Capability Delegation API

A proposal for time-constrained delegation of capabilities

mustaq@chromium.org, danyao@chromium.org, flackr@chromium.org, girard@chromium.org, jyasskin@chromium.org

Explainer: github.com/WICG/capability-delegation
Chromium bug: 1130558
Chromestatus entry

Document status: initial draft
Started: 2020-July-23. Last major modification: 2021-June-29

Overview

This document explores design choices for Capability Delegation and aims to define a concrete JS API for the feature. This is based on a preliminary proposal in Capability Delegation: Design Alternatives.

What is Capability Delegation

Capability delegation means allowing a frame to relinquish its ability to call a restricted API and transfer the ability to another (sub)frame it trusts.

To elaborate, many APIs in the Web are usable from JS in restricted manners. For example:

- Most browsers allow popups (through `window.open()`) only if the user has either interacted with the page recently or allowed the browser to open popups from the page's origin.
- A sandboxed iframe cannot make itself full-screen (though `element.requestFullscreen()`) without a specific sandbox attribute or a user interaction within the frame.

If an app wants to delegate a task to a known and trusted third-party frame, and the task relies on a restricted JS API, the app would utilize a Capability Delegation mechanism to enable that particular API in the third-party frame. Our focus in this document is a *dynamic* delegation mechanism not currently available today.

Goal of this document

We are interested in a time-constrained delegation mechanism here that does not expose the delegated capability to the target frame in a time-unconstrained manner. The use-cases we want to cover here (see the corresponding section below) are not supported by existing delegation mechanisms in the Web (`<iframe allow*>` attributes and feature policies) which are static in nature; see a related discussion here.

Ideally we want to define an API that meets all of the following criteria, but it is conceivable that we would have to make a tradeoff:

1. The API would support capability delegation in a general form. In other words, it would be usable for capabilities other than the ones we have today.
2. Other browsers are willing to ship the API. Failing that, the API would be shippable from the web platform perspective, which means consensus from Blink API owners as well as approval from external TAG reviewers.
3. The API won't allow developers to do something sensitive without users' knowledge.

Background

Motivation

Payment Request API

The first specific use-case we want to support through our delegation mechanism is Payment Request API. Online stores that outsource payment related tasks to third-party services rely on this API.

To elaborate, there are many “middle-tier” merchants that are big enough to have their own shopping websites but not quite big enough to have their own payment infrastructure. They outsource the payment collection and processing infrastructure to Payment Service Providers (like Stripe) due to the considerable security and regulatory complexities around card payments. This creates a situation where the “pay” button is rendered inside the top (i.e. the merchant’s) frame where it can blend better with the rest of the merchant’s website, while the payment processing code is inside a cross-origin iframe from the PSP. A payment processing is initiated through a call to `PaymentRequest.show()` which requires transient user activation to prevent malicious attempts like unattended or repeated payment requests. Because the top (merchant) frame’s user interaction is not visible to the iframe, the PSP code needs some kind of a delegation in response to a click in the top frame to be able to use `PaymentRequest.show()`.

We got a feature request recently (Jun 6 2019) that calls for the ability to delegate popup capability to an iframe; details are here. Any capability delegation mechanism would solve their use-case as long as there is a way to specify the “payment” capability.

Other use cases

- A website may want a third-party chat app in an iframe to be able to vibrate the phone on message receipt, even when the user is not active in the iframe.
- A web service that does not care about user location except for a “branch locator” functionality provided by a third-party map app can delegate its own location access capability to the map iframe.
- A presentation/slide website that has a “control panel” to selectively make other spawned windows fullscreen. This came from this developer request we received following Chrome’s UAv2 launch (before this launch, user activation visibility was an undefined concept). With Capability Delegation API, the “control panel” would delegate fullscreen capability to the selected window to bring a non-focused window to fullscreen.
- An authentication provider may wish to show a popup to complete the authentication flow before returning an authentication token to the host site.

- Portals has users interacting with multiple sites in a single context. We suspect they may need to delegate capabilities in some form.
- We have seen one video-autoplay case where a cross-origin subframe with a video remains hidden behind a thumbnail image on the top frame, and a click on the thumbnail removes the image and expects to play the video in the subframe. The current model for user-activation visibility prevents such use-cases but we can use capability delegation API to enable this (subject to browser defined autoplay settings).

Related past attempts

The idea of transient (time-constrained) delegation was explored a few times before; we are aware of the following proposals:

- Gesture delegation explained
- Delegating user activation to child frames
- Combining gesture delegation with feature policy
- Activation delegation through transfer.

Challenges

The main challenge faced by past transient delegation proposals is enabling a legitimate use-case without causing problems with unrelated APIs. One common theme of those past proposals has been to tie the "transient" part of the capability delegation with transient user activation. In other words, "delegating a capability" was done through "delegating user activation". This approach turned out to be problematic from a few perspectives:

- "User activation" is a low-level state (representing user interaction), hence neither a developer-facing API nor a capability. TAG asked a logical question whether such a low-level concept should even be exposed to developers. Note that we already expose this to developers but as a read-only state only (and TAG raised the same concern for that too).

- The transfer API here would let developers modify the user activation state of a `Window` which feels too much of a control. The browser, as a *User Agent*, should never let a script mimic real user interaction in any way. Unfortunately, a user activation "transfer" would act as a "mimic instrument" to fool the target `Window` into believing that the user interacted with it. For existing activation-gated APIs that rely on the user activation state as a *trusted source* of truth about user interaction, this seems unacceptable.

  (The "trust" argument above is analogous to that for trusted events: only the browser can ever fire a trusted event, and we never allow a script to dispatch such an event—even from the handler of a trusted event. As a result, a code can rely on the "trusted" bit of an event without the fear of being fooled by a malicious JS.)

See the first Web Platform Design Principle.

The problem with delegation of user activation becomes obvious when we consider the fact that there are 25+ different APIs in Chromium. It does not make sense to delegate all these capabilities when an app needs to delegate just one of them. For example, to process payment requests through a subframe, the host app should not need to delegate popup or autoplay or geolocation capabilities to the subframe.

Transient Capability Delegation

To address the (valid) concerns raised against the past attempts, we are proposing a model that focuses on delegation of a specific capability only (instead of delegating user activation) in a time-constrained manner. We will call it *Transient Capability Delegation (TCD)*.

In short, the delegation of a capability *X* would consume the sender's user activation to create a time-limited availability-timestamp T_X on the receiving end. In more details:

A. The sender's ability to use TCD would be gated by transient user activation. More precisely, a TCD request will consume the user activation in the sender's `Window` to prevent repeated requests (making TCD a transient activation consuming API) but the receiving `Window` won't get any user activation at all.
    - Question: perhaps for certain capabilities, a TCD request could be non-consuming (i.e., transient activation-gated) instead?

B. Internally a successful delegation would create a time-constrained availability-timestamp T_X in the recipient `Window`. The lifespan and behavior of T_X would depend on the capability, and would be defined by the spec owners of capability *X*. availability-timestamp T_X won't be exposed to JS.

C. On the receiving end, T_X would be "tied" to the recipient `Window` object so it would be non-transferrable by design.

Proposed Javascript API

We are proposing a new option to `Window.postMessage()` that facilitates TCD through the existing messaging mechanism (based on the feedback received during our last attempt).

```
targetWindow.postMessage('a_message', {delegate: 'payment'});
```

Privacy and security considerations

There is no privacy concern here because user data is not involved in any way.

From a security perspective, the proposal may seem like a “packaging” of capability-permission plus user-activation into a token which can be passed around. However, there are quite a few important distinctions between the general idea of a “passable token” vs the “availability-timestamp” created here:

User activation is never delegated: We deliberately hold back user activation from being passed on to the receiving end because we already know this would be problematic, see the Challenges section above for some details. In other words, a call to delegate an API never activates the receiving `Window` so the receiver side cannot abuse user activation in any way (e.g. trying to open a popup after receiving an availability-timestamp for “fullscreen”).

Delegation consumes user activation: Moreover, a delegation call consumes user activation so a sender `Window` cannot abuse the state either (like trying to delegate to multiple subframes). Note that by design consumption of user activation affects the whole frame tree.

The new availability-timestamp is inaccessible: The only “visible aspect” of the availability-timestamp created at the receiving `Window` is that the capability becomes usable there (see next point about “usability”). The availability-timestamp itself is not accessible from JS, hence transitive (or chained) delegation won’t be possible. In other words, it would be impossible to pass on the availability-timestamp to another potential receiver.

The new availability-timestamp can be single-use: The owners of each capability (spec) are free to decide how the capability could be used at the receiving `Window` after an availability-timestamp has been created there. For example, a capability that is “very abusable” (like popup or fullscreen) can reset the availability-timestamp after a single call to prevent repeated calls, while a capability that is “mildly abusable” (like <input type=”file”> dialog) can allow multiple calls. One suggested rule of thumb is that the availability-timestamp for an activation-consuming capability should be single-use.

Open questions

Here are some open questions around TCD. We can choose to go with a non-generic (capability-specific) delegation API to bypass some of these questions. But a generic solution is preferred (and TAG seemed to have the same opinion).

Specifying capability X

- **Question:** Would it be a string, like “payment-request”? Or an enumeration label?
  - We decided to use feature-identifiers in the Permissions Policy spec (details).
  - ~~**Question:** How does it relate to feature policy and/or iframe attributes?~~
  - ~~**Question:** We need a standard naming convention for *X*, right? Could even have "capability.subcapability" hierarchy defined. Would it be useful?~~
  - **Question:** If the UA does not recognize *X*, it would have to reject the delegation request. How to convey this "rejection" to the recipient?
- **Question:** Even when we have a list of candidate APIs for *X*, is it obvious which *X* should be chosen? For example, would the payment request API require “popup” capability instead of a “payment-request” capability?
- **Question:** Should it allow passing multiple capabilities?

Handling capability guards

Each capability is controlled by a set of “guards”; these guards are lower-level capabilities and states that serve as the input parameters defining the usability/availability of the capability. Here is a partial list of capabilities and their guards: go/web-capability-guards.

- **Question:** The proposal here seeks to bypass one specific capability guard (i.e. user activation). What about other guards?

For example, a payment request is guarded by transient user activation, and a delegated payment request would allow *bypassing* this requirement on the recipient `Window`. But what if we have other guards, like a user option like “never allow payment requests from a specific origin”?

Note that *bypassing all guards* is a tricky (if not unacceptable) problem because different guards for a specific capability may have different “contexts”. A concrete example: the “autoplay video with sound” capability in Chromium is controlled by at least three guards:

a. the availability of user activation,
b. effective autoplay feature policy at the requester frame, and
c. requester site’s MEI (media engagement index).

Each of these guards is defined through a different “context”:

a. user activation ⇒ frame tree (browsing context hierarchy),
b. effective feature policy ⇒ feature policies of all ancestor frames, and
c. MEI ⇒ frame’s origin.