by COOP.

Miscellaneous

What's the benefit of deploying anything other than a COOP: `same-origin`?

While COOP: `same-origin` is the only configuration which can safely provide access to threaded `SharedArrayBuffer`, developers may likely want to start with other COOP settings:

1. Deploying COOP protects an application from various classes of cross-origin information leaks (XS-Leaks) and is a useful security feature.
2. Starting with a less strict configuration (i.e., a COOP of `same-origin-allow-popups`) allows progressively improving a policy and refactoring any parts of the application which may need to change.

Outstanding spec issues: Handling of history, including back/forward button and the window.history API, Syntax of the reporting functionality, (mostly done) Deciding about requiring explicit agreement to inherit the opener's browsing context group.

Appendix B: Developer guidance for adopting COEP

Note: This is early guidance based on a preliminary version of the COEP spec.

COEP can be enabled separately from COOP using the following header:

```
Cross-Origin-Embedder-Policy: require-corp
```

When a document sets this header, every cross-origin subresource it loads must permit itself to be loaded by setting an appropriate `Cross-Origin-Resource-Policy` header or pass the CORS check by including an `Access-Control-Allow-Origin` header. Similarly, any iframe (including same-origin frames) embedded by the document must explicitly set its own COEP of `require-corp` to be permitted to load; this ensures that all descendants of a document with COEP are also subject to COEP.

However, these requirements do not apply to top-level navigations or opening popups: a document which enables COEP can open new windows without any restrictions.

Some common COEP deployment scenarios are listed below.

1. Loading only same-origin resources and iframes.

No changes are required to load same-origin resources. Same-origin iframes must allow themselves to be loaded by setting their own COEP header. Note that a document which enables COEP could potentially be embedded by an untrusted site with access to high-resolution timers; to protect themselves, documents should disallow cross-origin framing:

```
X-Frame-Options: SAMEORIGIN (or)
Content-Security-Policy: frame-ancestors 'self'
```

2. Loading same-site resources and iframes.

Any same-site resources must allow themselves to be loaded by setting CORP:

```
Cross-Origin-Resource-Policy: same-site
```

Or, in cases where the resources are also requested by other domains:

```
Cross-Origin-Resource-Policy: cross-origin
```

Alternatively, the resources can be requested via CORS by setting the `crossorigin` attribute:

```
<script src="//othersubdomain.example.org" crossorigin="anonymous" />
```

In this case, the server has to respond with a headers that allow the CORS check to pass, e.g.

```
Access-Control-Allow-Origin: oursubdomain.example.org
```

As above, any same-site frames need to set the COEP header and protect themselves from being embedded by untrusted sites by using CSP `frame-ancestors`.

3. Loading cross-origin resources and iframes

Loading cross-origin or public resources requires them to opt in via one of the following headers:

```
Cross-Origin-Resource-Policy: cross-origin (or)
Access-Control-Allow-Origin: *
```

Given that iframes loaded on a top-level page which enables COOP and COEP may get access to APIs which could allow transient execution attacks against the embedding document in browsers without out-of-process iframes, developers should attempt to minimize the number of cross-origin frames, or reduce their capabilities with the sandbox attribute.