Foundation > Capabilities

File Handling - Security Model

SUMMARY

A document that details the implementation of the File Handling API for security and privacy review

Authors	huangdarwin@chromium.org, estade@chromium.org
Contributors	cmp@chromium.org, estade@chromium.org, mek@chromium.org, mgiuca@chromium.org, oyiptong@chromium.org, harrisjay@chromium.org, jsbell@chromium.org
Team	pwa-dev@chromium.org

Status	**Draft** | Final
Created	2020-11-18
Last updated	2022-01-07
Launch Bug	https://crbug.com/1157254
Short Link	https://tinyurl.com/file-handling-security-model

This Document is Public

PLATFORMS

Desktop (MacOS, Windows, Linux, Chrome OS)

BACKGROUND

Powerful web applications need to express their ability to read and edit files from the file system on the user's device. Once a web application expresses this ability, it should be listed alongside native applications with similar abilities in host operating system surfaces, such as "Open with…" dialogs and context menus.

File Handling targets two primary use-cases:

1. Supporting existing web-based creative PWAs, such as GSuite. These PWAs need a frictionless way to import and export the user's work between the host operating system and other native applications.
2. Porting creative native applications, such as Photoshop, to the Web. PWA versions of these applications need a frictionless way to import the user's existing work.

We propose to provide installed Desktop Progressive Web Applications (dPWAs) access to this capability so they too can correctly interoperate with the user's operating system and file manager. This feature will be named "File Handling".

The File Handling API adds a new persistent capability to installed dPWAs on the Web platform: read/write access to a file the user selected outside of the browser. This API also some systems to apply descriptive icons and human-readable type names providedby dPWAs that have become default handlers for the relevant file type. File Handling is a feature depending on the File System Access API, as the file is exposed to dPWAs using a `FileSystemFileHandle`.

Chrome will protect the user's security and privacy by gating the new capabilities with the following measures:

1. The API will only be exposed to secure contexts (installed PWA requirement).
2. Granting access to a file will require a user decision on the host operating system
3. Granting access to a directory will not be supported.
4. The browser will show a confirmation prompt the first time a dPWA is used to handle opening a file, to avoid spoofing or unintentional file handler use, which is

especially (but not exclusively) concerning when a dPWA has “silently” become the default handler for a file type.

DETAILED PROPOSAL

The File Handling API will allow dPWAs, when being installed, to include in their manifest file types (both MIME types and file extensions). Chrome will register the dPWA with the OS as a file handler for each provided file type. Please see the design document or API explainer for more detailed information, or the File Handling Icons design document for information on this extension to File Handling.

In later versions of the implementation, on some systems such as Mac and Windows, Chrome will also register any icons and names provided by the PWA. These will be used to describe files in system surfaces such as the file handler, but generally only when the PWA has become the *default* handler for the associated file type.

Install Pipeline

To gain access to File Handling, a user must install a site as a PWA, which means the site must meet installability requirements. The OS integration step is executed as part of installation without any user-visible confirmation.

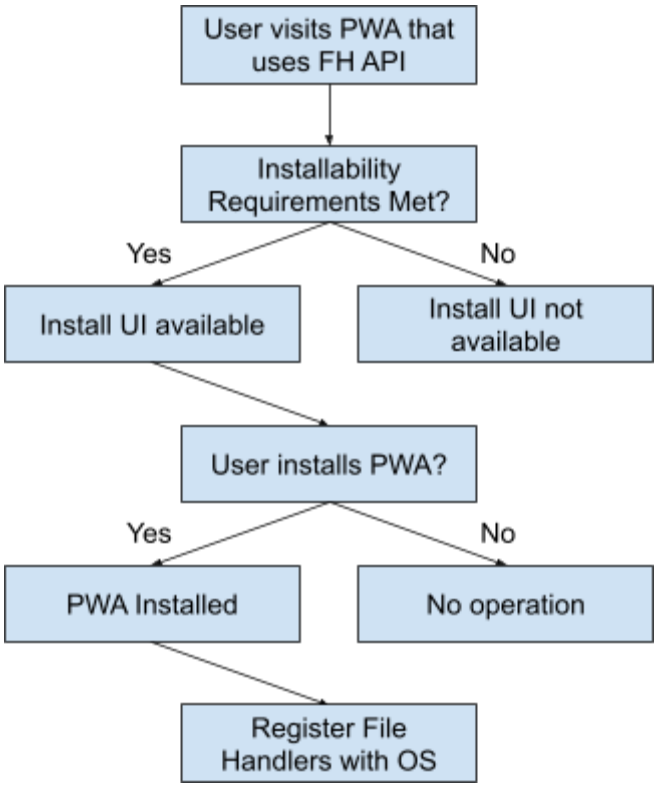

Figure 1: Install Flow

Invocation Flow

To gain access to a file via File Handling, a user must use existing OS UX flows to select a file. While these may differ slightly per platform, they will likely involve a context menu with an “Open With…” submenu, where the user must pick this installed PWA from a selection of applications that can handle this file. A PWA will only show up as an option here if they are registered as a File Handler for this type.

Currently, some platforms may set a PWA as the default handler if no other handler for a type already exists, so a double-click on a file can open a file in a PWA without explicit user choice. A prompt will be shown the when the file is thusly opened to ensure the user intended to open that file using this PWA. The prompt will contain an option to suppress future such prompts, either by automatically permitting the action or permanently denying and unregistering the PWA as a file handler. This setting can later be revised in the [App Settings page]. It will also be possible to suppress this prompt via admin policy.

Figure 2: Example Flow on MacOS.

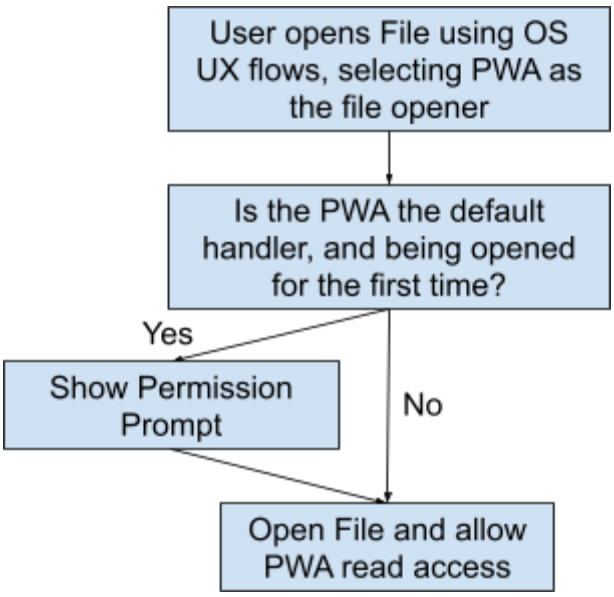

Figure 3: Invocation Flow

Deregistration

All File Handlers and Icons previously registered by the PWA will be deregistered when the user chooses to uninstall the PWA, as part of the uninstallation process. This data will also be removed if/when the user opts out of the FH API (selects “Don’t allow” and “Don’t ask again” from the prompt).

SECURITY RISKS VS BENEFITS

Providing read access to the user’s files and registering file handles to the operating system introduces new security risks, but these should be taken in the context of existing alternatives, which may be no more secure or even less secure than what the PWA platform can provide.

ALTERNATIVES WITHIN WEB PLATFORM

Read access to the user’s files is already possible through drag-and-drop, `<select type="file">`, or File System Access API pickers. The File Handling API adds new entry points from various native surfaces such as a file browser or Chrome’s download shelf.

Regarding write access to user files, the risk/benefit analyses made in the File System Access API apply here as well. File System Access is closely related to the File System Access API as it provides access to files in an expanded manner. In fact, the File Handling API uses `FileSystemHandle` from the File System Access API and therefore inherits its properties. See relevant discussion in File System Access spec. See also discussion of a more restrictive version of the proposal.

ALTERNATIVES OUTSIDE WEB PLATFORM

The API will make it easy and convenient to work with files using dPWAs, which will migrate some workloads from other platforms to the Web. We expect to improve security in many cases since the web generally uses a more restrictive security/privacy model. For example:

- Applications that use Electron instead of the web primarily to gain file handling access, such as VS Code. Electron has been shown to have serious security risks, due to poor update rates and increased severity of cross-site scripting attacks. This isn't to say that Electron is bad, but if we can support web developers in Chrome with a safe install model, it will be safer.
- Small utilities, such as archivers and image/PDF resizing/cropping, are often native applications that run with full user privileges (including the ability to read, write, and upload all of the user's files, of all file types). Some of these applications deliver malware.
    - It's less risky to use an application that only has access to specific file types, and only to files the user has explicitly chosen.
    - The user must still be aware that the dPWA may keep a copy of what it processes on behalf of the user.

THREAT MODEL

We are concerned about the following attacks. In the examples below, Alice is a web user, and evil.com (evil.com.example.com) is a seemingly benevolent site that will act maliciously if given the opportunity to do so. Numbers and short summaries are used to more easily reference each threat, but these threats are not numbered in any particular order.

To protect from each concern, different mitigations will be in effect. Some are already implemented by the underlying File System Access API or installed dPWA infrastructure, whereas others will be newly introduced as part of File Handling. Mitigations are lettered as subsections after a numbered threat.

1. **Comprehensibility**: Alice installs evil.com without understanding the File Handling implications. Example: Alice may follow some guide online to install evil.com, then open a sensitive file using evil.com, unintentionally granting evil.com access to the file. evil.com may now extract this sensitive information.
    a. Planned Mitigation: It is assumed that installation of a PWA implies a strong signal of user trust, and that users can generally expect installed applications to be able to install file handlers. Therefore, file handlers

cannot be registered until a PWA is installed.

   b. **Confirmation Prompt**: A confirmation dialog will be shown on all platforms when the PWA is first opened using a file handler. This would allow Alice to give explicit permission for the PWA to access the referenced file(s). This confirmation prompt shall be shown before the PWA is launched, and if denied, the launch will abort. This may also partially help with (2) Recall and (3) Recall after update, and (5) Spoofing, especially if the time between install and first use is long.

2. **New behavior after update**: Alice installs evil.com for a legitimate purpose. evil.com later updates their manifest to have different File Handlers.
   a. If Alice has already vetted the app as a file handler on a permanent basis, the decision will be reset to the initial ("ask") state when handled file types change.
3. **App identity spoofing**: Alice installs evil.com. evil.com gives itself a name and/or icon identical or similar to a trusted application (like "Microsoft Word"), pretending to be the trusted application and getting access to files when Alice chooses the spoofed version of "Microsoft Word" in the "Open With…" context menu item.
   a. The confirmation prompt should mitigate spoofing by including the name, icon, origin, current files, and associated file types. The name and icon are considered somewhat vetted during the installation process (and/or the identity update dialog), but the origin is the ultimate signal of trustworthiness.
   b. Icon spoofing mitigations:
      i. The icon will only be visible if the PWA is the default handler for that file type. If a user has sensitive documents of type .foo on their system, they probably already have a different default handler for them.
      ii. The user must open a file (perhaps of a different type) with the PWA, then grant it permission to open files, before it can read any file. So the malicious PWA spoofing a legitimate application has to be able to trick the user into opening a file using a different application than expected at least once, and also having the user accept the permission dialog, before the permissions dialog is bypassed in future file opens.
   c. (As with other rejected mitigations, rejected mitigations for spoofing are shown in the "Registered PWA name spoofing mitigations" and "Icon Spoofing mitigations" sections.)
4. **Unexpected Handles**: Alice installs evil.com to handle files of type ".foo". However, evil.com also adds itself as a handler for unintended sensitive types like ".docx".
   a. The confirmation prompt lists all handled file types.

5. **Excess Handles**: Alice installs evil.com to handle files of some type. evil.com registers itself as a handler for an excessive amount of types, until OS resources are exhausted.
    a. No mitigation planned. Chrome will rely on existing OS safeguards to prevent resource exhaustion, as (accidentally) malicious native apps would have the same issue.
6. **Accidental Default**: Alice double clicks a file in the operating system's UX, not understanding that evil.com is the default handler. evil.com now has read access to the file.
    a. Planned Mitigation: To avoid unintentionally giving access to sensitive information to malicious websites, Chrome should strive to avoid allowing PWAs to become the default handler, although this depends on the OS. The user will then have to click the equivalent of `Open With…` and select the desired PWA.
7. **Insecure Context**: Alice installs example.com in an insecure context. Because example.com is using an insecure context, evil.com injects unwanted scripts using a MITM (man-in-the-middle) attack into example.com, and is installed into Alice's device with File Handling access via example.com. evil.com could also identify that example.com is installed on Alice's device.
    a. Planned Mitigation: To avoid providing access to vulnerable insecure contexts, the underlying PWA install flow requires sites to have secure context.
8. **Access to sensitive files**: Alice installs evil.com and opens a sensitive file containing system diagnostic information or password dumps. evil.com may then read and send this sensitive information to an external server for identity theft. evil.com might also be able to identify the user, organization, or related users from information in these files.
    a. Planned Mitigations: There are no plans to selectively block access to extra-sensitive file types. In particular, formats like .txt or .docx may be commonly used, but also potentially sensitive, as they may hold user passwords and PII. The user in this case is deliberately sending information to the site, after which misuse is difficult to guard from.
9. **Persistent access request**: Alice is repeatedly requested to install evil.com, in order to get File Handling capabilities. Alice eventually installs evil.com in order to stop seeing the spammy request evil.com shows to install, giving evil.com File Handling access to Alice's files.
    a. Planned Mitigation: To avoid persistent access requests, PWAs will not be able to receive any indication that their handlers are registered, nor whether they are the default handler for any file type. This incidentally also gives user agents some leeway in deciding to register or unregister file types deemed appropriate or inappropriate, or for users to manually override certain associations.

10. **Embedded Content**: Alice installs example.com. example.com trusts and embeds evil.com for a legitimate reason (ex. Utility, ad network, tracker, etc). evil.com becomes compromised.
    a. See File System Access spec.

REJECTED MITIGATIONS

1. **Registered PWA name spoofing mitigations**:
    a. Disallow use of OS-registered PWA-specified names: To alleviate Threat (5) Spoofing, the application name shown by OS UX flows could be required to be the domain name, so that a site like “evil.com.com” couldn’t purposely register a PWA named “Bank Application”. This was rejected because domain names may be less understandable / user-friendly (and can still be partially spoofed, for example by using “banks.com” instead of “bank.com”).
    b. Append “Chrome” to PWA name: To alleviate Threat (5) Spoofing, the application name shown by OS UX flows could be appended by “(Chrome)”, so that a PWA registered as “MyApp” would be shown in OS UX as “MyApp (Chrome)”. This could prevent a user from mistaking the PWA for a native application, and could allow different browsers’ PWAs from being mistaken from each other. This was rejected because
        i. It would not be helpful in the PWA-is-default case, since the name is not displayed.
        ii. This may be bypassed by long application names, as some platforms display only a short portion of the name (often followed by “...”)
        iii. This would likely diminish adoption of the PWA platform. (See less secure alternatives.)

    Thus the permission prompt used ended up being a much stronger and complete mitigation.
2. **Icon Spoofing mitigations**: As discussed in the File Handling Icons design document, several mitigations for Threat (5) Spoofing via the registered icon exist, but they were rejected because it is important to allow sites to be able to register different icons for each file type association. Such mitigations include:
        i. Reusing the PWA site icon as the installed PWA’s file handling icon.
        ii. Allowing only 1 File Handling icon to be specified per PWA.
        iii. Showing the PWA-provided icon with a smaller Chrome icon in a corner.
        iv. Using a default blank “PWA” placeholder icon.
3. **Safe Browsing**: To alleviate threat (12) Access to sensitive files, safe browsing could be implemented to scan for and blocklist dangerous file types. This wasn’t implemented because types that may be dangerous, like .docx or .txt (which can

hold sensitive PII), are also common/primary use-cases for file handling. Therefore, a useful scan would also need to potentially scan the file hashes.

4. **Type allowlist**: Similar to Safe Browsing, we could create an allowlist for formats allowed and deemed "safe" by the security team, to use with File Handling. This was rejected because this would severely limit the utility of File Handling, which sites may want to use for custom, arbitrary formats. If an allowlist were to be used, the long tail of formats would all be rejected, and sites may encode complex information inappropriately in simpler "safe" formats, like .txt.
5. **Require reinstall to update file handlers**: To alleviate threat (4) New Behavior after update, and prevent installed PWAs from registering new file handles or changing file icons after install, file handles could only be updated after install. They would then require uninstall and subsequent re-install to update file handlers. This was rejected because it would provide an unfriendly UX, as reinstalling requires several manual steps that may differ between different browsers.
6. **Sensitive directories**: To alleviate threat (12) Access to sensitive files, file handling could protect the user from providing access to some sensitive files, by blocking access to certain files in certain blocklisted directories. This could be implemented in a similar way as FSA, as listed in the FSA mitigations section, which currently limits this only in file pickers (but not in drag-and-drop, etc). This was rejected because, like drag-and-drop, opening a file via File Handling is a user action done from native UI.
7. **Read only file handle**: In earlier versions of the proposal, the handle provided by the File Handling API only had read access. In the open-edit-write model common to editor apps, write access would then need to be procured by a separate call to `FileSystemHandle.requestPermission()`. However, this was judged to add significant extra friction without adding meaningful extra security. Users expect native apps that have been opened in this way (from a file browser, to act on a file) to be able to both read and write the given file(s), and thus their response to the initial confirmation prompt is given in the context of R/W access. Additionally, the prompt will clearly communicate this detail.

DOCUMENT HISTORY

For living documents, you may want to keep a history of significant revisions. When you add a significant new revision to the document, add a new line to the table below.

Date	Author	Description	Reviewed by
2019/08/01	harrisjay	Early Mitigations Document	
2020/04/06	oyiptong	Security Model Initial draft	

2020/11/19	huangdarwin	This document	mek, pwnall, jsbell
2022/01/07	estade	This document	

Related docs:

- Explainer
- Design Doc
- File System Access Explainer
- File System Access Security Model
- Intent to Implement
- Subtopics:
    - File Handling - Manifest Updates Design
    - File Handling Icons
- Promotability and Installability of PWAs