Chrome privacy sandbox enrollment design

shivanisha@chromium.org
Visibility: public
Status: Complete

Overview

This document describes Chrome's client-side design to support the enrollment for privacy sandbox, and includes the following aspects:

- Requirements from the server/enrollment UX backend
- Enrollment list delivery and processing at the browser
- API gating

Reference docs:
Public blog: https://developer.chrome.com/blog/announce-enrollment-privacy-sandbox/
https://github.com/privacysandbox/attestation/blob/main/how-to-enroll.md

Platforms

	Select applicable platforms in **Bold**	Reasoning, if applicable
Desktop	**Linux, Mac, Windows, Chrome OS, Lacros (go/lacros)**, Fuchsia	*All platforms that support privacy sandbox Ads APIs: Protected Audience API, Topics, ARA, SharedStorage, Private Aggregation*
Mobile	Android: **Chrome**, WebView (WebView is discussed in the follow up section)	*All platforms that support privacy sandbox Ads APIs*
	iOS	*Impacted Ads APIs are not available on iOS so this feature will not be supported on iOS*

Team

Chrome Eng: shivanisha@, xiaochenzh@, gtanzer@, lbrady@
PM: georgiaf@, mjv@

PRD

https://developer.chrome.com/blog/announce-enrollment-privacy-sandbox/

Launch entry bug

4260778

Code affected

- New component to be added for component updater,
- File format (protobuf) to be created.
- New functionality to read the component file in memory which will be accessed by the class below,
- New class defined in component/privacy_sandbox_attestations/ that provides interfaces to return whether a given site is enrolled or not and is attested for a specific PS API.
- New flag to be able to bypass given sites check for API gating.
- New metrics

Design

Requirements from the server/enrollment UX backend

This section summarizes what goes on at the server before an enrolled list is ready for Chrome's consumption. For simplicity, this summary considers the backend system as a single entity, while in reality it may have multiple parts. The backend system that processes the enrollment UX would need to accomplish the following prerequisites:

- Once an entity enrolls, the backend system does input validation and provides an attestation file to be kept at a well-known location on the entity's server.
- The backend system will then verify that the attestation file at the well-known address is correct. The well-known path is derived from the enrolled site as <enrolled-site>/.well-known/attestations
- Once that verification is complete, the backend system will create an enrolled sites list for Chrome's consumption, every X days if there has been a change. Note that even though the actual enrollment takes the DUNS number and the site, Chrome is only concerned with the site. DUNS verification is a prerequisite for a site to be added to the enrolled sites list but does not need to be part of the data that Chrome uses .
- Chrome's enrollment list will contain the following information for each enrolled site:
    - Site
    - For each API gated on enrollment: enabled or not
- Once a new enrolled site list is ready, Chrome's component updater, which will run every few hours, will pull the new list (details below). Checking every few hours for an update is done for all components and is not configurable per-component. Although for components which are likely infrequently changed, like the enrollment list, it is not much of an impact on system health since only changed components will be downloaded, and the downloads scale only with the delta changed.

Note that enrollment and attestation is used interchangeably in the doc, because from a Chrome browser's perspective, any site which is in the list is enrolled and attested for the APIs.

Enrollment list delivery and processing at the browser

The Component Updater is a piece of Chrome responsible for updating other pieces of Chrome. It runs in the browser process and communicates to Omaha servers (go/omaha) to find the latest versions of components, download them as deltas, and register them with the rest of Chrome. Some examples that are components are Chrome's subresource filter and privacy sandbox's first-party sets.

The primary benefit of components is that they can be updated without an update to Chrome itself, which allows them to have faster (or desynchronized) release cadences, lower bandwidth consumption, and avoids bloat in the (already sizable) Chrome installer. The primary drawback is that they require Chrome to tolerate their absence/lack of freshness in a sane way (detailed below).

In the normal configuration, the component updater registers all components during browser start-up (RegisterComponentsForUpdate() will be updated for the enrollment component), and then begins checking for updates six minutes later, with substantial pauses between successive updates. For supporting the enrolled sites list:

- A new component for the list of enrolled sites would be added. A component delivered using component updater is either a data file or a dynamically-linked library. In this case, it will be a data file, named "**Privacy Sandbox Attestations**". All components are delivered as CRX files (signed ZIP archives).
- Server side, component updater supports various file systems where the component can be stored: x20 (subresource filter list uses x20), cns or placer. The proposal here is to use x20 for this component given the familiarity, ease of use and that other privacy sandbox components are x20 based as well (private state tokens and first party sets).
- "**Privacy Sandbox Attestations**" component will be the same for all supported Chrome platforms so it just needs a single location for all platforms, /some_path.../VERSION/file(s)
- At the server-side, there is one time work needed as detailed here to create a CRX signing key, adding the component in Automation.java, adding the key KeySource*.java, adding the team's prod (MDB) group to mdb/client-update-release2-users and finally creating and editing the new Omaha product at http://omaharelease/.
- The steps for the above work is here:
  Server-side Components Implementation of Privacy Sandbox Enrollment
- Aside from the one-time work, there would be code needed to enable automating updating the new version of the list any time a site's enrollment is added/deleted or updated. The new list would be placed in the new version directory for omaha to pick it up. This work is part of the backend system described in the section above.

Local testing flag to bypass privacy sandbox enrollment

Local testing of gated APIs by developers does not necessarily need the site to be enrolled as Chrome would provide a flag that can be enabled locally to bypass the enrollment check for a given origin. It will be provided as a dev tools flag subsequently since as compared to chrome://flags, it enables it being available in non-english translations and it's a more natural place for permanent testing flags.
In the interest of timelines though, the first iteration of this flag will be in chrome://flags in M116. The flag will be disabled by default and have the type as a list of origins.

Fallback if component is not present

[Update Dec 12, 2023]: Note that Chrome is applying a default-allow behavior temporarily for the startup time lag where the list is not yet parsed in memory i.e. allowing the APIs to be allowed when the attested sites list is not yet present. This is being done to help with API loss since we are seeing API loss in the wild more than originally expected.

What should the browser do if the "**Privacy Sandbox Attestations**" component is not present in the expected directory? This can happen in the following scenarios:

Startup behavior
At browser start-up, when the component needs to be downloaded/installed/read. This is a transient state and will impact the gated API calls for the duration of either of the following startup scenarios:

1. component updater fetching the list for the first time (6 minutes post startup) + parsing the list in-memory or
2. reading the existing version of the component + parsing the list in-memory. Like 1, this is also asynchronous and does not block startup. Component updater itself has some cost to being available at startup (measured as O(100ms) from the network service), and thus we cannot block startup.

The fallback behavior for the gated APIs will then be to either allow for this duration or disallow. Even though, allowing is better for utility since disallowing could lead to a negative revenue impact even if the ad-tech is enrolled, default-deny is better for privacy as it prevents sites without attestations from using the Privacy Sandbox APIs.

A new metric, **PrivacySandbox.EnrollmentStatusWhenAPIInvoked.<APIName>** with values kNotAvailable, kAllowed, kNotAllowed, would help measure the percentage of times the component was not available. Additionally, we can divide the histogram between the first time the enrollment list is ever fetched (scenario 1) and subsequent runs (scenario 2).

Default-deny

- Privacy is always guaranteed and does not bias towards sites which are often first page loads. Default allow could lead to such pages getting access to the gated APIs even when they are not enrolled and attested. We might want to add UKM of the above metric to see if this is indeed the case.
- If PrivacySandbox.EnrollmentStatusWhenAPIInvoked.kNotAvailable is a very low percentage of the total, this seems the right call. The expectation is that it will be a low percentage in subsequent runs when a given version of the component is present (scenario 2 above). In scenario 1, when the component is fetched the first time, it is expected to be relatively higher but that also means we can't default-allow for that much duration, violating the enrollment guarantee.
- Android does a default-deny and would be consistent across both platforms.

When to fetch the component/parse it in memory

Chrome will fetch the enrollment list at the earliest possible chance after startup.
In Chrome, component updater will always fetch the list at the earliest possible time and it will be parsed into an in-memory map as soon as possible. We considered whether it will help save memory cost by only reading and parsing it when the privacy sandbox notice has been viewed by the user, but decided against the optimization. The rationale being the notice is shown to the user right around browser startup even without any user activation and is a modal dialog so the user has to interact with the notice ("OK" or "Take me to settings"). Given the small time in which the user is without the notice, any ordering with the notice will involve code complexity without much optimization.

Other scenarios

Other edge scenarios that can lead to the component being absent:

- Due to user error, if the file is removed from the profile directory. In this case, the browser will continue to work with the in-memory list until it is restarted. At restart, component updater will fetch the enrollment list.
- If component updater is disabled via the switch kDisableComponentUpdate, the enrollment site map will be considered empty and no APIs will be invoked unless bypassPrivacySandboxAttestationsCheck is enabled . But this is not a state we expect real users to be in, and is more of a local testing scenario, so should be ok.

Non Chrome chromium embedders

Since this is a Chrome-only feature, for other Chromium embedders, the APIs will not be gated. This includes WebView in the initial version since no gated APIs are supported on WebView , with the caveat that the Attribution Reporting API delegates from WebView to Android and would be gated as part of Android's attestation based gating.

Does Chrome fetch the attestation file

The short answer is **No**. Read below for the rationale.

Component updater will fetch the new enrollment list in O(hours) once it is ready after server-side processing of the enrollment. As mentioned in the server-side requirements section above, there are a few steps that need to be done between the entity enrolling and it being added to the enrollment list. Once the attestation file at the well-known address is verified by the server, the entity will be considered successfully registered, and will be added to the enrollment list. The question is whether it is worthwhile for the browser to fetch the attestation file from the well-known address as well, to speed up the time a site will be considered enrolled.
The delay between the browser receiving it via component updater depends on the server side processing being scheduled to add it to the list after the attestation file is sent and later verified. Since that's feasible to be scheduled within O(a few hours to a day), the recommendation is for the browser to receive a site's status only via the enrolled list and not fetch the attestation file separately.

The alternative would be to fetch the attestation file from the site's well-known address and verify in the browser if a site isn't present in the enrolled list. The downsides for this approach are:

- An additional network round trip in the critical API invocation path.
- Additional browser implementation to verify the attestation file, which is already being done at the server end.
- Server could maliciously alter attestations per user to fingerprint i.e. gaining a persistent 'n' bits of information (based on 'n' Privacy Sandbox APIs enrollment status) that is consistent across sites.

In memory representation of the list

The component file on the disk will be read in memory and stored as a map from sites to that site's API attestation status for each API (code in the next section).

- Impact on time: We intend to track the startup metrics (see Speed section) and have some local timing tests to validate the impact. We are expecting an insignificant impact on the startup time given that reading from the disk and parsing the list in memory is asynch and does not block startup. In terms of parsing time, the expectation is that it will be minimal as well, assuming the size of the enrollment list is going to be O(1000) of records. (A similar local study done by the First Party sets team showed a list (with ~700 sets) was read from disk in half a ms, and was parsed in less than 0.1 ms.)
- Impact on the memory of the process: Similarly, the impact on memory is expected to be minimal as mentioned below in the core principle section.

API gating based on the list in memory

The APIs that will be gated behind enrollment and attestation are:

- Protected Audience API
- Shared Storage
- Topics
- Attribution Reporting API
- Private Aggregation API

The following APIs will not be gated:

- Fenced frames created due to invoking Protected Audience API and selectURL. The rationale for this is described below in the Protected Audience API section.

The following changes will accomplish API gating:

- New Component to be registered as here (code).
- A new class PrivacySandboxAttestations will be added in components/privacy_sandbox/privacy_sandbox_attestations/ for PS APIs to access to determine if a given site is enrolled or not. [CL]
- Existing class PrivacySandboxSettingsImpl is already being used at API access points to check if a given API is allowed based on preferences and cookie blocking settings. The following PrivacySandboxSettingsImpl functions will be enhanced to further invoke the new PrivacySandboxAttestation::IsSiteAttested() detailed below, before returning the answer.
    - IsTopicsAllowedForContext [CL]
        - Attest `url`, which is from the calling site, with `kTopics`
    - IsAttributionReportingAllowed [CL]
        - Attest `reporting_origin` with `kAttributionReporting`
    - MaySendAttributionReport
        - Attest `reporting_origin` with `kAttributionReporting`
    - IsFledgeJoiningAllowed
        - No attestation. Whenever IsFledgeJoiningAllowed() is called, IsFledgeAllowed() has already been called, which handles attestation.
    - IsFledgeAllowed
        - Attest `auction_party`, which is a variety of entities in the auction, with `kProtectedAudience` (See section below on Protected Audience)
    - IsSharedStorageAllowed
        - Attest `accessing_origin`, which is the caller's origin, with `kSharedStorageAddModule`
    - IsSharedStorageSelectURLAllowed
        - No attestation. Whenever IsSharedStorageSelectURLAllowed() is called, IsSharedStorageAllowed() has already been called, which handles attestation.

        - IsPrivateAggregationAllowed
            - Attest `reporting_origin`, which is the worklet's origin, with `kPrivateAggregation`
        - Event-level reporting destinations
            - This is to require attestation of the recipient of the report
- PrivacySandboxSettingsImpl's histogram for the reason why a given API was denied will be extended with the kNotAttested value for this.

```
namespace privacy_sandbox {

enum class PrivacySandboxAttestationsGatedAPI {
  kTopics,
  kProtectedAudience,
  kPrivateAggregation,
  kAttributionReporting,
  kSharedStorage,

  // Update this value whenever a new API is added.
  kMaxValue = kSharedStorage,
};


using PrivacySandboxAttestationsMap = base::flat_map<
    net::SchemefulSite,
    base::EnumSet<PrivacySandboxAttestationsGatedAPI,
                  PrivacySandboxAttestationsGatedAPI::kTopics,
                  PrivacySandboxAttestationsGatedAPI::kMaxValue>>;

class PrivacySandboxAttestations {
 public:
  explicit PrivacySandboxAttestations(
      const PrivacySandboxAttestationsMap& attestations_map);
  ...

  // Returns whether `site` is enrolled and attested for
`invoking_api`.
  // (If the `kEnforcePrivacySandboxAttestations` flag is enabled,
returns
  // true unconditionally.)
  bool IsSiteAttested(net::SchemefulSite site,
                      PrivacySandboxAttestationsGatedAPI
invoking_api) const;

 private:
  PrivacySandboxAttestationsMap attestations_map_;
};

} // namespace privacy_sandbox
```

Since the enrollment site list check is in the critical path of the APIs e.g. ad-tech invoking runAdAuction, one of the requirements is for IsSiteAttested() to be synchronous and fast, which implies looking up the enrolled sites list as an in-memory map.

Does renderer process require the enrollment status
The existing API gating on user preferences (mentioned above) already needs to be done in the browser process, there is no real advantage for the renderer process to also check the enrollment status. Even if we checked the status in the renderer process, it would require an assertion check at the browser process because the renderer process could be compromised.
Given the above, for simplicity, the decision is to only check the enrollment list in the browser process.

Memory list update - when does it get refreshed?
That then brings up the question of when does the in-memory map get refreshed after a change in the enrollment site via component updater. As soon as the component updater sends back the event that a new version of the file is received, Chrome will update the in-memory map.

Note: We considered whether flatBuffers will help as it avoids the parsing stage and can be accessed without the in-memory map, but decided against it for simplicity. Given the minimal impact expected from the enrollment list on parsing time and memory, ease of parsing and that it is only needed by the browser process, it doesn't seem worth the complexity of using flatBuffers (as opposed to subresource filter component in Chrome which decided to use flatbuffers for parsing and because it needed to be shared across browser and renderer processes as shared memory).

Protected Audience API

Protected Audience API would require multiple API invocations to be gated, as follows:

- joinAdInterestGroup invocation - Will be gated via changes to PrivacySandboxSettings::IsFledgeJoiningAllowed if the "owner" of the IG is attested ,
- SSP in the runAdAuction arguments as "seller" in the AuctionConfig - Will be gated via changes to PrivacySandboxSettings::IsFledgeAllowed. Note this should include checking the component seller as well from their respective AuctionConfig
- sendReportTo() / forDebuggingOnly.reportAdAuctionLoss() / forDebuggingOnly.reportAdAuctionWin() - Will be gated via checking IsPrivacySandboxReportingDestinationAttested() on the report URL origin, which is allowed to be different from the seller or buyer origins.

    - All of these reports do not follow redirects (spec, code), so there is no question of impact on redirects.

- registerAdBeacon reporting beacon destination - Will additionally need to invoke a new method PrivacySandboxSettings::IsEventReportingDestinationAllowed() with the destination site in registerAdBeacon when reportEvent is invoked.
    - If the destination decides to redirect, it's that site's responsibility to share any data in the redirect URL as per the attestation guarantees. The redirected site is not checked for enrollment and attestation since the browser doesn't add any data directly to those redirects' URL.

- The winning ad origin **does not** need to be enrolled for the following reasons:
    - Irrespective of whether the winning ad is rendered in an iframe(temporarily) or in a fenced frame, to be able to communicate user information with the embedding site, it will require collusion. Since the embedding context (SSP) is already required to be attested, I think we should be fine with not requiring the winning ad’s origin to be attested.
    - Requiring winning ads’ sites to be enrolled is a huge lift on the ecosystem, since sometimes the ad’s origin is an ad tech but other times it could also be an advertiser and requiring advertisers to be enrolled is a big ask.
    - Fenced frames also technically enforce no communication between the FF and the embedding context. The existing privacy side channels e.g. network side channel will need to be enforced technically as well, post 3PCD.

- 3PAT verification parties receiving the report via the mechanism proposed in https://github.com/WICG/turtledove/issues/477. If a given 3P is not enrolled, the report will not be sent to that destination.
    - If the destination decides to redirect, it's that site's responsibility to share any data in the redirect URL as per the attestation guarantees. The redirected site is not checked for enrollment and attestation since the browser doesn't add any data directly to those redirects' URL.

Shared Storage

Shared Storage has 2 output gates and both of them will be gated behind enrollment:
- Origin of context invoking selectUrl and addModule

Will be gated via changes to PrivacySandboxSettings::IsSharedStorageAllowed

Topics

document.browsingTopics() is called in the context of a document. It will be gated by the document's origin's site. Topics has an additional API surface which strongly binds the resulting topics not to a document but to a subresource request via a Fetch extension which attaches topics as request headers:
fetch(<url>, {browsingTopics: true});
In this case, we can apply API gating on the site of <url>.
Will be gated via changes to PrivacySandboxSettings::IsTopicsAllowedForContext

The above should automatically work with the changes in PrivacySandboxSettings::IsTopicsAllowedForContext.

ARA
ARA invocation will be gated on the enrollment status in the following scenarios:

- The URL of the HTTP request for HTTP-based registration
    - The redirects on the requests are supported, and the stance there is that we check if each site in the chain is enrolled. If we encounter a site that is not enrolled, we skip that one (i.e. ignore the headers containing the params used by ARA), but do continue on to the next site in the chain and continue.
- From the execution context's origin for JS-API invocation
- Before sending any reports, to ensure that reports are only sent to valid enrolled origins.

These call-sites match the notion of a "reporting endpoint" in ARA.

Will be gated via changes to PrivacySandboxSettings::IsAttributionReportingAllowed

Private Aggregation API
Private Aggregation API invoked from the Protected Audience API or SharedStorage worklets will automatically be gated behind enrollment for those APIs since the respective worklets are gated. Additionally, there will be separate checks needed for PAAPI calls since attestation is per API but that should be handled via checks in PrivacySandboxSettingsImpl::IsPrivateAggregationAllowed(). Any other contexts that allow invocation of PAAPI will need to check enrollment gating similarly via invoking PrivacySandboxSettingsImpl::IsPrivateAggregationAllowed()
The above should automatically work with the changes in PrivacySandboxSettingsImpl.

Error reporting

The enrollment and attestation framework will do minimal error reporting, like logging to dev tools console when a check for a given site fails for an API. The individual APIs however can choose to do their own level of reporting.

Core principle considerations

Speed/Memory

Chrome's requirement is that no change should regress the speed launch metrics. Out of the many speed launch metrics, the ones that we need to track for this change are the following:

Loading/Responsiveness

- Given that enrollment affects ads APIs, we do not envision any impact on the main page's NavigationToFirstContentfulPaint unless the page is set up to invoke ads APIs in the critical loading path. Either way, it should not regress.
- Similarly, PageLoad.InteractiveTiming.FirstInputDelay4 and other interactivity metrics should not regress. The enrollment list check happens on the browser process's UI thread, and will be a synchronous lookup of the map and should be fast enough.

Memory

Chrome requires the following information for each enrolled site:

- Site
- For each API gated on enrollment: enabled or not

The estimated size of the list is O(1000)s of sites. This back of the envelope estimate of the number of sites is based on the fact that the main consumers are going to be ad-techs.

Some manual testing shows that the in-memory map for 1000 sites will be ~100KB and for 10000 sites will be ~1000KB. We are expecting the number of sites to be somewhere in that range.

Memory.Total.PrivateMemoryFootprint Memory.Browser.PrivateMemoryFootprint would be good to track. Regarding the other processes' metrics, like Memory.Renderer.PrivateMemoryFootprint, Memory.Gpu.PrivateMemoryFootprint: Since the only process relying on the enrollment list in memory is the browser process, none of the other processes' memory should be affected. Browser process's memory impact from this should be minimal as discussed in an earlier section.

Browser startup time

- Startup.FirstWebContents.NonEmptyPaint2 and the Android counterpart should not regress, given that at startup the enrollment list will be parsed and an in-memory list is created.

Security

The feature follows the Rule of Two, since it doesn't have untrustworthy inputs but has unsafe implementation language (C++) and High Privilege process (the browser process).

Also, there is no dependency on third_party or from google-3 code.

The backend enrollment system (not in Chrome), will validate that the enrolled site is attested by checking the file at the well-known path (served at <site>/.well-known/...). Thus enrollment policy configured by one origin impacts web visible behavior by other same-site origins.

Stability

No specific stability concerns.

Simplicity

The following considerations keep the overall design simple:

- Only checking the enrollment status in the browser process.
- Only acknowledging the enrollment list as the source of truth and the browser not duplicating the server's work of verifying the attestation file.
- Even though baking the list in the browser (like the public suffix list) would have been simpler, it would lead to chrome release schedule driven latency for any newly enrolled site to be reflected and would also lead to an increased binary size.

Efficiency

There is no change expected on the binary size except for the added code which should be small. Also, as discussed in above sections, it will have medium impact on the memory usage (in the range of 100 to 500KB on average "medium" t-shirt size as per go/chromedesigndoc)

Privacy considerations

This feature does not do any user data collection or sending user data to Google servers. It is pulling data from Google servers to be able to get the sites that are allowed to invoke Privacy Sandbox APIs.
This feature does not have any special considerations/behavior for incognito mode.
We do not plan to collect any URL-Keyed-Metrics for this feature.

Metrics

Success and Regression metrics

The metrics that will be tracked are:

- Metrics detailed in the Speed section.
- **Default-deny percentage:** The metric as described in the section "Fallback if component is not present".
- Parsing overhead metrics (tracked in https://bugs.chromium.org/p/chromium/issues/detail?id=1464311)

Experiments

Given that the startup time and memory usage metrics are not expected to be impacted enough to show up in the metrics and that we can locally measure the impact on memory and startup, we don't need a finch experiment for this feature.

Testing plan

No special testing. All code will be tested via automated browser tests.
For completeness but out of scope of this document: The end to end enrollment system, including the server side backend process that enrolls an entity, will require manual testing though.

Rollout plan

Waterfall rollout (standard dev-beta-stable progression).

Follow-up work

- **Fast-Follow:** Fallback plan if the metrics show a >1% of API invocations denied due to the time lag between startup and the time when the in-memory list is available to look up. If that happens we would need to consider alternatives to the synchronous approach that we currently have.
  - The first navigation waits for the component to be parsed in-memory before the navigation is committed. It can likely be done in parallel with the network request for the navigation. This likely requires less refactoring than the alternative below since deferring commits is supported via CommitDeferringCondition.
  - The alternative is to make the first API invocation block on the list to be parsed in-memory.
- **Fast-Follow:** Dev tools integration

- Analyze if Android Webview needs to be supported since certain APIs like ARA have plans for that. (Component updater support for WebView needs additional changes).
    - ARA should be fine since it is delegating to the OS/Android, where enrollment is checked.
    - If/when other gated APIs start being supported on WebView or if APIs already supported on WebView (e.g. private state tokens) start being gated, then we will need support for WebView.
- Analyze if private state tokens API should be gated behind attestations. [From svaldez@] There were some initial discussions but the idea is to not currently gate it behind enrollment. Possibly once the API is more mature and we have a better idea of the scope of valid use cases, enrollment might be a good idea to add as a requirement when issuers update their registrations.
- Analyze if binary transparency is needed for this component and if yes, its design and implementation. Other Privacy Sandbox components like private state token and first party sets also do not implement it in their initial version but might in the long-term.