Policy Container

Authors: antoniosartori@chromium.org, pmeuleman@chromium.org
Last modified: 2020-12

Related documents:

- CSPs in the Policy Container
- COOP/COEP in the Policy Container
- The problem of the initiator RenderFrameHost
- PolicyContainer in the ExecutionContext
- Policy container for blob and filesystem
- Policy container for workers

Objective

Create a PolicyContainer class that can serve as a container for security policies (such as Content-Security-Policy, Referrer Policy, COOP, COEP, sandbox policy) that are applied to a document and have common, meaningful inheritance properties.

This will not introduce new features, but it will change the inheritance rule of those security policies to match the expected behaviour, hereby fixing some existing bugs and highlighting new ones. It will also simplify the code and facilitate the implementation of future policies.

Background

There is a fair amount of security policies that share the same lifecycle and inheritance logic. They are delivered via headers or meta tags, parsed into structured types, stored alongside a document, applied according to their custom logic and possibly inherited (for example, when navigating to about:blank or when creating a blob). However, both in the specification and in Chrome's implementation, those policies do not share logic. This causes duplication of code/specs (for example, every policy needs to define inheritance, even if all policies inherit in the same way) and some probably unwanted inconsistencies. In Chrome, this has been the source of several bugs (1117687, 1115628, 1115298, 1115045, 1109167, 971231, 957606). Moreover, it makes it painful and error prone to define a new policy, since one has to rewrite or copy-and-adapt the same logic over and over (both in the spec and in chrome).

The purpose of the PolicyContainer is to encapsulate that common logic in one single object. The PolicyContainer is attached to a document and has rules for creation and inheritance. Policies are stored in the PolicyContainer. If you want to define a new policy, you just need to

specify how to store it into the PolicyContainer and how your policy should be applied: everything else comes for free.

Plan

We start with a prototype implementation of the PolicyContainer in Chrome for Referrer-Policy (ideally, we would like to start with Content-Security-Policy, but see Problem 1. below). We will then gradually add other policies to the container. With the experience gained from the implementation, we will consider introducing the Policy Container concept in the specifications.

Policies

The list of policies that we have in mind for the purpose of this document is:

- Referrer policy
- COOP and COEP
- Content Security Policies

Those policies are delivered via network through http headers or meta tag. They are parsed in the network process or in Blink. At the moment, we directly store the parsed policies where they are needed (for example, the blink::ExecutionContext or the RenderFrameHost). Note that some of them, like Content Security Policies, are needed at different places, hence stored in different places.
The policy container will simplify things here: at parsing time, we will store the parsed policies in the policy container, and use the policy container to propagate the policies where they are needed. In this way, we will have a single source of truth for the policies parsed values.

Inheritance

The true goal of the policy container is to take care of inheritance for local-scheme urls. Indeed, when a document creates a local-scheme[1] new document, we must apply the same (or almost) policies to the created document. Otherwise, this would offer a trivial way for the original document to bypass the security policies (just create an about:blank iframe and do everything from there).
In detail, inheritance will work as follows:

- **About:blank** (initial empty document) and **about:srcdoc** documents should inherit the policy container from the creator of the document. For an iframe, the creator is always the parent. For a popup, it is the document which called `window.open`.
- **about:blank** (navigated to) and **data:** documents should inherit from the initiator of the navigation.
- **Javascript:** navigations should not change the policy container.
- **Blob:** and **filesystem:** urls should inherit from the creator of the blob/file, hence the policy container must be persisted in the blob/filesystem storage.

[1] See https://fetch.spec.whatwg.org/#url. Note we also include filesystem: as local scheme, since it is analogous to blob: but it is chrome specific and not included in the Fetch spec.

Note: by '**inherit**' we mean that the local-url document gets a (deep) copy of the creator/initiator's policy container. The copy can later change independently of the original.

History

History navigations to local-scheme urls should reapply the original policy container, which the local-scheme url document got when it was first created. Hence the policy container must be stored in the history for **about:** and **data:** urls. Note that this is not needed for network schemes, since a history navigation to a network scheme will redo the network request, reparse header and meta tags, and hence be able to reconstruct the document policies. Also for **blob:** and **filesystem:** urls this is not needed: in fact, their policy container is persisted in the blob/filesystem storage. A history navigation to a blob/filesystem will reload the data from storage, hence also the policy container.
The same must happen for session restore, hence we also need to persist the policy container on disk in that case.

Design

PolicyContainer will be a class in content. A PolicyContainer contains the policies that apply to a document. It is owned by a RenderFrameHost (RFH). There is a Blink's replica of the PolicyContainer, which is owned by the LocalFrame and replicates the policies that must be enforced by Blink. It also has a mojo remote connecting to the browser's PolicyContainer, allowing to update policies, which is needed for policies delivered via meta tags.

Security

- The policy container should restrict which policies are exposed to blink to the strict minimum required for the renderer to function correctly, the other policies are restricted to the browser.
- Blink can update some policies, for instance Content Security and Referrer policies can be updated with meta tags. In these cases, blink is able to update the policy container and notify the browser of the change. These changes must be limited to more restrictive values, where the HTML specification allows it. For example, the browser process will ignore any update from the renderer that would reduce the CSP constraints.
  Note that referrer policy can be overridden fully by meta tags, including to less secure values.

PolicyContainer lifecycle

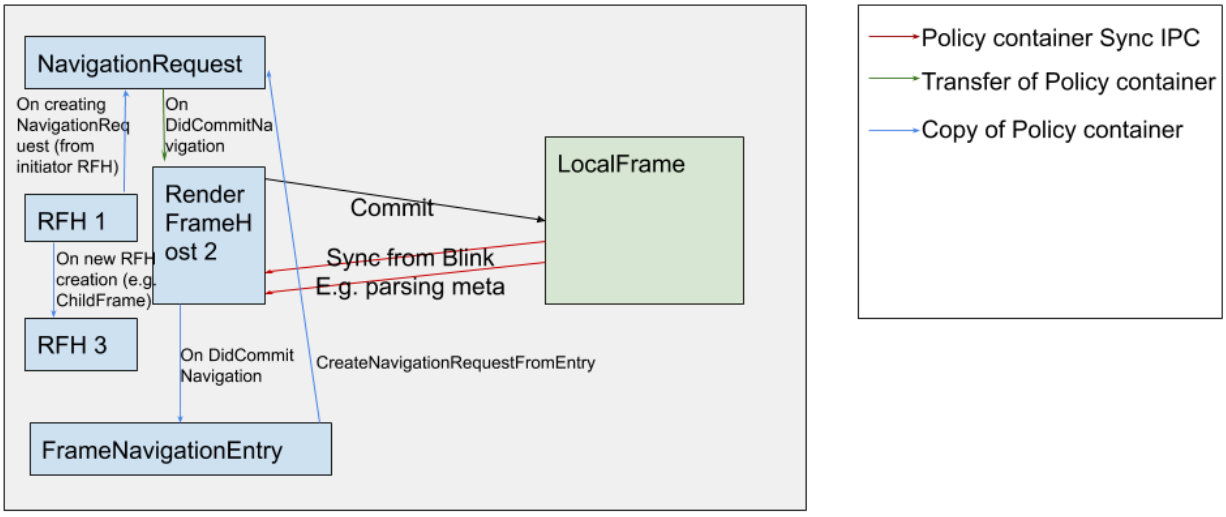
We want to have a content::PolicyContainer whose lifecycle in the browser is the following:

- Every RFH has a PolicyContainer.
- If a RFH creates a new RFH (e.g. OnCreateChildFrame or when opening a new window through creation of a new FrameTree) then its PolicyContainer is cloned.

- If a RFH is created without an initiator (e.g. first open tab) then it gets an empty (default) PolicyContainer.
- When creating a new NavigationRequest, if the navigation has an initiator, the PolicyContainer of the initiator RFH is cloned and attached to the NavigationRequest.
    - In some circumstances the initiator RFH might not be available when a request is initiated (see comment). To work around that, the policy container will outlive its owning RFH while the blink objects are still alive.
- At ReadyToCommit time, the NavigationRequest recomputes the PolicyContainer:
    - If the scheme is local, then the PolicyContainer remains the initiator's PolicyContainer
    - If the scheme is non-local, the PolicyContainer is initialized to an empty one, and policies are filled in from the parsed headers.
- When the RFH processes DidCommitNavigation, it moves the NavigationRequest PolicyContainer into its own PolicyContainer.

Regarding the browser/blink interaction, we add mojo methods to the LocalFrame and LocalFrameClient such that

- When the RFH creates the LocalFrame, it sends the PolicyContainer so that the LocalFrame can store its own PolicyContainer in blink.
- When the RFH commits a navigation to blink, it sends NavigationRequest's PolicyContainer along with the commit parameters. Blink stores in the LocalFrame.
- When blink updates policies (by parsing meta tags) it calls a mojo function to tell the RFH about the update.



Tasks

This is a tentative list of tasks for implementing the Policy Container.

1. Define PolicyContainer types

Define new mojom struct and C++ class

2. Attach the PolicyContainer type to the RenderFrameHost

3. Attach the PolicyContainerMojom to the LocalFrame

4. Add sync method from Blink to Browser

We would add a method to the LocalFrame and RenderFrameHost interface to sync the container when new policies are parsed by Blink.

5. Store ReferrerPolicy in the Container

6. Initialize PolicyContainer on RFH creation

When a RFH is being created, we clone the PolicyContainer of the creator RFH and attach it to the new RFH.

7. Attach initiator PolicyContainer to NavigationRequest

A NavigationRequest holds an initiator PolicyContainer (attached on creation time by cloning the PolicyContainer of the initiator RFH, if it exists, otherwise just an empty PolicyContainer).

8. Compute PolicyContainer in NavigationRequest (inheritance)

At ReadyToCommit time, the NavigationRequest can recompute the PolicyContainer for the target document (either inheriting the initiator's PolicyContainer, or creating an empty PolicyContainer and storing the policies from the parsed headers into it).

9. Commit PolicyContainer

At commit time, the RFH mojo-messages the LocalFrame with the new PolicyContainer (as part of the Commit message).

10. Blink → Browser updates of PolicyContainer

When blink parses the meta tags, it could update policies to the PolicyContainer. We would implement a mojo message LocalFrame → RFH to propagate this to the browser.

Note: We probably want to have some security checks on what blink can update. For example, it might make sense that blink can only add additional CSPs, but not remove them. For Referrer-Policy, blink will be able to completely overwrite it. However, it could be a security issue if we allowed blink to later override COOP and COEP, so we might want to protect them.

11. Store and retrieve PolicyContainer with history

For history navigations to local urls, we should store the PolicyContainer in the history and copy it within the NavigationRequest.

12. Add/adapt WP tests for Referrer-Policy inheritance

Although there are some WP tests already that check that Referrer-Policy is inherited from the parent in case of local-scheme iframes, we should adapt and expand them to cover all the cases we implemented above.

13. Add COEP to the PolicyContainer

COEP is a browser and network only policy, it should inherit similarly to other entries of the PolicyContainer. One particularity is that the policy is checked when embedding resources, which might go directly to the network service through the URLLoader[Factory]. We will continue with this behavior, copying COEP from the PolicyContainer to the URLLoaderFactories whenever needed.

14. Add COOP to the PolicyContainer

COOP is a browser only policy, it requires COEP to be fully computed. The reporting API is exposed to Blink, as an access that would be prevented by COOP needs to be reported. COOP systematically compares the policies and origin of the previous document and the new document during the navigation.

15. Add Content Security Policies to the PolicyContainer

Because of Problem 1. below, we would just store the raw policies (together with metadata like “is_report_only” and “is_from_meta_tag”) in the PolicyContainer. We need to add the CSPs to the PolicyContainer when parsing the headers (probably in the browser at navigation time, just before commit?) and when parsing the meta tags (in blink).

16. Switch to PolicyContainer inheritance for CSP

In the document_loader, we remove the initator_csp and instead rely on the PolicyContainer for inheritance. Since the PolicyContainer will contain the right CSP in all cases (the navigation made sure of that, by either copying the headers or inheriting the previous policy), we can just always parse that in blink. However, we should probably omit parsing errors which we displayed already in case of inheritance.

17. (optional) Switch to PolicyContainer for browser-side CSP checks other than frame-ancestors

Since the RFH now has an up-to-date version of the CSPs at all times, we could use that for the browser CSP checks (mainly frame-src and navigate-to) instead of passing the CSPs along the navigation as we do now. This would allow to remove quite some code, at the cost of reparsing the raw CSPs in the browser to the network::mojom CSP types for every check.

Future work

1. Once we have a PolicyContainer, we can implement correct inheritance for **blob** and **filesystem URLs** by storing the PolicyContainer of the blob creator alongside the blob in the blob store.
2. If we switch to using mojo types for blink CSP checks (requires some work!), we can store them in the PolicyContainer instead of the raw CSP strings. We could then avoid double-parsing CSPs (as we are doing at the moment) and clean up some code. Might be worth trying out.

Problems

1. CSPs at the moment are stored and handled in two completely different types in blink and in the browser. A clean solution would require refactoring blink CSP code to use mojo types. In the first step, we can just store the raw CSPs as strings and parse them every time we need them. Note that this might be a common issue of other policies that need to be enforced both in the browser and in blink.
   These are discussed in CSP in the Policy Container.
2. The initiator RFH might be gone at the time when we create the NavigationRequest. One possible solution would be to associate a handle of the browser side Policy Container with the frame in the renderer and use that as a parameter. A better solution, in my (antoniosartori@) opinion, is to let the policy container outlive its RenderFrameHost. We can keep it around and delete it only once its mojo remote is disconnected.

Decisions

1. Should PolicyContainer be just a mojo type or a full-blown interface?

If we distinguish between blink and non-blink policies (see point 2. below), then it could make sense to have a full-blown interface.
would it make most sense to have something like:
content/browser/[...]/policy_container.h with:

```
class PolicyContainer{
//Browser only policies
COOP coop;
// browser-blink policies
mojo::PolicyContainer _shared_policies;

// applies restrictions to what can be updated by the renderer,
e.g. in CSP. Called by RFH.
UpdateFromRenderer(const mojo::PC & other);
};
```

[...]/policy_container.mojom:

```
struct PolicyContainer{
  network.mojom.ReferrerPolicy referrer_policy;
  // others here such as CSP stuff
};
```

Update [...]/frame.mojom

```
//add IPC to frameHost to sync the policies
//(after commit from renderer).
```

Or have policy_container.mojom be an interface and define the IPC[s]?
Decision: Policy Container will include a class in content as well as an interface defining the sync of policies from blink to browser

2. Should we have a PolicyContainer in content/browser and expose only a subset to blink (through mojo)?

Camille suggested the split of browser-only policies with the ones known to Blink. It is unclear whether this has significant advantages to justify the added complexity:

- Avoid leak of the browser only policies. Is this true though since Blink has access to the request? Inherited policies would leak when they otherwise would not.
- Simplification of the syncing process check: The browser only policies can simply not be updated by Blink.
- Reduction of the attack surface, as less policies are susceptible to be manipulated by Blink.

Decision: Policy container will have both a class in content/browser and a mojo struct in blink. The mojom struct in blink will be a subset of the content/browser.

3. Should the PolicyContainer be exposed to the network service?

This popped up in the discussions previously, some policies are used by the network service, but we believe that this is not enough to warrant a PolicyContainer (or a sub part if we go with 2) exposed to the network service. In cases where the network needs a policy, we can transfer that policy directly.
CSPs are mostly not needed by the network service. COEP is directly passed to the URLLoaderFactory and the network.
Decision: Do not expose PolicyContainer to the network service.

4. Should we add a new pre-commit IPC to transfer the PolicyContainer from the browser to the renderer

We discussed adding an additional IPC just before the commit to transfer the PolicyContainer from the browser to the renderer process. This stemmed from the several adapter layers (web) we'd have to go through within the renderer to transform the data in the correct types and the increase of complexity for removing those layers due to the addition of new objects.
Clamy@ pointed out that the additional risk and complexity of this additional IPC would far outweigh the cost of going through the web conversions.
Decision: Use the existing commit interface to transfer the Policy container.

Open points

5. Should PolicyContainer be replacing referrer policy (and others) as a parameter in requests?

A non-negligible amount of uses of policies within the PolicyContainer will be as initiator for navigations. Piping the PolicyContainer in the different calls initiating navigations will facilitate future work.
This will be a non-trivial amount of work with potential risks.

6. Which object should own the PolicyContainer on blink's side?

Currently, we went for the LocalFrame. It should be noted that the executionContext is owning the referrerPolicy and CSP (and it's owning class, SecurityContext).

7. Should PolicyContainer be expanded to contain other inherited properties?

For instance the referrer of the initiator, the secure context bit, the origin, ip address space are all document properties that are inherited by local schemes in a similar way to policies, should these be grouped in this container? These would similarly benefit from centralized infrastructure, as this would ensure consistent inheritance of these properties.

Adding them to PolicyContainer would likely require renaming PolicyContainer since these properties are not policies. This is not hard but it does require a non-zero amount of work.

The current decision is to leave this for later. First off, PolicyContainer should be prototyped and proved useful on a few policies. Only once that is established should we consider expanding its scope.

8. Impact of noopener on PolicyContainer

COEP and COOP are not inherited when there's noopener.
COOP can set noopener late when the opening frame is Xorigin with the main frame and COOP is same-origin.
We may need to take this into consideration.

Related documents

COOP/COEP and Policy container
CSP in the Policy Container
Policy container worklog
Arthur's introductory diagram