International Journal of Information Security
https://doi.org/10.1007/s10207-022-00610-w

REGULAR CONTRIBUTION

Are chrome extensions compliant with the spirit of least privilege?

Pablo Picazo-Sanchez[1] · **Lara Ortiz-Martin**[2] · **Gerardo Schneider**[1] · **Andrei Sabelfeld**[1]

Abstract
Extensions are small applications installed by users and enrich the user experience of browsing the Internet. Browsers expose a set of restricted APIs to extensions. To be used, extensions need to list the permissions associated with these APIs in a mandatory extension file named manifest. In particular, Chrome's permission ecosystem was designed in the spirit of the least privilege. Yet, this paper demonstrates that 39.8% of the analyzed extensions provided by the official Web Store are compliant with the spirit of least privilege. Also, we develop: (1) a browser extension to make aware regular users of the permissions the extensions they install; (2) a web app where extensions developers can check whether their extensions are compliant with the spirit of the least privileged; and (3) a set of scripts that can be part of the vendors' acceptance criteria such that when developers upload their extensions to the official repositories, the scripts automatically analyze the extensions and generate a report about the permissions and the usage.

Keywords Browser extensions · Web security · Privacy

1 Introduction

Extensions are small and powerful applications that users can install in most web browsers to enrich the experience of browsing the web. Concretely, extensions in Chromium-based browsers (e.g., Chromium, Chrome, Opera, Brave, and Edge) are classified into *apps* and *browser extensions*. Extensions are stored in private repositories that most vendors manage and users can install extensions from them. One such example is Web Store, the largest browser extensions repository provided by Chrome containing 200,381 extensions as of December 2019.

Google recently released a timeline to stop supporting apps. The Web Store did not accept more apps since March 2020—the existing ones can be updated until June 2022. In June 2021 the support for apps on Windows, Mac, and Linux will be removed, and finally, the support on Chrome OS will be ended by June 2022 [10]. In contrast, browser extensions will still stay on course.

Extensions have to include, at least, one special and mandatory file called `manifest.json` where the options and the configuration of the extensions are defined. In particular, one such option is the set of permissions that the extensions have access to. Concretely, extension permissions are used to determine which APIs are exposed to the extensions. According to the official documentation provided by Google, only apps and browser extensions can effectively use the permissions they declare in the manifest [14,16]. The way the extensions define permissions is by listing them in the `permissions` or in the `optional_permissions` keys of the manifest file.

There are three types of permissions: *API*, *match patterns*, and *manifest permissions*. API permissions grant extensions access to a specific API, e.g., cookies, and storage. Match patterns are a particular case of regular expressions and provide access to a set of URLs, e.g., http://*/*, https://*/*, and <all_urls>. Finally, manifest permissions are explicitly defined in the manifest of the extensions and provide access to a particular capability, e.g., *clipboardRead, clipboardWrite, webRequest*. Google considers API and manifest permissions equally, as there are no big differences between them.

Extensions cannot use all APIs that Chrome exposes. For instance, both apps and browser extensions can use `chrome.cookies` API by defining *cookies* permission. On the contrary, `chrome.hid` API can only be used by apps (by using the *hid* permission) whereas `chrome.history` only by browser extensions. Therefore, we can classify extensions

✉ Pablo Picazo-Sanchez
pablop@chalmers.se

[1] Chalmers University of Technology, Gothenburg, Sweden

[2] Madrid, Spain

Published online: 04 September 2022

according to the API they have access to, i.e., the permissions they define.

While this classification might make sense from a technical point of view, it does not say much for regular users. For instance, when Alice is about to install an extension, she is warned by the browser because the extension requires the *notifications* permission. However, she is not warned when the same extension requires both *webRequest* and *webRequestBlocking* permissions that can alter and block network traffic, respectively. A list of permissions that warns the user at installation time can be seen in Google's online documentation [18].

Chrome's permission ecosystem was designed in the *Spirit of Least Privilege (SoLP)* [5]. The idea behind it was to limit the number of available resources exposed to extensions due to the sensitive information they can have access to. In addition to that, by defining specific permissions to extensions and isolating their execution into separate environments—also known as isolated worlds, the damage that malicious extensions may cause is also bounded [5].

Even though we found several similarities between extension and Android permissions [12] like the existence of a set of permissions that warns users, there are (at least) two main differences between these systems: all-or-nothing strategy and permission revocation. Extensions follow an all-or-nothing strategy, i.e., users have to agree at installation time on all the permissions the extension defines if they want the extension to be installed. Once they are installed, extensions can access the APIs that these permissions allow until they are uninstalled.

The second big difference is the ability to revoke permissions at runtime in Android apps, being this not possible in extensions nowadays. Recently, Chrome implemented an option by which users can define the scope of the extension, being possible to limit the execution of the extension to "On all sites," "When you click the extension" and "On the site you're currently on."

Contributions In this paper, we study in detail the permission ecosystem of extensions. Although the existence of overprivileged extensions has been pointed out in the past [8,13,22,28,30], our work is the first systematic effort to investigate the problem in depth. We conclude that, at installation time, 48.3% of the scrutinized extensions (apps + browser extensions) are compliant with the SoLP while 8.5% are underprivileged. In more detail, our contributions are:

- We analyze over 17,566 apps, 136,018 browser extensions, and 17,343 themes, extracting the permissions they define as well as the API calls they include in their files and conclude that 54.7%, 37.8%, and 99.9% of each type, respectively, are compliant with the SoLP (Sect. 5).
- We certify that 8.5% of the extensions are underprivileged, i.e., they contain JavaScript code to access to restricted Chrome APIs without defining the required permissions (Sect. 6).
- To spread our findings, we: (1) develop a browser extension to aware regular users of the permissions the extensions require; (2) create a web app where extensions developers can check whether an extension is compliant with the SoLP, and; (3) make our scripts public to be integrated as part of vendor's acceptance criteria (Sect. 7).

The rest of this paper is organized as follows. We introduce some concepts of browser extensions and the permission ecosystem that Chrome implements in Sect. 2. We extract permissions from the official documentation and from Chromium's source code and show which permissions can be used by each type of extension (see Sect. 3). In Sect. 4, we explain the methodology we follow to analyze browser extensions. In Sect. 8, we present the threats to validity. We discuss the limitations and future work in Sect. 9 whereas a summary of the main proposals published on this field can be seen in Sect. 10. Finally, Sect. 11 concludes the paper.

2 Extensions

Extensions are small applications that either add new functionality to the browser or modify its appearance. In Chrome, every extension stored in the Web Store has its own unique 32 characters long identification that does not change across versions. Once a user starts the installation procedure, Chrome downloads the `.crx` package of the extension to a temporal directory, it extracts all the files and parses the only mandatory file, the `manifest.json`. Finally, Chrome detects the family that the extension belongs, moves it to a permanent directory, and installs it in the browser.

Chrome classifies extensions into two main groups: apps and browser extensions (see Fig. 1). Apps are divided into packaged and hosted; additionally, packaged are split into legacy and platform. On the other hand, themes are a subset

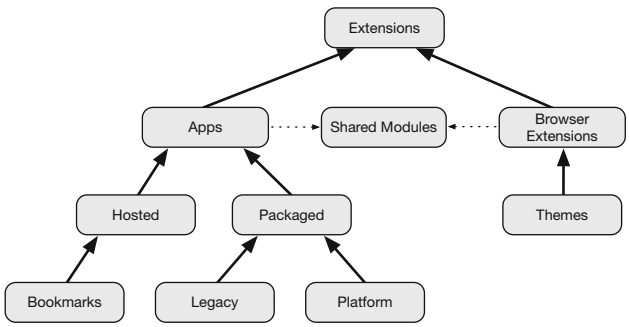


Fig. 1 Taxonomy of extensions in Chromium-based browsers

of browser extensions. Both apps and browser extensions can share common resources by defining Shared Modules.

Shared Modules are not extensions but common resources (e.g., JavaScript, HTML, images, CSS) that can be accessed by many browser extensions or apps. When the extension imports the shared modules (by including a reserved path `_modules/<shared_module_id>` in the root of the extension), all the granted privileges of the extension are automatically inherited by the shared modules' resources. They cannot be manually installed by users but are automatically installed when another extension requires them. Similarly, when the user uninstalls the extension that the shared module depends upon, it is automatically uninstalled as well. They are identified by including the `export` key in the manifest file of the extension.

For example, to include the script "foo.js" from a shared module with ID <shared_module_id>, it can be used from the extension as `<script src="_modules/<shared_module_id>/foo.js">`. Being the full URL to resources like: `chrome-extension://<extension_id>/_modules/<shared_module_id>/`. However, after parsing all the manifest files of our dataset, none of the analyzed extensions use Shared Modules.

In the following, we explain in more detail both apps and browser extensions.

2.1 Apps

Apps are standalone extensions (i.e., they do not cooperate with other apps) that run in the browser with a dedicated user interface. There are two types: *hosted* and *packaged*. Hosted apps include in the manifest a URL that is launched when the user opens them whereas packaged apps contain all the needed files to run the application in the `.crx` package. The latest version of Google Chrome at the time of writing (i.e., 97.0.4692.99) still allows users to install and use App extensions.

Hosted Apps. Hosted apps are split into *hosted* and *bookmarks*. Hosted apps consist of the manifest file having: (i) the URL to be launched; (ii) a list of associated URLs, and; (iii) a list of permissions. They are identified by the `app.launch.web_url` key in the manifest, which provides the URL that is loaded when the user opens the app. Listing 1 shows part of the manifest file of "WeVideo," whose id in the Web Store is *okgjbfikepgflmlelgfgecmgjnmnmnnb*, the most downloaded hosted app. Bookmarks are hosted apps that Chrome creates when the user selects "Add to desktop..." option. They can only have any combination of *background, clipboardRead, clipboardWrite, geolocation, notifications, unlimitedStorage* permissions [20]. The last permission, i.e., *unlimitedStorage*, needs to be declared in the manifest file of the extensions if they need an unlimited quota for storing information. It complements the *storage* which is limited by default to 5MB. Therefore, there is no API call to the *unlimitedStorage* permission and it should be seen as an extra option for the *storage* permission.

Listing 1 Manifest of the most downloaded hosted app.

```
{"app": {
   "launch": {
     "web_url": "http://www.wevideo.com/
         drive"
   }
 },
 "default_locale": "en",
 "icons": {
   "128": "wevideo_128.png"
 },
 "manifest_version": 2,
 "short_name": "WeVideo",
 "version": "4.4.0",
 ...
}
```

Packaged Apps. There are two families, *legacy* (packaged apps version (1) and *platform* (packaged apps version (2) apps. Legacy apps look like a windowed wrapper around a website and have the power of extension APIs. They are identified by adding the `app.launch.local_path` key to the manifest that identifies the resource that is loaded when the app is opened (see Listing 2 for an example of the manifest file of "Google Drawings" (whose id is *mkaakpdehdafacodkgkpghoibnmamcme*), the most downloaded packaged app v1 in the Web Store). On the other hand, platform apps are standalone applications that mostly run independently of the browser, being able to run in the background with no window interface. They are identified by adding the `app.background` key in the manifest, providing the script that is executed when the app is launched (an example of the manifest file of "Postman" packaged app v2 whose id is *fhbjgbiflinjbdggehcddcbncdddomop* can be seen in Listing 3).

Listing 2 Manifest of the most downloaded legacy app.

```
"default_locale": "en_US",
"manifest_version": 2,
"icons": {
    "16": "icon_16.png",
    "128": "icon_128.png"
},
"offline_enabled": true,
"app": {
    "launch": {
    "local_path": "main.html"
    }
}
```

Listing 3 Manifest of the most downloaded platform app.

```
"app": {
    "background": {
        "scripts": ["background.js"]
    }
},
"externally_connectable": {
    "ids": ["*"],
    "matches": ["*://*.getpostman.com/*"]
},
"permissions": [
    "webview", "system.display",
    "http://*/*", "https://*/*",
    "contextMenus", "unlimitedStorage",
    "storage", "fileSystem",
    "fileSystem.write", "notifications",
    "identity",
    {"socket": [
            "tcp-connect:*:*",
            "tcp-listen:*:*"
        ]}
],
"manifest_version": 2,
```

2.2 Browser extensions

Browser extensions are basically composed of *content scripts* and *background pages*. Content scripts are JavaScript files executed in the webpage context, i.e., they have access to the DOM. Background pages do not have direct access to the DOM but to a powerful API the browser offers. The way browser extensions are identified is by checking that none of the following keys appear on the manifest file: `app`, `export`, `theme`.

Themes. Themes are special browser extensions whose purpose is to change the appearance of the browser. The way they are identified is by including the `theme` key in the manifest. The main difference for browser extensions is that themes do not have HTML nor JavaScript files in the `.crx` package.

3 Permissions vs API calls

Chrome splits extensions APIs into *restricted* and *public* APIs. Public APIs are available to any extension, whereas, to use restricted APIs, they need to declare some permissions in the manifest. Chrome also differentiates permissions by the type of extensions that can use them. For instance, only browser extensions can use the *downloads* permission while only platform apps can use the *mdns*[1] permission.

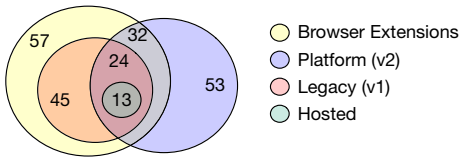


Fig. 2 Permissions according to the type of extensions

In this section, we explain how permissions are mapped to JavaScript APIs. For the experiments, we present in this section, we used Chrome 79.0.3945.130.

3.1 Permissions

Source Code. Permissions are defined in four files having 224, 99, 68, and 8 permissions, respectively. The first file (`api_permissions.h`) contains all the permission that Chrome has, i.e., all the permissions without distinguishing if they belong to extensions, websites, or internal permission management. The second file (`chrome_api_permissions.cc`) defines all the permissions concerning the browser management, i.e., permissions that JavaScript files coming from web servers or content scripts injected by extensions can define and execute. The third file (`extensions_api_permissions.cc`) defines the set of permissions that extensions can use. Finally, the `chrome_apps_api_permissions.cc` includes the permissions that apps can define in the manifest.

Official documentation. Google provides a large amount of documentation[2] about extensions such as source code examples, permissions, APIs, discussions, definitions, videos, and many other resources to make extensions developers' life more pleasant.

We collected all the permission we found online and in the source code, and included them all, in alphabetical order, in the manifest file of (i) a browser extension; (ii) a packaged v1 and v2 apps, and; (iii) a hosted app. As a result, browser extensions can use a total of 57 permissions, packaged v1 45, packaged v2 53 whereas hosted apps only 14 permissions (see Table 1). In Fig. 2, we can see that permissions of packaged v1 apps are a subset of the browser extensions, the number of common permissions between v2 and v1 packaged apps is 24 and between packaged v2 and browser extensions is 32. Hosted permissions are at the intersection of all the extensions with 14 permissions.

[1] mdns is an API to discover services over mDNS, comprising a subset of the features of the NSD spec: http://www.w3.org/TR/discovery-api/.

[2] https://developer.chrome.com.

Table 1 Extensions' permissions

Extensions	Permissions
Common	*AccessibilityFeatures, alarms, app.window, background, bookmarkManagerPrivate, clipboardRead, clipboardWrite, commands, contextMenus, downloadsInternal, fileBrowserHandlerInternal, geolocation, idle, nativeMessaging, newTabPageOverride, notifications, power, storage, system.cpu, system.display, system.memory, system.storage, tts, unlimitedStorage*
Browser extensions	*AccessibilityFeatures.modify, accessibilityFeatures.read, activeTab, bookmarks, browsingData, contentSettings, cookies, debugger, declarativeContent, desktopCapture, desktopCapturePrivate, downloads, downloads.open, downloads.shelf, fontSettings, gcm, history, identity, identity.email, management, pageCapture, printerProvider, privacy, proxy, sessions, tabCapture, tabs, topSites, ttsEngine, webNavigation, webRequest, webRequestBlocking, windows* ∪ Common
Legacy (v1)	*ActiveTab, bookmarks, browsingData, contentSettings, cookies, debugger, fontSettings, history, management, pageCapture, privacy, proxy, sessions, tabCapture, tabs, topSites, ttsEngine, webNavigation, webRequest, webRequestBlocking, windows* ∪ Common
Platform (v2)	*AccessibilityFeatures.modify, gcm, hid, identity, accessibilityFeatures.read, usb, app.window.alwaysOnTop, app.window.fullscreen,app.window.fullscreen.overrideEsc, app.window.shape, appview, audioCapture, browser, desktopCapture, desktopCapturePrivate, fileSystem, fileSystem.directory, fileSystem.retainEntries, fileSystem.write, identity.email, mdns, mediaGalleries, pointerLock, printerProvider, serial, syncFileSystem, system.network, videoCapture, webview* ∪ Common
Hosted Apps	*AccessibilityFeatures, app.window, background, bookmarkManagerPrivate, clipboardRead, clipboardWrite, commands, fileBrowserHandlerInternal, geolocation, downloadsInternal, newTabPageOverride, notifications, storage, unlimitedStorage*

3.2 API calls

Chrome exposes a set of APIs to extensions. To extract all the APIs and perform a mapping between permissions and JavaScript API calls, we parsed all the online documentation and extracted all the methods, properties, and events for each permission. We implemented a script that automatically generates the combination of all the permissions listed in Table 1 and systematically creates an extension with such combinations in the `permissions` key of the manifest. After that, the script automatically installs the extension with all the methods, events, and properties of the APIs and checks which ones are executable given the permissions defined.

Some conclusions arisen from this mapping are:

Public APIs. There are permissions like *app.runtime, app.window, events, extension, permissions, i18n, extensionTypes, types, windows* that are not needed because the APIs are public;

Manifest Keys. APIs like `automation`, `ommibox`, `bluetooth`, `devtools.*`, `commands`, `bluetoothSocket`, and `sockets.*` do not have to be included in the permissions but as keys in the manifest [29].

Dependency Permissions. Some APIs require extra permissions apart from the ones needed to have access to such API, e.g., to block HTTP requests, an extension needs *webRequestBlocking* apart from *webRequest*. Another example is *unlimitedStorage* permission, which increases the size of the local storage but depends on the *storage* one.

Public Elements. There are methods, properties, and events of restricted APIs that are public. As an example, `uninstallSelf()` and `getSelf()` functions of the management API are public.

No API. There are permissions without API. For instance, the *background* permission makes Chrome run when the user logs into the computer as well as keeping Chrome running until the user explicitly quits it.

Host Permissions. By including a host permission, i.e., the URL of a particular web page, domain, or the *<all_url>* wildcard, it automatically allows the extension to execute the APIs granted by *cookies, declarativeNetRequest, declarativeWebRequest, webRequest* permissions. Note that the information the extension will retrieve is concerning the host(s) defined in the `permissions` key. Also, there are some APIs that can be run by including different permissions, for instance, `chrome.webRequest` API, can be run by defining the *webRequest* permission but also the *activeTab* one.

Special Permissions. There are permissions like *webview* that allow extensions to use the `<webview>...</webview>` HTML tag instead of granting access to a JavaScript API.

4 Methodology

In this section, we explain in detail the methodology we used to analyze extensions. We do not attempt to detect whether they are malicious or not, but rather to demonstrate to what

extent extensions are compliant with the SoLP and provide empirical evidence of it.

We split our analysis into three main tasks: Web Store Scrapping, Mapping API-Permissions, and Static Analysis (see Fig. 3). In the following, we explain in more detail each one of these tasks.

Web Store Scraping As of December 2019, we crawled the Web Store, Google's official repository where extensions are stored and distributed. We downloaded a dataset of 170,927 extensions, composed of 136,018 browser extensions, 17,566 apps and 17,343 themes (see Table 2). Regarding apps, 1221 are legacy, 6778 are platform, and 9567 are hosted. For browser extensions, we show in Table 3 the category that they belong to. Finally, even though we detected 26,879 themes stored in the Web Store, we could download only those that are free for the user, i.e., 17,343.

API calls vs Permissions We split the analysis into apps, extensions, and themes. Despite them being stored and managed similarly by the Web Store, we individually analyze them because the way Chrome handles them is different and so are the permissions—as we explained in Sect. 3. In this work, we consider those permissions are part of either the `permission` or `optional_permissions` keys of the manifest, i.e., the ones listed in Table 1 for every type of extension. Therefore, APIs granted by *automation, bluetooth, bluetoothLowEnergy, bluetoothSocket, browserAction, commands, devtools.*, ommibox, pageAction, sockets.** keys are not part of this study.

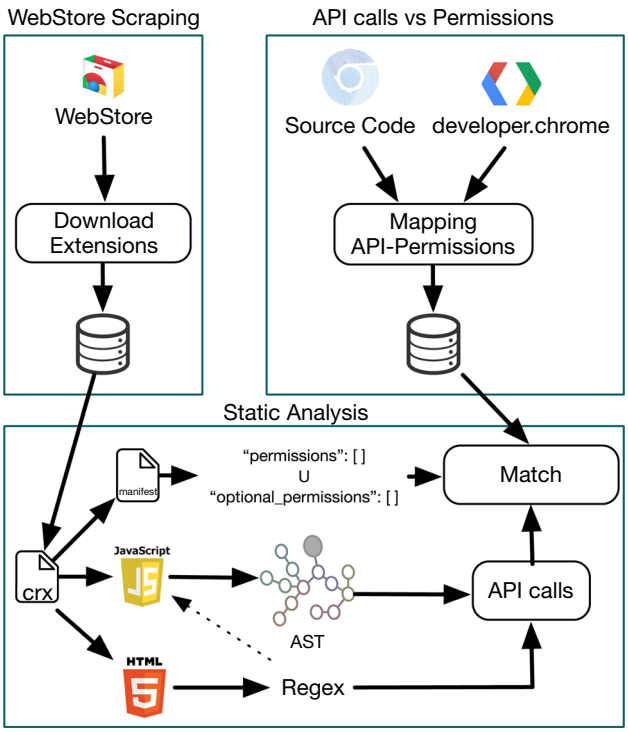


Fig. 3 General overview of our analysis

Table 2 Dataset downloaded in December 2019

Type		#Extensions
Apps	Hosted	9567
	Legacy (v1)	1221
	Platform (v2)	6778
Browser extensions	Browser extensions	136,018
	Theme	17,343
Total		170,927

Table 3 Categories of the Web Store that the Browser Extensions of the dataset belong to

Category	#Extensions
Productivity	36,679
Fun	25,981
Photos	21,971
Web development	11,311
Communication	10,758
Accessibility	9568
Search tools	8061
Shopping	5690
News	2566
Blogging	1769
Sports	1664
Total	136,018

Static Analysis For the analysis of the extensions, we extract not only the permissions they define in their manifest files but also all the API calls they implement in either HTML or JavaScript files. With this, we can check whether the API and therefore the permissions might be used by extensions or not at installation time.

To get the permissions of the extensions, we first parse both the `permissions` and the `optional_permissions` keys of the manifest. With this, we got a list of tentative APIs the extensions might use. In parallel, we statically analyze all the HTML and JavaScript files of the extensions to extract the API calls they define.

To analyze the JavaScript files, we generate the *Abstract Syntax Tree (AST)* of all the scripts by using Esprima[3]. This allows us to detect all the `CallExpression` statements and extract the privileged API the extensions include in the source code. Note that this parser can also be done with classical regular expressions, however, with ASTs we

[3] https://esprima.org.

can model more complex scenarios where, for example, extensions define `chrome` as a variable before using any of the available APIs, e.g., `var aux = chrome; var aux2 = cookies; var cc = aux.aux2.getAll();`. By reconstructing such a `CallExpression` in the AST, we easily get that the `CallExpression` is indeed `var cc = chrome.cookies.getAll()`.

For the HTML files, we used regular expressions to detect API calls, inline JavaScript, and keywords like `<webview>`. Figure 3 depicts a general overview of the process we followed to analyze the extensions.

As we discuss later in Sect. 8, our analysis does not consider complex forms of obfuscation, external code execution, or extensions that use code generation functions like `eval()` and `setTimeout()`.

5 Overprivileged extensions

Previously, we performed a mapping between permissions and APIs. Next, we analyze the extensions and detect those which are potentially overprivileged. In what follows, we provide a more precise definition of *Well-privileged Extensions*.

Definition 1 Well-privileged extensions are extensions with no more, no less permissions than those needed to fulfill the functionality that they were implemented for. If they require more permissions, they are overprivileged. If they require less, they are underprivileged.

To identify the functionality of the extensions we might need: 1) a formal specification of the functionality, and; 2) a formal justification of why the required permissions are needed to fulfill the functionality. However, checking this is a complex problem that is in general undecidable (being essentially a program verification problem).

Given these constraints, we provide a working definition similar to Wang et al. [40], that allows us to identify extensions compliant with the SoLP in a lightweight manner. We define overprivileged extensions and extensions compliant with the SoLP as follows.

Definition 2 An extension is *overprivileged* when there is at least one permission in its manifest file whose API is not defined in the source code of the extension at installation time. We say that an extension is compliant with the SoLP when is not overprivileged.

Data Preprocessing There are some direct actions we take before we analyze any type of extension. In particular, we directly mark extensions as:

Overprivileged: those with permissions that depend on others to be used and they are not included, e.g., extensions that include *unlimitedStorage* but do not define the *storage* one. Other examples are *downloads.open, downloads.shelf*;
SoLP: compliant with the Spirit of Least Privilege those that do not define permissions, i.e., `permission` $\cup$ `optional_permissions` $= \emptyset$, or they do so but only include URLs (or any host pattern matching).

After analyzing all the extensions of our dataset, we conclude that over 40% of them are compliant with the SoLP, or in other words, there are over 60% of overprivileged extensions (see Table 4).

5.1 Apps

In total, we got that 45.3% of the analyzed apps might be overprivileged, thus having 54.7% compliant with the SoLP. In more detail, there are 7962 possible overprivileged apps where 2885 correspond to hosted and 5342 to packaged apps. In more detail:

Hosted Apps. We extracted all permissions that hosted apps have in their manifest files, run the data preprocessing procedure explained earlier in this section, and analyzed the remaining apps, concluding that 69.9% of hosted apps are compliant with the SoLP while 2885 are potentially overprivileged hosted app downloaded over 60M times.

Legacy Apps. After preprocessing the extensions, we got 704 apps with at least one permission. We analyzed the remaining apps and concluded that there are 640 legacy apps, downloaded over 3M times, that might be overprivileged.

Platform Apps. We analyzed 6778 extensions and concluded that there are 2341 platform apps compliant with the SoLP which approximately represent 34.5% of all the analyzed packaged v2 extensions. In other words, there are 4437 potentially overprivileged platforms being downloaded over 69M times.

5.2 Browser extensions

After the data preprocessing, we got that 80,470 extensions might be overprivileged (59.2%) and therefore 55,548 are compliant with the SoLP (40.8%). These potentially overprivileged extensions have been downloaded over 900M times.

We grouped browser extensions according to the categories they belong to in the Web Store and show in Fig. 4 how they are distributed. It is interesting to see how the number of potentially overprivileged browser extensions per category is larger than those compliant with the SoLP in all the categories but blogging, fun, and sports. It is also noticeable the number of potentially overprivileged extensions in productivity and photos categories, being these extensions one of the most

Table 4 Summary of overprivileged extensions. SoLP stands for Spirit of Least Privilege compliant extensions

Type		#Extensions	#SoLP
Apps	Hosted	9567	6,682 (69.9%)
	Legacy (v1)	1221	581 (47.5%)
	Platform (v2)	6778	2,341 (34.5%)
Browser extensions	Browser extensions	136,018	55,548 (40.8%)
	Theme	17,343	17,338 (99.9%)
Total		170,927	82,490 (48.3%)

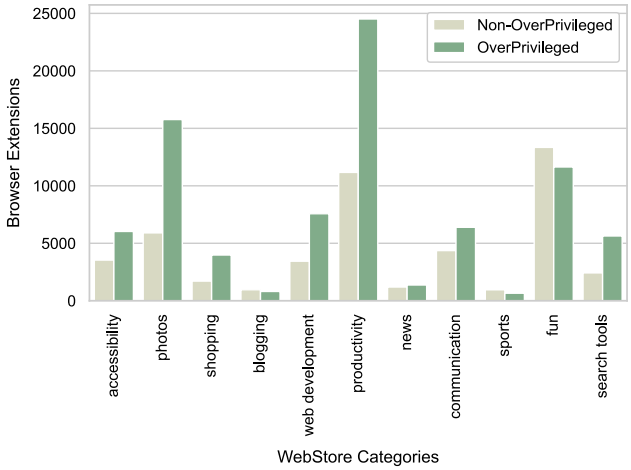


Fig. 4 Potentially overprivileged browser extensions classified into Web Store categories

downloaded ones—having approximately 488M and 107M of downloads, respectively.

Overprivileged browser extensions usually ask for more harmful permissions, i.e., permissions that pose privacy issues such as *cookies*, *bookmarks*, and *topSites*, get control over the extensions the user has (*management*) and intercept and modify web requests *webRequest, webNavigation, webRequestBlocking*). However, there is one special case: *tabs* permission. Even though we do not have evidence, we think that *tabs* permission is confusing for developers. The reason is because *tabs* is only needed to get the "url," the "pendingUrl," the "title," and the "favIconUrl" properties of a tab [15]. Yet, there are more than 80 methods and events under this API that do not require the permission to be used.

The number of extensions compliant with the SoLP that define and use network permissions (e.g., *webRequest, webRequestBlocking, webNavigation*) is so low in comparison to the potentially overprivileged extensions and the consequences might be catastrophic. Extensions can alter, block, or replicate and send any ongoing communication, including the *Content Security Policy (CSPs)*, without the user being notified.

We analyzed the three most defined permissions of potentially overprivileged extensions according to the category they belong to in the Web Store and compared the results with the extensions compliant with the SoLP.

In summary:

cookies permission is present in the top 10 of the most used permissions of the potentially overprivileged extensions in all the categories but "fun." There are 18,991 browser extensions that define such a permission but 12,174 do not use it.
management is a common permission among potentially overprivileged extensions in "Sport," "Web development," "Productivity," "Blogging" and "Communication" categories. Such a permission allows extensions to enable, uninstall and configure other extensions the user has. There are 3139 potentially overprivileged extensions that define such a permission and do not use it.
webRequest is popular among browser extensions of all categories (15,602), but 10,667 do not use it.

Themes. Despite themes should not have permissions, we found out that 4 include them in their manifests. In Table 5, we show the information we got from them. In particular, the permissions are *tabs, bookmarks, http://*/*, https://*/*, activeTab*, http://api.flickr.com/. To check whether the themes are overprivileged, we manually deleted the permissions from their manifest and demonstrated that the permissions were indeed not needed and themes still changed the default look & feel of the browser.

Permissions Evolution. We downloaded a new dataset of browser extensions 6 months later and repeated the same experiments; however, we could not find any new findings (see Table 6 in Sect. 1).

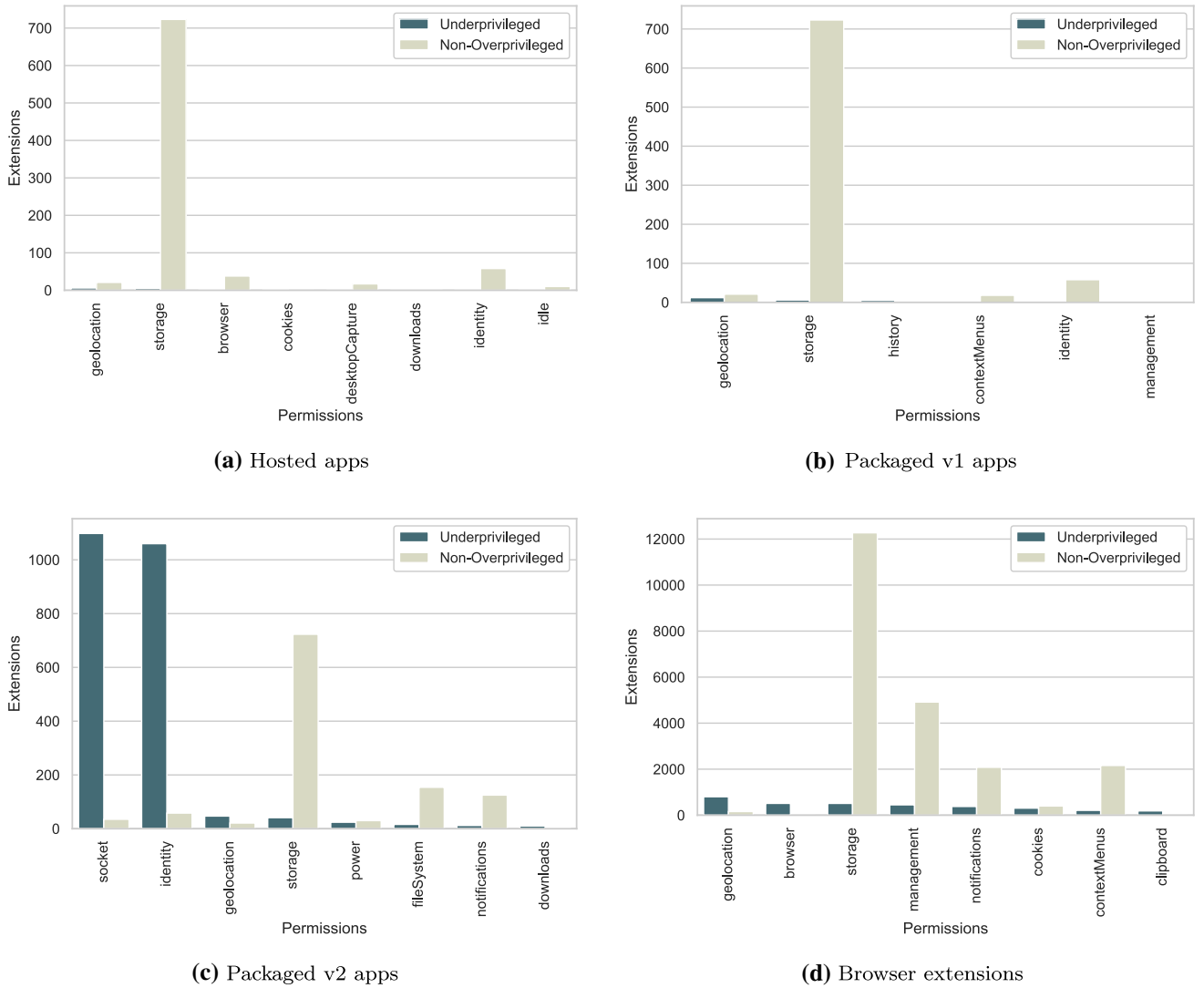
6 Underprivileged extensions

In this section, we study how many extensions have API calls in their files by which they do not have enough permissions to execute them, i.e., how many extensions are underprivileged. Chrome governs the access of browser extensions to privileged APIs. However, extensions can take advantage of how Chrome implements the privileged architecture to get private information without defining the corresponding permissions.

Here, we have to differentiate between background pages and content scripts. When background pages call any privi-

Table 5 Themes stored in the Web Store. The third column (type of files) has the number of files of each type in parenthesis

Name	Extension ID	#Files	Type of files	Permissions
Krishna with Radha	anglkkfahcceepncmdbjngjcmkagojhb	14	json(2), jpg(2), png(8), html(1), pak(1)	*activeTab*
Kinorul Theme	fodejocmlfpendnjfbpamndoklajikhk	13	json(2), jpg(1), png(9), html(1)	*bookmarks, tabs, http://*/*, https://*/**
Premium-Black Dark Theme	denonjmlifkgigajfggbbagckemhmphi	9	json(6), jpg(1), png(2)	*identity*, https://www.googleapis.com/
My First Extension	ddmjeblijdbbnjecbdclkldbccdjhcia	7	json(1), png(1), html(3), js(1), cc(1)	*tabs*, http://api.flickr.com/
Enrique Iglesias Theme	cphepdgiboheikghhnjfhaemoilfjklp	6	json(2), jpg(1), png(2), html(1)	*tabs*



(a) Hosted apps

(b) Packaged v1 apps

(c) Packaged v2 apps

(d) Browser extensions

Fig. 5 Underprivileged vs extensions compliant with the spirit of least privilege

leged API without the corresponding permission, the browser simply blocks such attempts and returns an error instead. However, when these privileged APIs are accessed by content scripts, extensions can inherit the permissions that users might have previously granted to web pages and thus, retrieve sensitive information without having to declare such permissions at installation time.

One such example is the `geolocation` API. Extensions that need to retrieve the user's geolocation through the API the browser offers (e.g., `navigator.geolocation.getCurrentPosition(geoSuccess);`) would have to include such permission in the manifest file, otherwise the browser blocks the API request and generates a JavaScript error. However, if Alice has specifically granted a web page to get her geolocation, and the extension runs on that page, then it can call the API and have access to Alice's geolocation bypassing then the permissions ecosystem.

More precisely:

Definition 3 We say an extension is *underprivileged* when there is at least an API call by which the extension does not have enough privileges to execute it.

In more detail, for hosted apps we got that 18 out of 9567 are underprivileged whereas for packaged apps we found that 22 out of 1221 v1 (Legacy apps) and 1269 out of 6778 v2 (Platform apps) are so. Regarding browser extensions, we detected 13,217 trying to execute restricted API without the required permissions.

We extracted the most used API calls and generated plots for hosted and packaged apps v1 (see Fig. 5a, b, respectively). In general, we observed that the number of extensions compliant with the SoLP and non-underprivileged extensions is always larger than the underprivileged ones. From the security point of view, this should always be the case because it does not pose any extra security issue (note that extension compliant with the SoLP might of course be malicious).

However, this is not the case for platform apps (packaged v2) and browser extensions (see Fig. 5c, d, respectively). Contrarily to hosted and legacy apps, the number of platform apps that try to use *socket, identity* permissions and the number of browser extensions that try to access the APIs granted by *geolocation, browser* permissions is larger than the legitimate extensions. The amount of extensions is significant enough to be considered a development error.

The number of packaged v2 apps that try to use both *socket* and *identity* permissions is almost the same, having a common file that includes calls to `chrome.identity.getAuthToken()` and the `read()`, `create()` and `connect()` methods of the `chrome.socket` API.

From the experiments, we conclude that almost 82% of underprivileged platform apps attempt to send and receive data over the network using TCP and UDP connections without consent (*socket* permission). This permission is not common in apps compliant with the SoLP (around 0.5% of them define and use it). Also, we observe a similar pattern with *identity* permission, where 80% of underprivileged platform apps try to get OAuth2 access tokens using such a permission.

Regarding browser extensions, we observe that around 21% of the underprivileged ones, attempt to geolocate the user (*geolocation* permission) whereas just 0.6% of those compliant with the SoLP define and use that permission correctly. This confirms that extensions try to bypass the permission ecosystem. Similarly, almost 5% of the underprivileged browser extensions attempt to access data stored in the clipboard.

We notice that over 13% of the browser extensions try to open new tabs using the (chrome.browser.openTab()) method—allowed by defining the *browser* permission. Such a method used to be the way apps can open new tabs and it is deprecated for browser extensions. For this reason, we think that when developers migrated their apps to browser extensions, they forgot to clean up and update the APIs consequently.

Finally, we extracted how many browser extensions are potentially overprivileged and underprivileged at the same time, obtaining 11,563 in total.

7 Tools for users, developers, and vendors

To disseminate our findings, we release different solutions depending on the target users[4]. That is, we differentiate between non-technical users, browser extension developers, and browser vendors.

For regular users, we provide a browser extension that they can freely install in the browser. This extension analyzes the extensions of the user by using the *management* permission and retrieving the IDs of the already installed extensions. Also, by using the same permission, when a new extension is about to be installed, our extension can warn the user about the possible risks and information the extension may have access to. If the user agrees, the new installation will take place otherwise it will block the new extension to be installed.

Given the number of potentially overprivileged extensions, we set up a web page with detailed information about the analysis we carried out in this paper. Concretely, we included all the analyzed extensions and show the reasons why they are marked either as potentially overprivileged, underprivileged, or both. We also included extensions that are compliant with the spirit of least privilege.

Finally, we make all the scripts we used for the analysis public. The scripts can be part of the browser extensions development process so the developers can run the scripts

[4] https://github.com/Pica4x6/SoLP.

before releasing their extensions in order to be compliant with the spirit of least privileges and if not, how to achieve so. Also, they can easily be included as part of the official repositories and, show messages to the users who are about to install a new extension—similar to the information that is currently shown by Google like the rating, number of downloads, or permissions that the extension requires to be installed.

8 Threats to validity

In this section, we present a systematic approach to validate our results based on previous work [2,33]. In particular, we focused on three main threats: 1) the replication of the results so that future researchers can reproduce the results presented in this paper; 2)3) generalizability, and; 4) the *external validity* of the results. The external validity means whether the results can be generalized as well as it analyzes if the findings of the study are of relevance for others. According to the original authors, in the case of quantitative research (experiments), this primarily relies on the chosen sample size [33].

Replicability In this paper, we study Chrome's permission ecosystem by performing an in-depth study about how extensions define and use permissions. To obtain how many extensions are compliant with the spirit of least privilege, we relied on Esprima, a third-party library to extract the AST of the JavaScript files and used regular expressions to statically extract API calls where it was not possible, i.e., HTML files.

Note that extensions developers can use coding patterns, or techniques like external source code execution, classes emulation and closures [25], and minimization to hide whether an extension is trying to access a particular API. In addition, extensions that try to obfuscate their behavior by code generation functions like `eval()`, `Function()`, `setTimeout()` and `document.write()`.

Regarding the dynamic code execution, we extracted the CSP that extensions need to define in the manifest file in case they want to use the `eval()` function to give an upper limit of extensions that need further analysis. We got that less than 33k browser extensions define it (24% of the browser extensions). We are working on a prototype that allows us to understand under which circumstances (e.g., triggers, user actions, web pages, day time, location, etc.) the extensions access the APIs.

Instead of relying on the official documentation, we obtained a list of possible permissions defined in Chromium's source code and automatically generated extensions with a combination of all the permissions we found. After that, we dynamically installed the extensions in the browser and checked whether they could actually use the permissions the documentation states. While most of the permissions match with the official documentation, we found some slight differences. For instance, according to the official documentation, the permissions hosted apps can define are: *background, clipboardRead, clipboardWrite, geolocation, notifications, unlimitedStorage, unlimited_storage*[5] or a combination of them [20]. However, we empirically observed that the permissions that hosted apps can use is larger (14 permissions in total) than it is online stated (6).

Extensions that are marked as compliant with the spirit of least privileged do not have false positives, meaning that they have one line of code that access a privileged API governed by a permission defined in the manifest file at installation time. On the other hand, as we rely on static analysis, we may thus have false negatives, subject to the precision of the analysis when detecting overprivileged extensions. This limitation is shared with the related work that also leverages static analysis (e.g., [4,6,21,34]). Therefore, some of the extensions marked as potentially overprivileged would need further analysis to claim whether they are indeed overprivileged or not.

Generability Given the popularity of Chrome, the extension ecosystem that this browser implements has become a *de facto standard*. However, we realized that every browser implements a different set of APIs and even different methods. In [7], most of the available APIs that the main browsers implement (Chrome, Edge, Opera, and Safari) are described and compared. Therefore, even though the theoretical approach can be used in other ecosystems, the results presented in this paper cannot be generalized to the extensions of other browsers, not even to those based on Chrome like Edge and Opera.

External Validity Manually analyzing all the extensions to validate our process is infeasible given the amount of data. Instead, we randomly chose a subset of extensions and manually analyzed them. To determine the sample size, we followed the strategy presented by Israel [23]. In our case we have a size population of 136,018 extensions so, according to Israel, we need to manually analyze 662 extensions. In this way, we meet 95% accuracy, 99% of the time, meaning that our methodology marks extensions correctly according to the permissions they define in the manifest file at installation time.

The manual analysis of the extensions consisted in certifying that the output of both Esprima and regex matched with the scripts the extensions include as well as checking whether the APIs defined in the source code correspond to the permissions defined in the manifest.

[5] unlimited_storage is an Alias defined in the extensions/common/permissions/extensions_api_permissions.cc file for UnlimitedStorage permission.

9 Discussion and limitations

Shared Modules. Despite we demonstrated that shared modules are not a popular option among extensions developers, we see shared modules as a promising option to avoid duplicated resources. We understand that developers cannot be forced to use them but Google might perform a static analysis by checking the Hash of all the resources included in the `.crx` package of the extension and 1) delete from the package and add it to the "export" key of the manifest of the extension. From the security point of view, this might also be a huge improvement since it is well known that files like jQuery are usually altered to include malware [27]. To solve this problem, Google might statically analyze the extensions looking for the external JavaScript libraries (e.g., jQuery, react, d3.js, etc.), include a mechanism like *Subresource Integrity (SRI)* to verify that these external resources have not been altered, and automatically add them as shared modules so that all the extensions use the same file. Apart from that, we argue that by using Shared Modules for common resources, Google might implement a deduplication technique allowing it to save space and optimize its resources [31,41].

Updates. We confirmed what the official documentation says about updates and permissions. In particular, when extensions are about to be updated, Chrome detects whether the extension requires more permissions than before and if so, a new alert is warned to the user. However, if the extension is already overprivileged but it does not have API calls in the source code to use such APIs, but the update contains the API calls to use such permissions the user will not be aware of it [19]. This might be one of the reasons to give more permissions to the extensions than needed in advance. There are some actions that both Google and extensions developers should adopt to avoid this problem and make extensions more privacy and security compliant. Essentially, everything revolves around the principle of minimum permissions so extension developers should only ask for permissions when they need them. Our scripts will be released so developers can use them to be compliant with the spirit of least privilege. On the other hand, our experiments might be included as part of the acceptance process that Google performs to decide whether an extension is or is not allowed to be stored in the Web Store. By doing so, Google can easily detect whether an extension is overprivileged, reject the extension and give feedback to the developer about how to avoid such an issue. Since our experiments rely on static analysis of the extensions' source code, we measured the time needed to analyze one extension which, on average, was of 2 s on a MacBook Air.

Well-Formed Hosted Apps. Even though the purpose of this paper is the analysis of the permission ecosystem for extensions in Chrome, we performed an additional experiment to check how many hosted apps fulfill the structural constraints. According to the official documentation, hosted apps "*must contain an icon and a manifest that has details about how the app should function. Note: Unlike extensions and packaged apps, a hosted app has no access to the files inside its .crx file*." We conclude that there are only 56 of the 9567 which are well-formed according to the official guidelines. In other words, 9511 hosted apps are not well-formed and contain more files than they should have. We leave for future work the question of how these files interact with Chrome and whether they may affect the security and privacy of the user. As an example, there is a hosted app[6] that has 18 files with a total size of 500 Mb.

Manifest V3. Google introduced in 2020 a new version of the manifest file of the extensions; however, it was not until January 2021 that Google started accepting extensions with this new version in the Web Store. In this new version, developers will have to define in advance under which circumstances the extensions will retrieve the (sensitive) information that some permissions govern, the so-called rules [17]. This is a promising step toward security since extensions might be statically analyzed in advance by both, Google and users. Even though we strongly think this is a step toward more secure and private browser extensions, we do not think that it will solve the problem. The problem of whether the extensions need a particular set of permissions to work will remain, being still possible for an extension to get the history and share it with other parties as part of its functionality despite its goal is just changing the color of the background. To evaluate how this new version affects our analysis, we downloaded a fresh dataset as of March 2021 and we did not find any extension using this new version of the manifest file.

10 Related work

Permissions in extensions have been traditionally overlooked. Most researchers extracted the most common permissions of previously detected malicious extensions [1,13,22,24,26,28] and concluded that they are overprivileged. However, as far as we know, this is the first in-depth study where we (1) analyze the permissions ecosystem in Chrome; (2) download and study more than 150 k Chrome extensions; (3) automatically and empirically demonstrate how overprivileged the extensions are, and; (4) publicly release our scripts to help users, developers, and Google to get feedback about how extensions define and use permissions and guidelines to be compliant with the principle of least privilege [5,32].

One of the first authors who analyzed the permissions ecosystem on both Chrome and Android was Felt et al. [12,13]. They were not focused on overprivileged extensions

[6] ghgffeelgghaomkheipfdfakkedjjkok.

or Android apps but on whether the permissions are effective at protecting users. Regarding extensions, the authors used a dataset of 1000 extensions concluding that 14.7% are overprivileged. In our work, we not only used a dataset of more than 150 k extensions but also concluded that the number of extensions compliant with the spirit of least privilege is 48.3%. Our paper, similar to Felt's works, relies on static analysis to analyze the permissions of the extensions.

The main differences between our work and the reviewed literature are that authors usually propose a new set of policies, rules or languages to detect, mitigate or avoid security issues. For instance, malware detection [1,9,24,26,30,37], fingerprinting [35,36,39] and advertising [3,38,42] are three of the most common research topics in extensions. Contrarily to these works, we focused on: final users, extensions developers and extension official repositories. For users and extensions developers, we released two solutions: a browser extension and a web page where they can check whether an extension is or is not compliant with the spirit of least privilege, underprivileged, and some hints about how to be compliant with that principle. Our scripts can also be included as part of the extensions vendor's acceptance criteria.

Guha et al. [22] proposed a safe JavaScript language to increase the security of extensions and access to the API. Complementing this work, we first parsed the entire Web Store and analyzed approximately 80% of the extensions from the spirit of least privilege point of view. After such analysis, other solutions can be implemented on the user's side like a mechanism that detects whether the extension is overprivileged and automatically removes those permissions which are not used at installation time.

Liu et al. [28] discussed the security model for extensions in Chrome and showed how to perform large-scale bot attacks, being the main cause of the coarse-grained privilege management and the access to DOM elements. Ten years later, we corroborated that extensions still suffer from the same coarse-grained issues.

Aggarwal et al. [1] detected 218 spying browser extensions that access private information and, after obfuscating it, is sent to external servers. Authors analyzed the permissions of such extensions and, despite the number of extensions not being significant enough, the permissions of the spying extensions (*tabs, cookies, storage, <all_urls>, history, geolocation, activeTab*) match with our analysis of overprivileged extensions.

Recently, some private security researchers released a tool that automatically generates comprehensible security reports [11]. They are motivated by the fact that extensions functionality change over time without the users being notified, e.g., "*a malicious third party were to gain control of the extension, perhaps by buying it from the developer or compromising the developer's account. The third party could add malicious code and push the new version out to existing users without triggering another security review*." Under these circumstances, researchers developed a tool for security teams of organizations that automatically parses the `.crx` files of the extensions generating automatic reports.

11 Conclusions

In this paper, we analyzed the permissions of Chrome extensions and determined whether they are overprivileged, underprivileged, or both. We downloaded 170,927 from the Web Store and got not only the permissions the extensions define in the manifest files but also do we extracted the API JavaScript calls that extensions have in the source code at installation time. We concluded that 48.3% of the scrutinized extensions are compliant with the spirit of least privilege whereas 8.5% are underprivileged. Last but not least, we developed (1) a browser extension to aware regular users of the extensions they have installed; (2) an app where browser extensions developers can check whether their extensions are compliant with the spirit of least privileged, and; (3) a set of scripts that can be part of the vendors' acceptance criteria so when developers upload their extensions, our scripts automatically analyze the extensions and generate a report about the permissions they define.

Funding Open access funding provided by Chalmers University of Technology. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Facebook.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Appendix A

We include in Table 6 a comparison of the top 10^{th} most used permissions of extensions in both datasets. In general, we can see how potentially overprivileged extensions remain proportionally with respect to the previous dataset.

Table 6 Evolution of permissions in browser extensions of two datasets downloaded with a separation of 6 months

#Rank	Dataset 2019		Dataset 2020	
	Permission	#Ext	Permission	#Ext
1st	*tabs*	56,463	*tabs*	56,364
2nd	*activeTab*	25,053	*activeTab*	27,173
3rd	*storage*	20,204	*storage*	19,354
4th	*cookies*	12,174	*cookies*	11,352
5th	*webRequest*	10,667	*webRequest*	10,128
6th	*webRequestBlocking*	8999	*webRequestBlocking*	8141
7th	*notifications*	6817	*notifications*	6054
8th	*contextMenus*	6661	*contextMenus*	5987
9th	*webNavigation*	3229	*management*	2784
10th	*management*	3139	*webNavigation*	2699

References

1. Aggarwal, A., Viswanath, B., Zhang, L., Kumar, S., Shah, A., Kumaraguru, P.: I spy with my little eye: analysis and detection of spying browser extensions. In: Euro S&P, pp. 47–61 (2018)
2. Ampatzoglou, A., Bibi, S., Avgeriou, P., Verbeek, M., Chatzigeorgiou, A.: Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. Inf. Softw. Technol. **106**, 201–230 (2019)
3. Arshad, S., Kharraz, A., Robertson, W.: Identifying extension-based ad injection via fine-grained web content provenance. In: RAID, pp. 415–436 (2016)
4. Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M.: VEX: vetting browser extensions for security vulnerabilities. In: USENIX, pp. 339–354 (2010)
5. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: NDSS (2010)
6. Barua, A., Zulkernine, M., Weldemariam, K.: Protecting web browser extensions from javascript injection attacks. In: ICECCS, pp. 188–197 (2013)
7. Browser support for JavaScript APIs. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Browser_support_for_JavaScript_APIs
8. Carlini, N., Felt, A.P., Wagner, D.: An evaluation of the google chrome extension security architecture. In: USENIX, pp. 97–111 (2012)
9. Chen, Q., Kapravelos, A.: Mystique: uncovering information leakage from browser extensions. In: CCS, pp. 1687–1700 (2018)
10. Moving forward from chrome apps. https://blog.chromium.org/2020/01/moving-forward-from-chrome-apps.html
11. CRXcavator: Democratizing chrome extension security. https://duo.com/blog/crxcavator
12. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: CCS, pp. 627–638 (2011)
13. Felt, A.P., Greenwood, K., Wagner, D.: The effectiveness of application permissions. In: USENIX WebApps, pp. 7–7 (2011)
14. Chrome app and extension permissions. https://support.google.com/chrome/a/answer/7515036?hl=en
15. chrome.tabs. https://developer.chrome.com/extensions/tabs
16. Declare permissions. https://developer.chrome.com/extensions/declare_permissions
17. Declare permissions. https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/
18. Declare permissions and warn users. https://developer.chrome.com/apps/permission_warnings
19. Extensions and apps in the chrome web store. https://developer.chrome.com/webstore/apps_vs_extensions
20. Hosted apps. https://developer.chrome.com/webstore/hosted_apps
21. Guarnieri, S., Livshits, B.: GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code. In: USENIX, pp. 151–168 (2009)
22. Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: S&P, pp. 115–130 (2011)
23. Israel, G.D.: Determining sample size (1992)
24. Jagpal, N., Dingle, E., Gravel, J.P., Mavrommatis, P., Provos, N., Rajab, M.A., Thomas, K.: Trends and lessons from three years fighting malicious extensions. In: USENIX, pp. 579–593 (2015)
25. Javascript closures. https://www.w3schools.com/js/js_function_closures.asp
26. Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., Paxson, V.: Hulk: eliciting malicious behavior in browser extensions. In: USENIX, pp. 641–654 (2014)
27. Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., Vigna, G.: Revolver: an automated approach to the detection of evasive web-based malware. In: USENIX, pp. 637–652 (2013)
28. Liu, L., Zhang, X., Yan, G., Chen, S.: Chrome extensions: threat analysis and countermeasures. In: NDSS (2012)
29. Manifest file format. https://developer.chrome.com/docs/extensions/mv2/manifest/
30. Perrotta, R., Hao, F.: Botnet in the browser: understanding threats caused by malicious browser extensions. IEEE Secur. Privacy **16**(4), 66–81 (2018)
31. Picazo-Sanchez, P., Algehed, M., Sabelfeld, A.: Dedup.js: discovering malicious and vulnerable extensions by detecting duplication. In: ICISSP, pp. 528–535 (2022)
32. Schneider, F.B.: Least privilege and more [computer security]. IEEE Secur. Privacy **1**(5), 55–59 (2003)
33. Siegmund, J., Siegmund, N., Apel, S.: Views on internal and external validity in empirical software engineering. In: ICSE, pp. 9–19 (2015)
34. Somé, D.F.: Empoweb: Empowering web applications with browser extensions. In: S&P, pp. 227–245 (2019)
35. Starov, O., Laperdrix, P., Kapravelos, A., Nikiforakis, N.: Unnecessarily identifiable: quantifying the fingerprintability of browser extensions due to bloat. In: WWW, pp. 3244–3250 (2019)
36. Starov, O., Nikiforakis, N.: Extended tracking powers: measuring the privacy diffusion enabled by browser extensions. In: WWW, pp. 1481–1490 (2017)
37. Starov, O., Nikiforakis, N.: Xhound: Quantifying the fingerprintability of browser extensions. In: S&P, pp. 941–956 (2017)
38. Thomas, K., Bursztein, E., Grier, C., Ho, G., Jagpal, N., Kapravelos, A., Mccoy, D., Nappa, A., Paxson, V., Pearce, P., Provos, N.,

Rajab, M.A.: Ad injection at scale: assessing deceptive advertisement modifications. In: S&P, pp. 151–167 (2015)
39. Trickel, E., Starov, O., Kapravelos, A., Nikiforakis, N., Doupé, A.: Everyone is different: client-side diversification for defending against extension fingerprinting. In: USENIX, pp. 1679–1696 (2019)
40. Wang, H., Liu, Z., Liang, J., Vallina-Rodriguez, N., Guo, Y., Li, L., Tapiador, J., Cao, J., Xu, G.: Beyond google play: a large-scale comparative study of chinese android app markets. In: IMC, pp. 293–307 (2018)
41. Xia, W., Jiang, H., Feng, D., Douglis, F., Shilane, P., Hua, Y., Fu, M., Zhang, Y., Zhou, Y.: A comprehensive study of the past, present, and future of data deduplication. Proc. IEEE **104**(9), 1681–1710 (2016)
42. Xing, X., Meng, W., Lee, B., Weinsberg, U., Sheth, A., Perdisci, R., Lee, W.: Understanding malvertising through ad-injecting browser extensions. In: WWW, pp. 1286–1295 (2015)