Technical Design Document for a Browser Extension to Block Online Trackers

Background & Requirements

This document aims to provide a technical design to build a browser extension to block tracker domains from tracking user activity across multiple site visits. Trackers set third party cookies on various sites with user information or capture user metrics like browser, browser version, window size, language etc. to form a unique fingerprint of the user and track site visit data against this fingerprint.
Understanding the below mentioned approaches would require basic understanding of first party and third party cookies, browser APIs to intercept network requests at different points in the request response lifecycle (like before request is sent, after response headers are received etc.) and certain browser APIs like canvas API, Battery API, Navigator API which might be used to create unique fingerprint of the user. Some references on topics are added below for a quick read.

First party and third party cookies - https://clearcode.cc/blog/difference-between-first-party-third-party-cookies/
Request response lifecycle - https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest
Browser APIs (WebExtensions standard) - https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API

Technical Constraints

1. Given the difference in API for extensions across different browsers - development for each browser will have its own nuances. We will try to mitigate this by using the WebExtension API and bridging the gap through polyfills for certain browsers.
   Polyfill to make WebExtensions API work in chrome - https://github.com/mozilla/webextension-polyfill
   Safari supports the WebExtension API from version 14 onwards.
   Internet Explorer is not considered in the scope due to a disparate API.
   Detailed browser compatibility for WebExtension API - https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Browser_support_for_JavaScript_APIs
2. Other extensions installed in the browser may conflict with the functioning of this extension, especially extensions which intercept and modify the network requests from the browser.

Assumptions

1. Measures to solve fingerprinting like prevention of usage of canvas API, battery API, rotation of user agent etc. are not considered in the scope of the POC. This is purely in interest of the timeline for the POC. However, basic details of how this can be achieved is outlined in the Future Scope section below.

Similar Work for Reference

1. DuckDuckGo Privacy Extension - Prevents third party cookies from a known list of trackers, reduces fingerprinting, disables FLoC. https://github.com/duckduckgo/duckduckgo-privacy-extension
2. PrivacyBadger - This extension blocks third party cookies from a list of trackers. It enhances the list of trackers as it learns about its “tracking behaviour” while the user is browsing.
   https://github.com/EFForg/privacybadger

Problem statement

This technical design aims to identify and block trackers from tracking users online. For the scope of this POC, we will consider 2 major contributors to this problem -

1. Websites can make network requests to tracker domains which in response set cookies on the tracker domain with information about users visiting the current website. This information is then enhanced by the tracker domain as the user visits multiple websites which make a network call to this tracker domain.

2. Tracker domains read information like IP address, user agent and other attributes in the request to form a unique fingerprint of the user and store visit information of the current website against that fingerprint. This information is further enhanced as the user browses sites which make network requests to this tracker.

Approaches

Approach 1 - Prevent setting of third party cookies

1. Intercept network responses using the **onHeadersReceived** lifecycle hook of WebExtensions API in **blocking** mode.
2. Check if the request URL domain / subdomain matches the **documentUrl** parameter's (attribute of the object passed as first parameter in the callback) domain / subdomain. If yes, proceed to step 5. If no, proceed to step 3.
3. Check if **responseHeaders** (pass "responseHeaders" string in the **extraInfoSpec** array parameter to get responseHeaders in callback) have **Set-Cookie** header. If not, proceed to step 5. If yes, proceed to step 4.
4. Strip the **Set-Cookie** headers from the response headers.
5. Allow the request response cycle to continue.

Shortcomings of Approach 1

1. This approach will not prevent requests from being sent to the tracker domain. Hence the tracker domain can still obtain user information from the request like **IP address, user agent, HTTP headers** etc. and may use techniques like fingerprinting to identify users uniquely and track them across sites.
2. Starting from Chrome 72, **Set-Cookie** header cannot be modified or removed without specifying **'extraHeaders'** in **opt_extraInfoSpec** parameter of the onHeadersReceived addListener function. Specifying 'extraHeaders' in opt_extraInfoSpec can have performance impact (as mentioned in chrome documentation)
3. Multiple extensions may listen to the **onHeadersReceived** event and thus lead to conflicts in the final headers returned.

Approach 2 - Prevent network requests to a known list of tracker domains

1. Intercept network requests using the **onBeforeRequest** lifecycle hook of WebExtensions API in **blocking** mode.
2. Check if the request URL domain / subdomain matches the **documentUrl** parameter's (attribute of the object passed as first parameter in the callback) domain / subdomain. If yes, proceed to step 4. If not, proceed to step 3.
3. Check if the domain or any subdomains of the requested URL matches a known list of tracker domains available publicly. If yes, block the request. If no, proceed to step 4.
4. Allow the request response cycle to continue.

Shortcomings of Approach 2

1. Every request's domain and subdomains need to be compared with a list of tracker domains which may slow down the overall resource load time for the user.
2. In some cases, components like social widgets etc. may not be loaded thus affecting website functionality and leading to a bad user experience.
3. Multiple extensions may listen to the **onBeforeRequest** event and lead to conflicts.

Although approach 2 has shortcomings, they have a relatively low user impact compared to shortcomings of approach 1 and do not affect the primary goal of the extension i.e. to preserve user privacy. Also, the shortcomings can be solved to a large extent using certain measures. Below is the point by point proposed solutioning for the above shortcomings.

Resolutions to Shortcomings of Approach 2

1. The performance overhead of comparing each request's domain and subdomains with a list of tracker domains can be substantially reduced by using an appropriate data structure to store the tracker domain list. In this case a javascript object (which acts like a dictionary or hash map data structure) can be used. This will prevent comparing each domain iteratively with the request domain and the comparison can be achieved with a constant time complexity.
2. For social media widgets, a strategy like click to load can be adopted. This will render a click to load widget in place of the social media widget and load the social media widget only when the user intends to interact with it. Thus we are able to provide the functionality of social media widget and also reduce the default tracking. Although if the user intends to interact with the social media widget then the user can still be tracked. However we have reduced the overall footprint of this tracking by preventing default tracking on page load. Also in this case users can be informed that they might be tracked by this social domain. *(Not part of POC)*
3. Similar to approach 1, approach 2 also has a problem of conflict with other extensions. However, in this case the only conflict that can occur is if some other extension cancels a tracker domain request before our extension. This still serves our purpose of blocking requests to tracker domains. Only disadvantage this might have is that it might affect our analytics (i.e. capturing metrics like how many tracker requests were successfully blocked). This can also be mitigated by listening to the **onErrorOccurred** event (which would be fired if another extension cancels the request before ours), checking the request URL domain to match tracker domains list and then logging to our analytics module from this event listener.

Since all the shortcomings of approach 2 can be mitigated, approach 2 seems to be the more favourable choice. We will choose approach 2 for this technical design. Point 2 will not be taken up in the scope of POC due to time constraints and can be taken up as a separate feature in the production implementation phase.

Other Technical Considerations

Tracker Domains List - We will be using the publicly available list of common tracker domains which are updated by DuckDuckGo crawlers regularly in an automated way - https://github.com/duckduckgo/tracker-radar

Browser Caching ***(Not part of POC)***- Browsers like chrome have in-memory cache and requests which are served from in-memory cache are not intercepted by the web request API. Hence if our blocking logic in the **onBeforeRequest** handler changes, we need to call **handlerBehaviorChanged** function to flush the in-memory cache. We need to have a well defined criteria on which to call **handlerBehaviorChanged** since flushing the cache is an expensive operation.

Security -

1. When using content scripts (probably in phase 2 of implementation for features like click-to-load for social widgets etc.) we need to make sure that we don't make XMLHttpRequest on a URL which is provided by content scripts. This is because malicious scripts can forge such messages and force the extension to load content from a URL without browsers CORS policy being applied.
2. When loading unknown HTML or JSON from a webpage, do not run APIs which execute the script like **eval** or **element.innerHTML**. This is because webpages can inject malicious scripts and force the extension to run these. Prefer safer APIs which do not execute the script like **JSON.parse** and **innerText**. Since we are not loading any HTML / JSON / script from the webpage, this shouldn't affect our implementation.

Performance *(Not part of POC)* - Performance metrics can be captured at critical points like the overall average time taken by the callback registered on onBeforeRequest event. The Performance APIs mark and measure methods can be used to measure the execution times and then send it to a backend performance monitoring service. This is not considered in the POC phase as the extension is not performing any heavy computations or network requests which can lead to severe performance issues.

Backward compatibility - Store the version number of the currently installed version of the extension in local storage. This will help in cases where we have to run certain logic whenever the extension updates by comparing the last installed version and updated version of the extension.

Browser compatibility -

Chrome	Firefox	Edge	Safari	Opera	IE
22+	55+	79+	14+	15+	No

Program Execution Flow Diagram (*Use Miro web app to view the below program execution flow diagram)*
https://miro.com/app/board/o9J_lxKZR4I=/

High Level Program Architecture

The program will mainly have the following functional directories and files -

1. **popup.html** - HTML code for the popup when the user clicks on the extension icon. It will support below sections -
    a. Radio button to show if tracker blocking is active on the site in the current active tab. Users can change this option.
2. **popup.js** - This script will be imported in popup.html and will largely perform below functionalities -
    a. Fetch list of whitelisted domains by user from local storage and check if the current site in the active tab is whitelisted. Set the value of the radio button in the HTML accordingly.
    b. Listen to the change event on the radio button in the HTML and if the value is false then add the current active site domain in the whitelistedDocumentDomains object in local storage. If the value is true then remove the current active site domain from the whitelistedDocumentDomains object in local storage.
3. **background.js** - Background script registered in manifest.json. It does following functionality -
    a. Listen to the onInstalled event and update the version number of extension to local storage. Set initial values of whitelistedDocumentDomains, trackerDomains, blockedTrackerLogs as blank objects in local storage.
    b. Listen to the onBeforeRequest event and run the blocking logic. Detailed steps of blocking logic are listed in the execution flow diagram in the above section.
    c. Use the browser.alarms.create API to periodically fetch the tracker radar data of tracker domains and save to local storage. Signatures of these functions are shown below.

```
//Handlers
const handleOnInstalled = (details) => {
    //Save current version to localStorage
    //Set initial values of whitelistedDocumentDomains,
    //trackerDomains, blockedTrackerLogs as blank objects
    //in local storage
}
const handleUpdateTrackerList = (event) => {
    //call utils.fetch to get latest tracker list
    //and save it to trackerDomains in local storage
}
const handleBlocking = (details) => {
    // Check if details.url domain / subdomains matches details.documentUrl
    // domain / subdomains.
    // If above matching fails run the tracker list matching logic
}
//Alarms
```

```
browser.alarms.create('updateTrackerList', { periodInMinutes: 24 * 60 })
//Listeners
browser.runtime.onInstalled.addListener(handleOnInstalled);
browser.alarms.onAlarm.addListener(event => {
   //if event.name equals updateTrackerList call handleUpdateTrackerList(event)
})
browser.webRequest.onBeforeRequest.addListener(handleBlocking, {
   urls: ["<all_urls>"]
}, ['blocking'])
```

4. **utils.js -** Contains generic utilities like fetching data via XHR, breaking domain string into subdomains etc.
5. **models** - This directory contains all modules which will be used as model classes like **storage.js** class which is used to get and set data from local storage, **trackers.js** class which stores tracker data and has utils to match request URL against tracker domains list.

```
class Tracker(){
   constructor(){
       this._trackerList = {}
   }
   checkTrackerBlocking(url){
       //Run matching logic and returns output as true / false
   }
   updateTrackersList(){
       //updates trackers by fetching from tracker radar source
   }
}
class Storage(){
   getFromStorage(key){
       //return value of key from local storage
   }
   setToStorage(key, value){
       // sets value against key in local storage
   }
}
```

6. **constants.js -** This module contains all constants.
7. **manifest.json** - This is the configuration file where popup.html and background.js files are registered. Other extension configurations like name, icon, version, permissions are also configured in this file.

Edge Cases

1. The tracker domains list might not have loaded when the first network request is intercepted. In this case just let the request go through as holding the request till the tracker data loads might affect user experience.
2. The XHR request to load tracker domains list might fail. To mitigate this we can do 3 re-retries at an interval of 1 minute. If the request still fails, we can notify our error monitoring service and also register an alarm to re-retry the request after a longer interval of 30 minutes.
3. Requests originating from within an Iframe from a page should not be blocked by matching the domain of the request with the parent frame document and considering these requests as third party. Instead the request domain should be matched with the Iframe document domain. This will be taken care by matching the request domain with the domain of **documentUrl** attribute provided in the onBeforeRequest callback as an object parameter. This attribute points to the url of the document in which the resource will be loaded.

Validation of Approach in Dev Testing

In the dev testing phase, the approach will be validated by visiting a list of commonly used sites and checking that the extension blocks tracker domains on these sites. We also need to validate that the basic site functionality and interaction is working fine. This can be initially done manually and then further evolved to use a UI automation framework like Selenium, Codecept, Cypress etc. to automatically test a list of sites for blocked requests to tracker domains and use techniques like screenshot matching and click interactions to make sure that the site is not broken.

Further Validation of Approach in Production Environment *(Not part of POC)*

This is not part of POC implementation but is being considered so that the core architecture can be built to accommodate these features in the future.

1. Pass logs of blocked tracker domains and the referrer (anonymously without capturing user info) to a backend service which saves it in the database. Run an automated script on a daily basis on this database to cross check if the blocked list of trackers matches the tracker domains list in tracker radar data.
2. Run automated tests using UI automation frameworks like Selenium, Codecept, Cypress etc on the referrer domains in the logs stored in DB (as mentioned in point 1). Check if the blocked domains listed against the referrer in the logs are actually setting third party cookies. If yes then our extension has correctly blocked these domains. If not then introspect further to check why these domains are being blocked.
3. After 15 days of installation or after blocking 100 trackers, show the user a question in the popup section (whenever the user opens the popup next time) to take user feedback whether the user has visibly noticed less cross site ads or tracking across sites compared to earlier experience.

Experiments to Improve the Product *(Not part of POC)*

This is not part of POC implementation but is being considered so that the core architecture can be built to accommodate these features in the future.

1. Whenever a user whitelists a site to allow tracking, ask user feedback whether this change was due to the site appearing broken. If the user answers as yes, this information can be passed to the backend servers to store the information and further analysis can be done on a case by case basis as to why the site was breaking.
2. A feedback loop can be established with the extension and the tracker radar data to increase the number of tracker domains being blocked. For requests which are not blocked by the extension, in the **onHeadersReceived** event check if third party cookies are being set. If yes, then report such domains for further analysis to a backend service which can further pass on this data to tracker radar data.
3. Additional features in the extension like prevention of fingerprinting, disabling FLoC, click to load for social widgets etc. (mentioned in future work section below) can be rolled out gradually to users using an Experiments module. A master list of experiments will enable these features for certain user cohorts (like say for only chrome users or firefox users on desktop etc). The success of this experiment will be tested using analytics and user feedback through survey form. If the experiment is successful, the feature is validated and can be rolled out to larger cohorts.
4. Analytics of interaction and visibility of various UI elements can be anonymously captured against certain attributes like device type (desktop, tablet, mobile), browser, screen size to understand the usability of the various UI elements in different user environments.
5. Custom AB testing module to be developed to put users anonymously into 2 cohorts and test various hypotheses on UI elements like type of element, colour, placement etc. and check conversion against each hypothesis.

Technologies to be used

- Language - Javascript (ES6), HTML, CSS
- Build tool - Webpack
- Linting - ES Lint for code linting, commit lint for commits *(Not part of POC)*
- Pre-commit checks - Husky and lint-staged *(Not part of POC)*
- Node package manager - npm
- Test framework - Karma *(Not part of POC)*

Future Scope *(Not part of POC)*

This is not part of POC implementation but is being considered so that the core architecture can be built to accommodate these features in the future.

1. Provide users option to login so that user preferences and whitelists can be stored across devices. (This will be just an option. Users can still use the extension without login).
2. Implement techniques like user agent rotation, prevent canvas api and reduce uniqueness of values provided by other browser APIs like navigator API, Battery API etc. to reduce the probability of tracker domains uniquely identifying users based on these parameters through fingerprinting.
3. Disabling FLoC interestCohort API in chrome browsers by deleting the interestCohort from the document .
4. Provide developer tools to enable developers to easily debug and monitor requests, domains being blocked.
5. Integrate open source error monitoring tools like Sentry to proactively monitor, debug and resolve errors.
6. Replace social widgets with click to load widgets which the user can opt to load rather than being loaded by default thus preventing tracking on load by these widgets.
7. Enable mechanisms to reduce tracking via first party cookies like reducing expiry time of such cookies.
8. Use the surrogate scripts provided in tracker radar data to prevent certain sites from breaking.