How do we Stop Spilling the Beans Across Origins?

A primer on web attacks via cross-origin information leaks and speculative execution

aaj@google.com, mkwst@google.com

Intro

Browsers do their best to enforce a hard security boundary on an origin-by-origin basis. To vastly oversimplify, applications hosted at distinct origins must not be able to read each other's data or take action on each other's behalf in the absence of explicit cooperation. Generally speaking, browsers have done a reasonably good job at this; bugs crop up from time to time, but they're well-understood to be bugs by browser vendors and developers, and they're addressed promptly.

The web platform, however, is designed to encourage both cross-origin communication and inclusion. These design decisions weaken the borders that browsers place around origins, creating opportunities for side-channel attacks (pixel perfect, resource timing, etc.) and server-side confusion about the provenance of requests (CSRF, cross-site search). Spectre and related attacks based on speculative execution make the problem worse by allowing attackers to read more memory than they're supposed to, which may contain sensitive cross-origin responses fetched by documents in the same process. Spectre is a powerful attack technique, but it should be seen as an iterative improvement over the platform's existing side-channels.

This document reviews the known classes of cross-origin information leakage, and uses this categorization to evaluate some of the mitigations that have recently been proposed: Cross-Origin Read Blocking (CORB), Cross-Origin-Resource-Policy (CORP; formerly From-Origin), Sec-Metadata, SameSite cookies and Cross-Origin-Window-Policy (COWP). We attempt to survey their applicability to each class of attack, and to evaluate developers' ability to deploy them properly in real-world applications. Ideally, we'll be able to settle on mitigation techniques which are both widely deployable, and broadly scoped.

Attacks

A significant contributor to the threat model of web applications is their large attack surface -- a malicious cross-origin attacker can force the browser of a logged-in user to make requests to any endpoint of an application to which she is authenticated. Applications generally cannot distinguish such requests from legitimate traffic initiated by the application itself, and therefore cannot reject them. Historically, this has led to the following classes of vulnerabilities:

1. **Cross-site script inclusion (XSSI)**: Any script-like response to a GET request can be directly included as a <script> by a cross-origin attacker who knows the resource's URL. If that response includes authenticated information, the attacker can often extract it, usually either by observing environment changes caused by executing the script, or via reflection.
    - **Current defenses**: A common XSSI protection relies on setting a parser-breaking prefix (`)]}'\n`) on script responses, fetching them with CORS, and evaluating their contents after stripping the prefix. Other alternatives include using POST requests, relying on unpredictable URLs, or setting a non-script MIME type with an accompanying `X-Content-Type-Options: nosniff` header.

2. **Cross-site request forgery (CSRF)**: One of the top client-side vulnerabilities on the web, CSRF stems from the fact that any application endpoint which responds to GET or POST requests and modifies server-side state may be directly requested by the attacker. As the browser automatically attaches cookies to cross-origin requests, the server cannot reliably tell them apart from legitimate requests sent by the application, resulting in the attacker's ability to execute actions on behalf of a logged-in victim.
    - **Current defenses**: CSRF is commonly prevented by requiring form submissions to carry a secret token verified by the server. However, developers need to remember to individually check for the presence of the correct token on all state-changing requests; omitting this check is a frequent source of vulnerabilities.

3. **Detecting the result of loading cross-origin resources**: Browsers expose information about the success or failure of a resource load (e.g. for images or scripts), even for cross-origin fetches. In many applications certain resources are only available to a subset of users, allowing the presence of a load or error event to be used to determine the user's logged-in status or, in applications with fine grained ACLs, deanonymize the user.
    - **Current defenses**: No general, reliable solution exists for these attacks.

4. **Timing attacks based on response size or server processing time:** The ability to send cross-origin GET and POST requests and accurately measure the response time lets attackers infer information about the response, even if they cannot view it directly. This enables damaging attacks such as cross-site search, based on exfiltrating secrets from applications with search functionality, and often reveals other application-specific traits.
    - **Current defenses**: There is no robust protection against this class of attacks. Applications may apply CSRF-like protections to sensitive endpoints, or require the presence of a special request header for APIs requested by same-origin endpoints; however, this may not be feasible in applications which allow users to bookmark or share URLs.

5. **Pixel-perfect timing attacks to extract the contents of renderable resources**: An attacker who can display a cross-origin resource (a document or image) in a window they control can learn the color values of its individual pixels. Attacks rely on setting up CSS rules and SVG filters to introduce substantial timing differences during rendering based on the color of a chosen pixel; detecting the color is then achieved by inspecting the embedding window's rendering performance using client-side APIs such as requestAnimationFrame.
    - **Current defenses**: X-Frame-Options prevents resources from being embedded in an iframe, allowing developers to protect document formats (HTML, plaintext, natively rendered PDFs) from this class of attacks. No defense currently exists for images.

6. **Polyglot-based data exfiltration**: Due to lax parsing rules for some resource types, such as stylesheets or plugin formats, attackers may exfiltrate data from server responses which include (properly escaped) user-controlled contents, or in some cases achieve script execution in the context of the hosting origin. Examples of past attacks include CSS-based data stealing, Comma Chameleon and Rosetta Flash.
    - **Current defenses**: Some attacks have already been mitigated by browsers and plugins by performing stricter MIME type checking, e.g. when loading cross-origin

CSS. Other mitigations rely on developers removing plugin-dependent patterns which allow for content sniffing (e.g. HTML-escaping data in non-HTML responses, or adding static, application-controlled prefixes to certain response types).

7. **Attacks based on framing**: This large class of issues includes common problems such as clickjacking, which allows attackers to force the user to interact with cross-origin frames, and several more esoteric threats. For example, an attacker who controls an iframe loaded on a high-value page may be able to exfiltrate text by adjusting scrollbar width to induce DOM reflow and detecting how this affects the position of their inner iframe.
   - **Current defenses**: X-Frame-Options or CSP frame-ancestors.

8. **Spectre**: Speculative execution features in modern CPUs may allow attackers to read the contents of process memory by performing timing attacks from JavaScript. Even if a browser implements process isolation, an attacker can force the loading of cross-origin responses into a process executing the attacker's scripts (e.g. by including them as an <img> or <script>), and then use speculative side-channel attacks to extract their contents.
   - **Current defenses**: Partial mitigations available to developers are listed here, but they do not cover all scenarios susceptible to attacks (details below).

The attacks outlined above rely on the ability to force the loading of cross-origin resources in a context which allows the attacker to extract some information about them, in spite of the usual same-origin policy restrictions. One separate, but conceptually related class of web information leaks is based on **direct DOM access**, where the long-standing ability to directly access certain properties of the Window object of cross-origin documents (e.g. enumerate window.frames) may allow attackers to infer sensitive information about application state. As with several other issues, there are currently no reliable defenses against this class of attacks.

Importantly, these categories of vulnerabilities (in particular, CSRF and XSSI) account for a sizeable fraction of security issues discovered in modern web applications. Conversely, framing-based attacks such clickjacking have largely been successfully mitigated by more narrowly-scoped protections via X-Frame-Options and frame-ancestors in CSP -- a compelling example of the security value of allowing applications to restrict certain types of unwanted cross-origin interactions.

It's worth noting that most browsers' third-party cookie blocking mechanisms may be a robust protection against leaking sensitive data from signed-in users, but only insofar as they actually prevent credentials from being delivered to an interesting site. Since interesting sites are often those with which the user regularly interacts, they're unfortunately likely to be carved out from protections either manually or automatically.

Having discussed the threats, let's now move on to a quick review of proposed defenses.

Protections

A number of approaches have been proposed to mitigate the risks posed by one or more of the threats described above, by preventing sensitive resources from loading into a context to which an attacker has access. Here, we'll walk through some of the more interesting mechanisms:

Naming cheat sheet
Several proposals have been renamed once or more; original names are included for reference below:
Cross-Origin Read Blocking (**CORB**) = Cross-Site Document Blocking (XSDB)
Cross-Origin-Resource-Policy (**CORP**) = From-Origin
Cross-Origin-Window-Policy (**COWP**) = Cross-Origin-Isolate = Cross-Origin-Options

Cross-Origin Read Blocking (CORB) / formerly XSDB: Explainer

Authors: Lukasz Anforowicz (Google), Charlie Reis (Google)

CORB prevents cross-origin resource loads for several types of responses (primarily, HTML and JSON, which cannot be legitimately loaded as resources) to keep them out of untrusted execution contexts. In browsers with process-based isolation it can prevent passing data from protected responses to untrusted renderer processes running attacker-controlled scripts, mitigating speculative side-channel attacks on CORB-eligible resources. It is currently the only Spectre protection which is likely to be enabled by default in user agents.

Pros: Enabled by default, without requiring application changes -- "free" Spectre mitigation for non-embeddable MIME types which commonly include authenticated data.

Cons: For compatibility reasons, doesn't protect all resources (e.g. anything other than HTML, XML or JSON), leaving room for attacks on images, JavaScript responses, file downloads and other MIME types. Focuses on Spectre, without mitigating other cross-origin attack types, e.g. timings or CSRF.

	XSSI	CSRF	Load detection	Timing	Pixel perfect	Spectre	Direct DOM
CORB	✔[1]	✖	✖	✖[2]	✖	✔[3]	✖

Cross-Origin-Resource-Policy (CORP) / formerly From-Origin: Discussion, spec

Authors: Anne van Kesteren (Mozilla), John Wilander (Apple)

Cross-Origin-Resource-Policy is an HTTP response header served on resource requests, controlling which origins are allowed to embed a given resource. It's analogous to X-Frame-Options, but applies to all kinds of responses (scripts, stylesheets, images), preventing them from being exposed to a cross-origin page.

[1] CORB protects against XSSI for some responses, but it does not cover the text/javascript MIME type.
[2] CORB may partially mitigate timing attacks if the server supports RFC8297 and browsers reject responses immediately when they are determined to be CORB-eligible without receiving the full response.
[3] Spectre protections are limited to CORB-eligible resource types and rely on browser process isolation.

Pros: Simple to use; especially in self-contained applications with no cross-origin dependencies it may be easily adopted by setting the header on all responses. The presence of a response header provides an explicit signal to the browser that the origin may wish to opt into process isolation.

Cons: More difficult to adopt in applications with resources requested by cross-origin documents (requires enumeration of all trusted origins in the response header). Does not affect server-side processing of requests, which leaves them open to CSRF and most side-channel attacks.

	XSSI	CSRF	Load detection	Timing	Pixel perfect	Spectre	Direct DOM
CORP (+ X-F-O)	✔	✖	✔[4]	✖[5]	✔	✔[6]	✖

Sec-Metadata: Spec, discussion

Authors: Artur Janc (Google), Mike West (Google)

The proposed Sec-Metadata HTTP request header indicates the provenance of a resource request (same-origin, same-site or cross-site, potentially with more granularity) to allow the server to make decisions based on the sender of the request and/or its destination. This enables servers to quickly reject unexpected resource requests and allows for more flexible server-side authorization logic.

Pros: Protects against most cross-origin attacks by letting the server refuse to process requests sent by untrusted senders. Can be adopted in applications with complex cross-origin dependencies; facilitates deployment by allowing developers to review origins requesting their resources before enforcing any restrictions.

Cons: More work to adopt by requiring server-side code changes. Doesn't provide user agents with an explicit signal that the application wants to opt into process isolation.

	XSSI	CSRF	Load detection	Timing	Pixel perfect	Spectre	Direct DOM
Sec-Metadata	✔	✔[7]	✔	✔[8]	✔	✔[9]	✖

SameSite cookies: Spec

Author: Mark Goodwin (Mozilla), Mike West (Google)

The most mature feature which allows the limiting of cross-origin interactions. SameSite cookies do not directly prevent attackers from loading cross-origin resources, but they cause such requests to be sent without credentials, rendering the responses of little value to the attacker.

Pros: Protects against most cross-origin attacks. Setting the SameSite attribute on cookies is a small, self-contained change.

[4] To protect from load status detection, CORP must be set on both success and error replies.
[5] The same considerations as for CORB apply here.
[6] CORP protections against Spectre relies on the browser's implementation of process isolation.
[7] To reliably prevent CSRF, Sec-Metadata must indicate if a request is a result of a top-level navigation.
[8] Local attackers may still conduct related attacks by observing traffic size on forced top-level navigations.
[9] Sec-Metadata protection against Spectre relies on the browser's implementation of process isolation.

Cons: SameSite cookies have proven difficult to adopt in existing applications, as they miss flexibility to allow some resources to be requested across origins; an origin using a SameSite cookie for authentication will not be able to provide authenticated APIs such as credentialed CORS endpoints, and will break common framing scenarios. Requires "strict" mode to robustly defend against CSRF, leading to top-level navigations being sent without cookies, which is incompatible with some applications.

	XSSI	CSRF	Load detection	Timing	Pixel perfect	Spectre	Direct DOM
SameSite cookies	✔	✔[10]	✔	✔	✔	✔[11]	✖

Cross-Origin-Window-Policy / formerly Cross-Origin-Options, Cross-Origin-Isolate: Proposal
Author: Ryosuke Niwa (Apple)

While other proposals attempt to prevent an attacker-controlled context from learning the contents of responses, they do not restrict the attacker's ability to directly interact with the DOM of cross-origin windows. To help implement Spectre protections in browsers without out-of-process iframes, Cross-Origin-Window-Policy allows documents to break direct DOM access, potentially preventing cross-origin navigations or traversal of the document's frames.

Pros: Preventing direct DOM access by cross-origin windows protects from attacks based on frame counting and navigation of the window to an attacker-controlled destination ("tabnabbing"). It can serve as a signal that the application wants to opt into the brower's process-based isolation.

Cons: May complement other mechanisms, but by itself does not offer substantial protection against information leaks.

	XSSI	CSRF	Load detection	Timing	Pixel perfect	Spectre	Direct DOM
COWP	✖	✖	✖	~[12]	✖	~[13]	✔

Historical note: Earlier isolation proposals
Similar concerns motivated several past proposals, including Entry Point Regulation and Isolate-Me -- ambitious attempts to lock down the attack surface of sensitive applications against cross-origin attacks. However, arguably due to the large scope and complexity of both proposals, they have not gained significant traction.

Allowing windows to protect themselves from direct cross-origin DOM access was proposed as part of disown-opener in CSP3 (discussion).

[10] Requires SameSite cookies in "Strict" mode.
[11] Relies on the browser's implementation of process isolation.
[12] Cross-Origin-Window-Policy does not protect against timing attacks, but can complement defenses against cross-site search and related issues by preventing the attacker from navigating cross-origin windows.
[13] Cross-Origin-Window-Policy helps achieve Spectre protection in browsers without out-of-process iframes.

Summary

While concerns about Spectre are a direct motivation for the mechanisms discussed above, we propose that it is critical to consider the broader problem of cross-origin information leaks and design defenses for this more general class of attacks. This is especially important for any opt-in protections whose value depends on adoption by application developers, for two reasons:

1. **Web developers don't understand Spectre**, and they shouldn't need to do so in order to protect their applications. However, they have long had to deal with other vulnerabilities discussed in this document (CSRF, XSSI, timing attacks). Providing mechanisms which can protect from a larger class of attacks, especially those known to developers, increases their security value and makes it more likely that they will be adopted in real applications.

2. **Web developers don't understand browser process models**, but are familiar with the concept of allowing application resources to be loaded only from a small set of origins from which the developer expects requests (for example via CORS, or when handling data sent via postMessage). Aligning security mechanisms with the standard web model of policing cross-origin relationships, instead of focusing on ad hoc mitigations tailored to Spectre, may make the protections more understandable and increase the likelihood of their adoption.

In practice, a thoughtful combination of the security features outlined above is likely to be sufficient to address Spectre as well as other cross-origin information leaks: smaller sites can augment CORB protections for Spectre by adopting Cross-Origin-Resource-Policy and X-Frame-Options, existing larger applications can prevent most cross-origin attacks by checking the Sec-Metadata request header and setting Cross-Origin-Window-Policy on all responses, and particularly sensitive new sites can leverage authorization schemes using an auxiliary "SameSite" cookie cryptographically tied to the application's authentication token.

In the end, we strongly believe that the success of browser efforts in this area depends on keeping a broader view of the attacks outlined in this document, and understanding how proposed mitigations fit together to allow developers to add meaningful protections in their applications.

Thanks to Krzysztof Kotowicz, Michal Zalewski, Devdatta Akhawe and Jasvir Nagra for useful feedback on earlier drafts of this document.