Site Isolation for web developers

By Mathias Bynens
V8 JavaScript whisperer

Chrome 67 on desktop has a new feature called *Site Isolation* enabled by default. This article explains what Site Isolation is all about, why it's necessary, and why web developers should be aware of it.

What is Site Isolation?

The internet is for watching cat videos and managing cryptocurrency wallets, amongst other things — but you wouldn't want `fluffycats.example` to have access to your precious cryptocoins! Luckily, websites typically cannot access each other's data inside the browser thanks to the Same-Origin Policy. Still, malicious websites may try to bypass this policy to attack other websites, and occasionally, security bugs are found in the browser code that enforces the Same-Origin Policy. The Chrome team aims to fix such bugs as quickly as possible.

Site Isolation is a security feature in Chrome that offers an additional line of defense to make such attacks less likely to succeed. It ensures that pages from different websites are always put into different processes, each running in a sandbox that limits what the process is allowed to do. It also blocks the process from receiving certain types of sensitive data from other sites. As a result, with Site Isolation it's much more difficult for a malicious website to use speculative side-channel attacks like Spectre to steal data from other sites. As the Chrome team finishes additional enforcements, Site Isolation will also help even when an attacker's page can break some of the rules in its own process.

Site Isolation effectively makes it harder for untrusted websites to access or steal information from your accounts on other websites. It offers additional protection against various types of security bugs, such as the recent Meltdown and Spectre side-channel attacks.

For more details on Site Isolation, see our article on the Google Security blog.

Cross-Origin Read Blocking

Even when all cross-site pages are put into separate processes, pages can still legitimately request some cross-site subresources, such as images and JavaScript. A malicious web page could use an **<img>** element to load a JSON file with sensitive data, like your bank balance:

```
<img src="https://your-bank.example/balance.json">
<!-- Note: the attacker refused to add an `alt` attribute, for extra evil points.
```

Without Site Isolation, the contents of the JSON file would make it to the memory of the renderer process, at which point the renderer notices that it's not a valid image format and doesn't render an image. But, the attacker could then exploit a vulnerability like Spectre to potentially read that chunk of memory.

Instead of using **<img>**, the attacker could also use **<script>** to commit the sensitive data to memory:

```
<script src="https://your-bank.example/balance.json"></script>
```

Cross-Origin Read Blocking, or CORB, is a new security feature that prevents the contents of **balance.json** from ever entering the memory of the renderer process memory based on its MIME type.

Let's break down how CORB works. A website can request two types of resources from a server:

1. *data resources* such as HTML, XML, or JSON documents
2. *media resources* such as images, JavaScript, CSS, or fonts

A website is able to receive *data resources* from its own origin or from other origins with permissive CORS headers such as **Access-Control-Allow-Origin: ***. On the other hand, *media resources* can be included from any origin, even without permissive CORS headers.

CORB prevents the renderer process from receiving a cross-origin data resource (i.e. HTML, XML, or JSON) if:

- the resource has an **X-Content-Type-Options: nosniff** header
- CORS doesn't explicitly allow access to the resource

If the cross-origin data resource doesn't have the **X-Content-Type-Options: nosniff** header set, CORB attempts to sniff the response body to determine whether it's HTML, XML, or JSON. This is necessary because some web servers are misconfigured and serve images as **text/html**, for example.

Data resources that are blocked by the CORB policy are presented to the process as empty, although the request does still happen in the background. As a result, a malicious web page has a hard time pulling cross-site data into its process to steal.

For optimal security and to benefit from CORB, we recommend the following:

- Mark responses with the correct `Content-Type` header. (For example, HTML resources should be served as `text/html`, JSON resources with a JSON MIME type and XML resources with an XML MIME type).
- Opt out of sniffing by using the `X-Content-Type-Options: nosniff` header. Without this header, Chrome does do a quick content analysis to try to confirm that the type is correct, but since this errs on the side of allowing responses through to avoid blocking things like JavaScript files, you're better off affirmatively doing the right thing yourself.

For more details, refer to the CORB for web developers article or our in-depth CORB explainer.

Why should web developers care about Site Isolation?

For the most part, Site Isolation is a behind-the-scenes browser feature that is not directly exposed to web developers. There is no new web-exposed API to learn, for example. In general, web pages shouldn't be able to tell the difference when running with or without Site Isolation.

However, there are some exceptions to this rule. Enabling Site Isolation comes with a few subtle side-effects that might affect your website. We maintain a list of known Site Isolation issues, and we elaborate on the most important ones below.

Full-page layout is no longer synchronous

With Site Isolation, full-page layout is no longer guaranteed to be synchronous, since the frames of a page may now be spread across multiple processes. This might affect pages if they assume that a layout change immediately propagates to all frames on the page.

As an example, let's consider a website named `fluffykittens.example` that communicates with a social widget hosted on `social-widget.example`:

```
<!-- https://fluffykittens.example/ -->
<iframe src="https://social-widget.example/" width="123"></iframe>
<script>
  const iframe = document.querySelector('iframe');
  iframe.width = 456;
```

```
  iframe.contentWindow.postMessage(
    // The message to send:
    'Meow!',
    // The target origin:
    'https://social-widget.example'
  );
</script>
```

At first, the social widget's **`<iframe>`**'s width is **123** pixels. But then, the FluffyKittens page changes the width to **456** pixels (triggering layout) and sends a message to the social widget, which has the following code:

```
<!-- https://social-widget.example/ -->
<script>
  self.onmessage = () => {
    console.log(document.documentElement.clientWidth);
  };
</script>
```

Whenever the social widget receives a message through the **`postMessage`** API, it logs the width of its root **`<html>`** element.

Which width value gets logged? Before Chrome enabled Site Isolation, the answer was **456**. Accessing **`document.documentElement.clientWidth`** forces layout, which used to be synchronous before Chrome enabled Site Isolation. However, with Site Isolation enabled, the cross-origin social widget re-layout now happens asynchronously in a separate process. As such, the answer can now also be **123**, i.e. the old **`width`** value.

If a page changes the size of a cross-origin **`<iframe>`** and then sends a **`postMessage`** to it, with Site Isolation the receiving frame may not yet know its new size when receiving the message. More generally, this might break pages if they assume that a layout change immediately propagates to all frames on the page.

In this particular example, a more robust solution would set the **`width`** in the parent frame, and detect that change in the **`<iframe>`** by listening for a **`resize`** event.

Key Point: In general, avoid making implicit assumptions about browser layout behavior. Full-page layout involving cross-origin **`<iframe>`**s was never explicitly specified to be synchronous, so it's best to not write code that relies on this.

Unload handlers might time out more often

When a frame navigates or closes, the old document as well as any subframe documents embedded in it all run their **`unload`** handler. If the new navigation happens in the same renderer process (e.g. for a same-origin navigation), the **`unload`** handlers of the old document and its subframes can run for an arbitrarily long time before allowing the new navigation to commit.

```
addEventListener('unload', () => {
  doSomethingThatMightTakeALongTime();
});
```

In this situation, the **`unload`** handlers in all frames are very reliable.

However, even without Site Isolation some main frame navigations are cross-process, which impacts unload handler behavior. For example, if you navigate from **`old.example`** to **`new.example`** by typing the URL in the address bar, the **`new.example`** navigation happens in a new process. The unload handlers for **`old.example`** and its subframes run in the **`old.example`** process in the background, after the **`new.example`** page is shown, and **the old unload handlers are terminated if they don't finish within a certain timeout**. Because the unload handlers may not finish before the timeout, the unload behavior is less reliable.

Note: Currently, DevTools support for unload handlers is largely missing. For example, breakpoints inside of unload handlers don't work, any requests made during unload handlers don't show up in the Network pane, any **`console.log`** calls made during unload handlers may not show up, etc. Star Chromium issue #851882 to receive updates.

With Site Isolation, all cross-site navigations become cross-process, so that documents from different sites don't share a process with each other. As a result, the above situation applies in more cases, and unload handlers in **`<iframe>`**s often have the background and timeout behaviors described above.

Another difference resulting from Site Isolation is the new parallel ordering of unload handlers: without Site Isolation, unload handlers run in a strict top-down order across frames. But with Site Isolation, unload handlers run in parallel across different processes.

These are fundamental consequences of enabling Site Isolation. **The Chrome team is working on improving the reliability of unload handlers for common use cases, where feasible.** We're also aware of bugs where subframe unload handlers aren't yet able to utilize certain features and are working to resolve them.

An important case for unload handlers is to send end-of-session pings. This is commonly done as follows:

```
addEventListener('pagehide', () => {
  const image = new Image();
  img.src = '/end-of-session';
});
```

Note: We recommend to use the `pagehide` event over `beforeunload` or `unload`, for reasons unrelated to Site Isolation.

A better approach that is more robust in light of this change is to use `navigator.sendBeacon` instead:

```
addEventListener('pagehide', () => {
  navigator.sendBeacon('/end-of-session');
});
```

If you need more control over the request, you can use the Fetch API's `keepalive` option:

```
addEventListener('pagehide', () => {
  fetch('/end-of-session', { keepalive: true });
});
```

Conclusion

Site Isolation makes it harder for untrusted websites to access or steal information from your accounts on other websites by isolating each site into its own process. As part of that, CORB tries to keep sensitive data resources out of the renderer process. Our recommendations above ensure you get the most out of these new security features.

Thanks to Alex Moshchuk, Charlie Reis, Jason Miller, Nasko Oskov, Philip Walton, Shubhie Panicker, and Thomas Steiner for reading a draft version of this article and giving their feedback.

Last updated July 12, 2018.