Least Privilege for Browser Extensions

Adrienne Porter Felt
University of California, Berkeley
apf@berkeley.edu

Abstract

Browser extensions let developers add extra functionality to the browser. Although this enables popular new features, extensions threaten browser security because they are written by unknown third-party developers. An extension could be directly malicious, or a well-intentioned developer could write buggy code that leaks privileges to a malicious web site operator. This thesis advocates the development of an extension system that limits extensions' privileges to the fewest privileges possible without crippling legitimate functionality.

We motivate the reduction of extension privileges with a study of 25 *Mozilla Firefox extensions. Currently, Firefox extensions have unrestricted access to browser privileges: extensions can delete files from the hard drive and launch processes. Our study shows that* 88% *of the studied extensions do not require the most powerful privileges. We consider how the Firefox extension system could be changed to reduce extension privileges and remove the privilege gap. We then examine the new Google Chrome extension system, which supports restrictions on extensions as recommended by this work. We test the performance of their security mechanisms and study* 25 *popular Google Chrome extensions to see whether they are appropriately privileged.*

1 Introduction

Web browser extensions are phenomenally popular: roughly one third of Firefox users have at least one browser extension [20]. Browser extensions modify the core browser user experience by changing the browser's user interface and interacting with web sites. For example, the Skype browser extension rewrites phone numbers found in web pages into hyperlinks [5]. Although there have been several recent proposals for new web browser architectures [18, 10, 31], little attention has been paid to the architecture of browser extension systems.

Mozilla Firefox introduced the first popular extension system. The Firefox extension system gives extensions complete browser privileges, including full file system and network access. Consequently, malicious and compromised extensions are significant security threats. Malicious extensions have been found in the wild, spying on users and installing persistent malware [33, 13, 26]. Even benign extensions written by well-intentioned developers put users at risk; many extensions interact extensively with arbitrary web pages, creating a large attack surface that attackers can scour for vulnerabilities. Once a vulnerability is found, an attacker can usurp the extension's privileges. At DEFCON 2009, Liverani and Freeman presented attacks against a number of popular Firefox extensions [21].

These attacks raise the question of whether browser extensions require such a high level of privilege. We hypothesize that many Firefox extensions can be satisfied with a significantly reduced set of privileges. To investigate this question, we review 25 popular extensions from the Firefox "recommended" directory. We find that only 3 of the 25 extensions require file system access. The remainder are over-privileged, motivating the design of a new extension platform that limits an extension to a set of privileges chosen at install time.

In a *least privilege* extension system, extension privileges are statically requested by the developer at install time. Static permissions have two advantages: simplifying the extension review process and mitigating vulnerabilities. Mozilla attempts to reduce the malware potential of extensions by reviewing all extensions before listing them in the official extension directory; reviewers could leverage static permissions to automatically accept low-privilege extensions. Extensions that require unusually high-level permissions could receive extra scrutiny. Static permissions would also reduce the severity of extension vulnerability exploits, since an exploit would be limited to the reduced privileges of the extension.

We analyze the Firefox extension API to determine its suitability for least privilege. We find that it does not provide sufficiently fine-grained privileges. For example, many extensions store settings with an interface that can read and write arbitrary files. We identify several influential interfaces that could be altered to improve the API. Additionally, we present an algorithm for finding methods in the Firefox extension API that can lead from a less-privileged interface to a more-privileged interface. These points would need to be monitored or altered so that extensions remain confined to their static permissions.

Google Chrome recently released a new extension system, partially motivated by our work. It is built with security in mind. In particular, extensions are packaged with static permissions. We conduct a study of 25 popular Google Chrome extensions to see whether extensions still exhibit a privilege gap. We find that the Google Chrome extension system is largely successful in reducing the privilege gap, although we identify some potential areas of improvement.

2 Threat Model

A browser extension is a third-party software module that extends the functionality of a web browser, letting users customize their browsing experience. In Section 2.1, we explain how extensions obtain privileges from the browser. We then consider two threat models: malicious extensions and benign-but-buggy extensions. Malicious extension authors try to trick users into installing a fake or altered extension. Benign-but-buggy extensions are well-intentioned but contain vulnerabilities that can be exploited by malicious web site operators or active network attackers. We present these two threat models as they apply to general extension systems and give concrete examples of each for the Firefox extension system.

2.1 Extension Privileges

Firefox extensions execute with the full privileges of the browser. This extension system was originally intended for expert browser developers, but its use has expanded widely. Extensions and internal browser components use the same interfaces (known as XPCOM interfaces). The interfaces include services such as network connectivity, file system access, window managing, and general utility tools like XML parsing. The XPCOM API is implemented in C++ and can be accessed from code written in C++ or JavaScript. Most extensions are written wholly in JavaScript, but some extensions may include compiled C++ components.

Google Chrome extensions are separated into three parts: content scripts, core extensions, and an optional binary. Content scripts run on web pages from specific domains but don't have API access. Core extensions can access extension APIs and make `XMLHttpRequests` from specific domains. The APIs provide access to browser resources like the window and bookmark managers, but file system access is not available via the extension APIs. Extension binaries run with full browser privileges and have local system access. Each of these subcomponents runs in a different process, and they communicate through a data-only message passing channel. An extension's privileges are determined by its developer-defined *manifest file*, which specifies what domains the content script and core extension may access and whether the extension includes a binary. Separating extensions into three isolated components is intended to make it more difficult to obtain the higher level privileges; content scripts, which have the highest attack surface, have the lowest privileges.

We do not consider browser plug-ins in this paper. Although plug-in security is an important area of research [18, 17], browser extensions have different security and functionality needs. Plug-ins render specific media types (such as PDF and Flash) or expose additional APIs to web content (such as the Gears APIs). The authors of the rendered content are different than the authors of the plug-in, and the behavior of the plug-in differs between instances based on the characteristics of the content it is rendering. Plug-ins are requested explicitly by web sites, usually by loading content with a specific MIME type. By way of contrast, extensions interact with web pages without their explicit consent and typically do not execute code that is not bundled with the extension. Furthermore, plug-ins are wholly binaries, and extensions are usually written mostly or entirely in JavaScript.

2.2 Malicious Extensions

Malicious extensions are similar to other types of malware. We consider the case where a malicious extension author tricks the user into installing the extension with false advertising or by altering and redistributing a known extension. In this scenario, the user is making a conscious decision to install the malicious extension. We do not consider extensions that are installed as the payload of another piece of malware [12], through a browser vulnerability [23], or via a clickjacking attack. Users who only download extensions from a centralized directory (e.g., the official Firefox Add-on Directory) can be protected from malicious extensions via a review process that detects and removes malware from the directory. Examples of malicious extension installation tactics include:

- **Trojan.** Infostealer.Snifula pretends to be the legitimate extension NumberedLinks. In addition to acting as the NumberedLinks extension, it also logs the contents of forms and sends them to the attacker [33].
- **Impersonation.** One piece of spyware known as TSPY_EBOD.A claims to be an Adobe Flash Player update [13]. It monitors the user's browser activities and sends them to the attacker. It spreads by advertising itself in forum posts without the user's knowledge. Unlike a trojan, TSPY_EBOD.A does not actually include any legitimate functionality (i.e., it does not bundle itself with a real Flash player update).
- **Add-on directory.** The Firefox add-on directory contains "experimental" extensions in the process of being reviewed. Master Filer was listed as undergoing the review process. It claimed to have actual functionality but all versions installed malware [26].

Installation warnings about high extension privileges could be an effective deterrent if infrequent. Static permissions would also simplify the review process.

2.3 Benign-but-buggy Extensions

In the *benign-but-buggy* threat model, we assume the extension developer is well-intentioned but not a security expert. The attacker is another party who takes advantage of a vulnerability in the extension and usurps the extension's privileges. If the vulnerable extension has access to powerful privileges, the attacker might be able to install malware on the user's machine, intercept web passwords, or perform other similarly dangerous actions.

The attacker can be a web attacker or an active network attacker. The web attacker controls a web site, canonically `https://attacker.com/`, that the user visits. (Note that we do not assume that the user confuses the attacker's web site with another web site.) Typically, the attacker attempts to corrupt an extension when the extension interacts with the attacker's web site. In addition to the abilities of a web attacker, an active network attacker can intercept, modify, and inject network traffic (e.g., HTTP responses). The active network attacker threat model is appropriate, e.g., for a wireless network in a coffee shop.

Firefox extensions combine two dangerous qualities: high privileges and rich interaction with untrusted web content. Taken together, these qualities risk exposing powerful privileges to attackers. We describe four classes of attacks against browser extensions and the relevant mitigations provided by the Firefox extension system:

- **Cross-Site Scripting.** Extension cross-site scripting (XSS) vulnerabilities result from interacting directly with untrusted web content. For example, if an extension uses `eval` or `document.write` without sanitizing the input, the attacker might be able to inject a script into the extension. In one recent example [21], a popular RSS aggregation extension evaluated data from the `<description>` element of an arbitrary web site without proper sanitization. To help mitigate XSS attacks, Firefox provides a sandbox API, `evalInSandbox`. When evaluating a script using `evalInSandbox`, the script runs without the extension's privileges, thereby preventing the script from causing much harm. However, use of this sandbox evaluation is discretionary and does not cover every kind of interaction with untrusted content. For example, sharing through the DOM can provide the untrusted page with access to the extension's prototype chain.

- **Replacing Native APIs.** A malicious web page can confuse (and ultimately exploit) a browser extension by replacing native DOM APIs with methods of its own definition. These fake methods might superficially behave like the native methods and trick an extension into performing some misdeed [9]. To help mitigate this class of attack, Firefox automatically wraps references to untrusted objects with an `XPCNativeWrapper`. An `XPCNativeWrapper` is analogous to X-ray goggles: viewing a JavaScript object through an `XPCNativeWrapper` shows the underlying native object, ignoring any modifications made by the page's JavaScript. However, the `XPCNativeWrapper` security mechanism has had a long history of implementation bugs [4, 3, 1]. Recent work has demonstrated that these bugs are exploitable in some extensions [21].

- **JavaScript Capability Leaks.** JavaScript capability leaks [11] are another avenue for exploiting extensions. If an extension leaks one of its own objects to a malicious web page, the attacker can often access other JavaScript objects, including powerful extension APIs. For example, an early version of Greasemonkey exposed a privileged version of `XMLHttpRequest` to every web page [32], letting attackers circumvent the browser's same-origin policy by issuing HTTP requests with the user's cookies to arbitrary web sites and reading back the responses.

- **Mixed Content.** An active network attacker can control content loaded via HTTP. The most severe form of this attack occurs when a browser extension loads a script over HTTP and runs it. The attacker can replace this script and hijack the extension's privileges to install malware. A similar, but less powerful, attack occurs when an extension injects an HTTP script into an HTTPS page. For example, we discovered that an extension [6] injects an HTTP script into the HTTPS version of Gmail. (We reported this vulnerability to the extension's developers on August 12, 2009, and the developers released a fixed version the next week.)

Even though we might be able to design defenses for each of these attack classes, we argue that the underlying issue is that Firefox extensions interact directly with untrusted content while possessing a high level of privilege.

3 Privilege Gap Study

A natural approach to avoid malware and mitigate extension vulnerabilities is to reduce the privileges granted to extensions. To evaluate the feasibility of this approach, we study 25 popular Firefox extensions to determine how much privilege each needs to implement its features. We find that most extensions do not require arbitrary file system access but all receive it, meaning that most Firefox extensions are over-privileged.

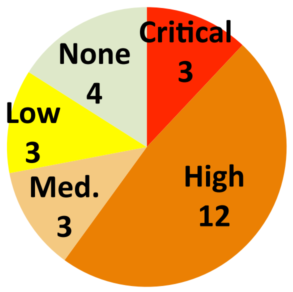


Figure 1. The severity rating of the most dangerous behavior exhibited by each extension.

3.1 Methodology

We randomly selected two extensions from each of the 13 categories in the "recommended" section of the Firefox Add-on directory. (See Appendix A for a list.) We excluded one of the selected extensions because it was distributed only as a binary; all of the others were entirely written in JavaScript. We verified that the 25 subject extensions were also highly ranked in the "popular" directory.

To determine the extensions' functionality, we ran each extension and manually exercised its user interface. We compared the user interface behavior with the source code until we felt certain we had explored the full functionality of the extension. We did not automate this process because we felt that understanding "behavior" (as opposed to implementation) requires human judgement. We then assigned one of five ratings to each behavior based on the Firefox Security Severity Ratings [8]:

- *Critical:* Can run arbitrary code on the user's system (e.g., arbitrary file access)
- *High:* Can access site-specific confidential information (e.g., cookies and password) or the Document Object Model (DOM) of all web pages
- *Medium:* Can access private user data (e.g., recent history) or the DOM of specific web pages
- *Low:* Can annoy the user
- *None:* No security privileges (e.g., a string class) or privileges limited to the extension itself

3.2 Results

Of the 25 subject extensions, only 3 require critical privileges (see Figure 1). Therefore, 22 of the subject extensions are over-privileged because all extensions have the ability to perform critical tasks. We summarize these results below:

- Three extensions, all download managers (one cross-listed in another category), require the ability to create new processes. (These are the only three extensions that actually require critical privileges.) One extension converts file types using system utilities, another runs a user-supplied virus scanner on downloaded files, and the third launches a new process to use the operating system's shutdown command.
- None of the extensions we studied require arbitrary file access. Several extensions access files selected by a file open dialog, and most use files to store extension-local data. The download managers interact with files as they are downloaded.

- 17 extensions require network access (e.g., observing raw browser network data or making XMLHttpRequests) and/or web page access (i.e., manipulating a displayed page's DOM). 10 require network access and 11 require access to web pages. Of the 10 extensions that require network access, 2 require access only to a specific set of origins. Of the 11 that need access to web sites, 2 extensions are satisfied with specific origins and 3 extensions request content for display only (and do not need access to the result). All of these extensions require access to both the HTTP and HTTPS versions of domains.
- A number of extensions make legitimate use of credential-related managers (password, login, and cookie managers). Extensions that are affiliated with specific sites want to authenticate users. For example, the Delicious extension checks the `delicious.com` and `de.li.ciou.us` cookies to see if a user is currently logged in. All of these extensions limited their credential actions to a small number of sites.
- Nearly all of the extensions require access to an extension-local preference store to persist their own preferences, but only one changes global browser preferences (to switch languages).

3.3 Discussion

Although every Firefox extension runs with the user's full privileges, only three of the extensions we analyze actually require these privileges. The remaining 22 extensions exhibit a *privilege gap*: they run with more privileges than required. Almost all of the extensions can function without any local system access, which suggests that an extension system could be successful without providing this privilege at all. The infrequency of critical privileges also suggests that special treatment for extensions with local system access (e.g., dire installation warnings or a special review process) is feasible. Since local system access is the most critical privilege, we feel there is strong motivation to pursue a least privilege system.

As further motivation for a least privilege system, 7 of the 25 extensions only require low or no privileges. These extensions could be passed through a review process without review or installed by a user with high confidence that they do not pose a threat. Giving these extensions powerful permissions is a completely unnecessary risk.

Seventy percent of extensions in our study require network or web page access. This indicates that removing network or DOM access would cripple an extension system. Unfortunately, it is not currently possible to selectively grant network/DOM access without revealing passwords and form contents (making access to credential managers redundant). This is a negative finding: network/DOM access can be used to spy on user's browsing activity, and this study shows that most extensions require this ability. Access to this popular privilege is consequently not sufficient as an indicator of possible malware. However, removing network/DOM access from 30% of extensions would still significantly reduce the possibility of turning a benign-but-buggy extension into spyware. An additional number of benign-but-buggy extensions would benefit from limiting network/DOM access to a small, defined set of origins (i.e., which would likely not include banks).

4 Retrofitting the Firefox API

We consider how the Firefox extension system could be retrofitted to support limiting extensions to certain sets of privileges. The Firefox API design is central to privilege reduction because it determines how permissions can be allocated. If performing a common low-privilege operation requires the use of a high-privilege interface, then extensions will inappropriately need high privileges to function. We also present an algorithm for finding places in the API that require monitoring to prevent extensions from escaping their initial set of permissions.

4.1 Interface Privilege Gap

If we limit an extension to precisely the set of interfaces it requires to function, is there still a privilege gap? If so, this would indicate that some interfaces are more powerful than they need to be. We analyze the implementation of the 25 extensions from our behavioral study (Section 3) to determine how much power the extension's implementation would *receive* if the extension were limited to the set of interfaces in its current implementation. We then compare that to the amount of power the extension *needs* to function.

We identified use of the extension system API by searching for explicit interface names in the extensions' source code. This methodology under-approximates the set of interfaces. We then manually correlated the interfaces with the extensions' functionality. We rated each interface according to its security level, using the same criteria as when we rated extension behavior (Section 3.1).

We find that extensions commonly use powerful interfaces to accomplish simple tasks because the Firefox APIs are coarse-grained. Despite the fact that only 3 display critical-level behavior, 19 use a critical-rated interface (see Figure 2); this means that 16 would be over-privileged if we were to implement a least-privilege system. An additional 3 use high-rated interfaces despite needing only medium or lower privileges, for a total of 19 extensions that use interfaces that would be inappropriately privileged in a least-privilege system. The interfaces that are causing the privilege gap would need to be adjusted to more appropriately reflect extensions' intended use. Of the 228 interfaces used by the twenty-five extensions, we rated 7% critical and 23% high. These interfaces would require further scrutiny. Notably, however, 56% of the referenced interfaces require no privileges at all and therefore do not require changes. These findings are not indicative of a flaw in the Firefox API; they are a natural consequence of the API being designed with only functionality in mind. The API was not built to accommodate a least privilege system, so it is expected that it would require retrofitting.

More specifically, Figure 3(a) shows common security-relevant behaviors and identifies whether there is a privilege gap. The file system interface is a common point of excessive privileges. Most extensions use the file system interface, which can read and write arbitrary files. These extensions could make use of lower-privilege file storage interfaces if such interfaces existed. For example, 11 of the extensions could be limited to files selected by the user via a file open dialog (analogous to the HTML file upload control), and 10 extensions could be limited to an extension-local persistent store (like the HTML 5 `localStorage` API) or an extension-specific directory. The download managers could also be limited to the downloads folder. Another common point of over-privilege is the preference service, which most extensions use to store extension-local preferences. This service can also change browser-wide settings and preferences belonging to other extensions.

Figure 3(b) shows a breakdown of the interface usage of extensions. The number of critical interfaces holds steady across all extensions, even as the total number of interfaces used by each extension differs greatly. Further examination shows that the same set of critical interfaces are used by almost all extensions. Separating the popular file- and preference-related interfaces into multiple interfaces grouped by appropriate privilege level would be a significant first step towards reducing the API privilege gap; see Appendix C for a more detailed discussion.

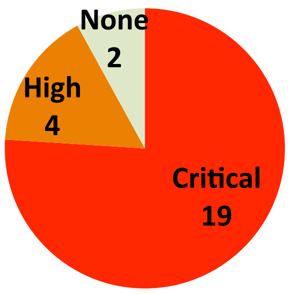
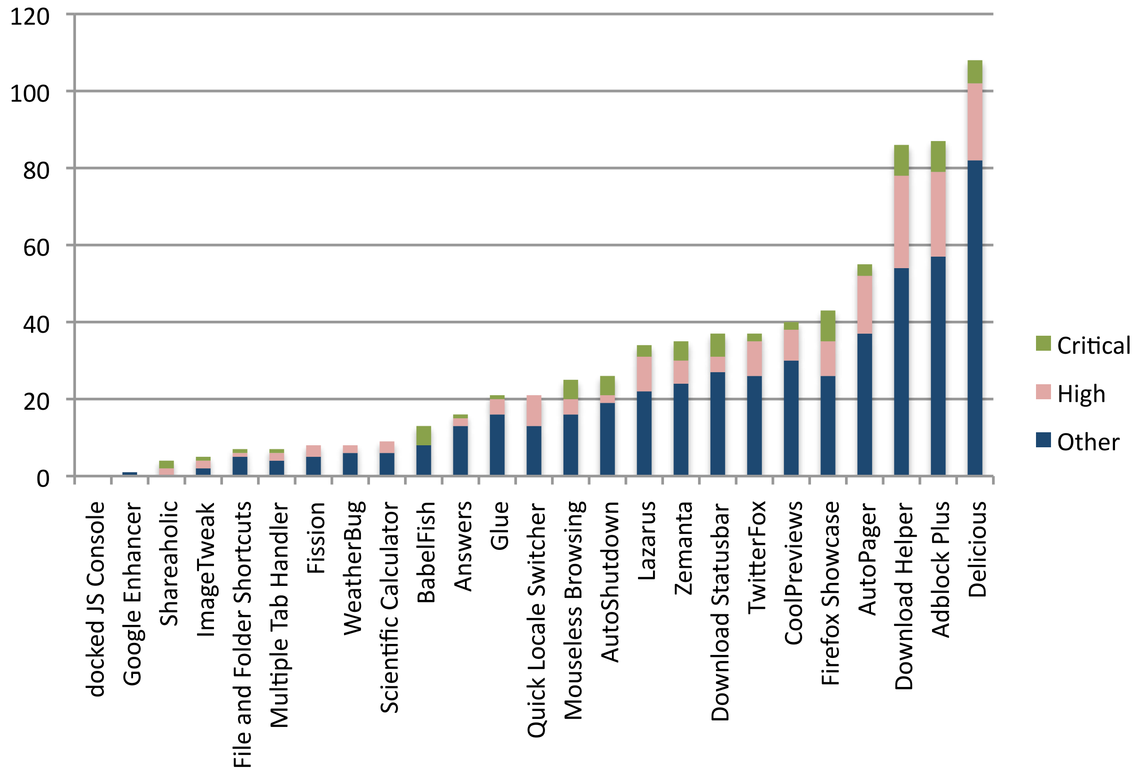


Figure 2. The severity rating of the most dangerous interface requested by each extension.

Behavior	**Interface**	**Disparity?**	**Frequency**	
Process launching (C)	Process launching (C)	No	3	(12%)
User chooses a file (N)	Arbitrary file access (C)	Yes	11	(44%)
Extension-specific files (N)	Arbitrary file access (C)	Yes	10	(40%)
Extension-specific SQLite (N)	Arbitrary SQLite access (H)	Yes	3	(12%)
Arbitrary network access (H)	Arbitrary network access (H)	No	8	(40%)
Specific domain access (M)	Arbitrary network access (H)	Yes	2	(8%)
Arbitrary DOM access (H)	Arbitrary DOM access (H)	No	9	(36%)
Page for display only (L)	Arbitrary DOM access (H)	Yes	3	(12%)
DOM of specific sites (M)	Arbitrary DOM access (H)	Yes	2	(8%)
Highlighted text/images (L)	Arbitrary DOM access (H)	Yes	2	(8%)
Password, login managers (H)	Password, login managers (H)	No	3	(12%)
Cookie manager (H)	Cookie manager (H)	No	2	(8%)
Same-extension prefs (N)	Browser & all ext prefs (H)	Yes	21	(84%)
Language preferences (M)	Browser & all ext prefs (H)	Yes	1	(4%)

(a) The frequency of security-relevant behaviors. The security rating of each behavior is abbreviated in parentheses. If the interface's privilege is greater than the required behavioral privilege, there is a disparity.



(b) Interface usage breakdown, showing how many security-relevant interfaces were used by each extension.

Figure 3. Results of the interface survey.

4.2 Reachability Analysis

Even if a developer explicitly requests only a small number of interfaces, other interfaces could be reachable from that set. For example, a developer might request access to a low-type object with a method that returns a critical-type object; even though the developer has not asked for the critical-type object, it is available. We consider this a form of privilege escalation. To fully limit the privilege levels of extensions, we must control these *escalation points*, either by adding a reference monitor (e.g., to implement an access control approach) or by taming the interface (e.g., to implement an object-capability approach). We analyze a subset of the Firefox extension API to find these escalation points.

We analyze the API for escalation points by organizing XPCOM interfaces into a *security graph*. The nodes in our security graph are interface types and the directed edges represent reachability. We manually label the severity of 613 interfaces (of 1582 total), including all the interfaces used by the subject extensions, and insert them into the graph. We then automatically compute when an extension with a reference to one interface might be able to obtain a reference to another interface by deductive inference on the types used in the interfaces.

Our deductive system computes which additional interfaces a principal (the browser or an extension) can obtain from one starting interface. We are able to do this because Firefox XPCOM APIs are strictly typed and defined in an Interface Description Language (IDL). We compiled the type signatures from the Firefox 3.5 XPCOM interfaces by adding a Datalog back-end to the Firefox IDL compiler. Our deductive system analyzes these interfaces using the following rules:

- **Subtyping.** If an interface is a subtype of another interface, then the supertype is reachable from the subtype.
- **Method.** Access to a type implies access to that type's methods.
- **Getter.** A getter is treated as a method with no parameters that returns its relevant type.
- **Setter.** A setter is treated as a method with no return value that takes its relevant type as a parameter.
- **Type forgery.** An extension can pretend to have an instance of a type to pass in as a parameter to a method that requires that type. Eventually an error may be raised, but there is no guarantee as to when. This rule is appropriate for XPCOM because an extension can create a JavaScript object that implements an XPCOM interface by implementing the requisite methods and announcing support in its `queryInterface` method.
- **Return.** Anyone who can invoke a type's method (or getter) receives a return type. The return type is therefore reachable from the type that the method is attached to. Note that the parameters' types don't matter because of the type forgery rule.
- **Parameter.** When a method (or setter) is invoked, it receives access to the input parameter types. We add an edge from the type that the method is attached to to the parameter types.

We formalize these rules in Figure 4(a). The rules track which principal implements each concrete instance of the interface. Note that our internal state tracks both extension and browser reachability, but our output graph only reports the interfaces reachable by the extension. We write $\rho \hookrightarrow^{\eta} \alpha$ when principal ρ has a reference to an interface α implemented by principal η. This edge is added to the graph when ρ is the extension and η is the browser.

Notice that the type forgery rule permits us to reason about each interface individually instead of requiring us to build a graph over sets of interfaces. If the type forgery rule were absent, interface dependencies would appear because certain types would be needed as parameters to complete method calls. As a result, certain sets of interfaces would be more powerful than others based on their ability to fulfill method parameter type requirements. However, the type forgery rule removes the restriction that certain types are needed to invoke a method. Consequently, we can evaluate each interface on its own, without considering what it could reach in the presence of other interfaces.

$$\frac{\vdash \rho \hookrightarrow^{\eta} \alpha \qquad \Vdash \alpha.\mathit{subtype}(\beta)}{\vdash \rho \hookrightarrow^{\eta} \beta}\textsc{Subtyping} \qquad \frac{\vdash \rho \hookrightarrow^{\eta} \alpha \qquad \Vdash \alpha.\mathit{method}(\beta)}{\vdash \rho \hookrightarrow^{\eta} \beta}\textsc{Method}$$

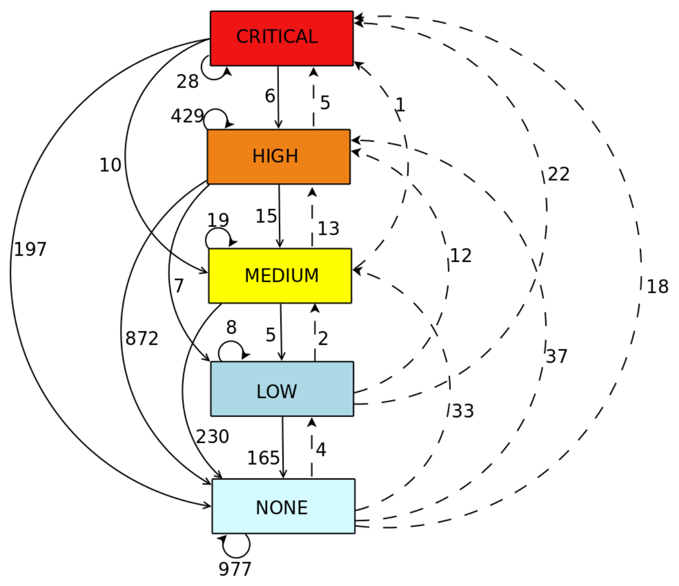
$$\frac{\Vdash \alpha.\mathit{getter}(\beta)}{\Vdash \alpha.\mathit{method}(1 \to \beta)}\textsc{Getter} \qquad \frac{\Vdash \alpha.\mathit{setter}(\beta)}{\Vdash \alpha.\mathit{method}(\beta \to 1)}\textsc{Setter}$$

$$\frac{}{\vdash \rho \hookrightarrow^{\rho} \alpha}\textsc{Type Forgery}$$

$$\frac{\vdash \rho \hookrightarrow^{\eta} \alpha \to \beta \qquad \vdash \rho \hookrightarrow^{\gamma} \alpha \qquad \vdash \eta \hookrightarrow^{\delta} \beta}{\vdash \rho \hookrightarrow^{\delta} \beta}\textsc{Return}$$

$$\frac{\vdash \rho \hookrightarrow^{\eta} \alpha \to \beta \qquad \vdash \rho \hookrightarrow^{\gamma} \alpha \qquad \vdash \eta \hookrightarrow^{\delta} \beta}{\vdash \eta \hookrightarrow^{\gamma} \alpha}\textsc{Parameter}$$

(a) Inference rules for reachability in a type system with type forgery, such as the Firefox extension API.



(b) Firefox extension API reachability graph. Upward edges (dashed) could lead to privilege escalation.

Figure 4. Reachability analysis rules and results

Our deductive system is an over-approximation because we do not consider the actual implementation of the interfaces. Deductions based on the handling of input parameters might be overly conservative because it is not known which methods are called on the input parameters in the implementation. For example, suppose type `foo` has a method that accepts type `bar` as a parameter, and suppose type `bar` has a method `getFile` that returns a file type. Without consulting the implementation, we cannot know whether `foo` actually ever calls `bar.getFile`, but we know it is possible. Our conservative analysis therefore concludes that the file type is reachable from `foo`.

We computed the security reachability graph for the Firefox extension interfaces by implementing our rules in Datalog. Figure 4(b) summarizes the graph by coalescing interfaces with the same security rating into a single vertex and contracting the unlabeled interfaces. (We labeled 613 of 1582 total XPCOM interfaces.) Of the 2920 edges in the graph, 147 edges go "up" the graph. These upward edges represent potential escalation points that make reducing the privilege of extensions difficult.

Because our analysis is an over-approximation, some of these edges might not actually be exploitable given the Firefox implementation of the extension interfaces. However, even these edges might become exploitable if an extension replaces the built-in implementation of the relevant interface. To retrofit security onto the Firefox extension API, we recommend preventing privilege escalation by removing these edges, either by adding runtime access control checks or by taming the interfaces at design time. We suggest that any new extension system be aware of this issue and avoid introducing escalation points into their API.

5 Google Chrome Extensions

In the new Google Chrome extension system, extensions run with a restricted set of privileges instead of the user's full privileges. The browser grants an extension access only to the privileges explicitly requested in the extension's manifest file. Different installation warnings are displayed to the user based on the privilege level of the extension. The extension system also provides mechanisms for isolating extensions from web content, to make extensions less vulnerable to web attackers. In this section, we evaluate the success of their least privilege design and test the performance of their isolation mechanisms.

5.1 API Evaluation

The Google Chrome extension system API is categorized by functionality (e.g., `tabs`), and different interface groups do not reference one another. Furthermore, the interfaces themselves do not grant any critical- or high-level privileges. Extensions that want critical privileges must include a native binary, and they are not permitted in the official extension directory unless the developer signs a contract with Google. Extensions that want web site access ask in their manifests for access to either a specific set of domains or all domains. An extension can register content scripts for the `file` URI scheme, but it will be treated as if it contained a native binary (i.e., a contract is required). Figure 5 shows how extensions implement behaviors that exhibited a permission disparity in the Mozilla Firefox system. The Google Chrome API is largely successful at avoiding the privilege gap.

The Google Chrome extension system still exhibits a privilege gap for partial DOM access. Partial DOM access allows an extension to modify page content or layout without exposing, for example, the contents of HTTPS forms or password fields. The proposed Gleam API addresses this issue: it lets an extension register to operate on text, images, links, videos, and selected text without receiving the ability to access the DOM [2]. Although the Gleam API is a step in the right direction, it does not fully address the technical challenge of providing partial DOM access. Some extensions need direct and arbitrary access to the DOM (e.g., to make structural changes to the page). Recent work on securely sharing and restricting DOM objects might be applicable to this problem [24].

One potential problem with the Google Chrome extension API is that it is not as full-featured as the Mozilla Firefox extension API. There may be extensions that are not possible to build with the Google Chrome extension

APIs alone. Consequently, a developer might write an extension with a native binary to implement this functionality. The extension would thereby gain access to file system privileges without necessarily needing them. This can be addressed in two ways: increasing the robustness of the API, or leveraging techniques for sandboxing untrusted native code in the browser [34].

5.2 Reduced Privileges

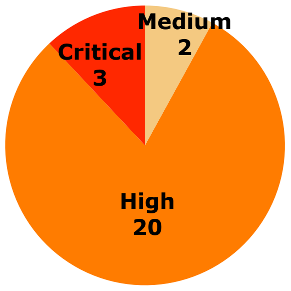
We evaluate the privileges of the 25 most popular Google Chrome extensions [16]. (See Appendix B for a list.) For each extension, we examine its manifest to determine the privileges requested by the extension. We determine extension behavior by manually exercising the user interface and reviewing the source code. We then measure the privilege gap by comparing the extension's requested privileges with its behavior. The purpose of this study is twofold: to determine whether developers are requesting the correct set of least privileges for their extensions, and to examine to what degree Google Chrome extensions have similar privilege needs to the extensions in our Mozilla Firefox study. The Google Chrome extensions we survey are not as diverse as the extensions in the Firefox survey because the Google Chrome extension platform is new, extensions are not sorted by category, and 9 of the extensions were developed by Google employees.

Figure 6(a) shows the highest severity privileges that extensions ask for in their manifests. Of the three critical-rated extensions, two include binary code and one injects JavaScript into documents from the local file system when they are loaded in the browser. The 20 high-rated extensions request arbitrary web/network access, and the two medium-rated extensions request web/network access for a limited number of origins. We found one extension (Cooliris) that requests more privileges than required to implement its behavior. This extension requires 3D accelerated graphics, which could be provided to extensions without granting the extension the ability to run arbitrary code. (The current APIs do not provide this functionality.) However, the rest of the extensions request appropriate privileges given their feature set. We conclude that extensions in the Google Chrome extension system do not typically need local file system access and therefore possess significantly fewer privileges than extensions in the Firefox extension system. More Google Chrome extensions require high-level privileges; we believe this is due to the less varied nature of new Google Chrome extensions.

In addition to analyzing the highest severity privilege requested by an extension overall, we also examine how privileges are distributed within the extension (see Figure 6(b)). Even if the extension as a whole has arbitrary DOM access via content scripts, its core might have access to only a limited set of origins, or vice versa. During our review, we found that two extensions requested more privileges for their core than necessary. This indicates that a small number of developers might have difficulty reasoning about how to request the appropriate privileges for different extension subcomponents. However, the privileges requested for the core extension were a subset of the privileges requested for the content scripts, leaving the overall privilege level of the extension the same.

Behavior	**Implementation**
User chooses a file (N)	File picker, with `<input type='file'>` (N)
Extension-specific data & preferences (N)	HTML5 storage (N)
Specific-domain network access (M)	List specific XHR domains in manifest (M)
DOM of specific sites (M)	List specific domains in manifest (M)
Page for display only (L)	Open a new frame with the page (L)
Highlighted text/images (L)	Full DOM access required (M/H)

Figure 5. The Google Chrome extension system is fine-grained.



(a) Highest privilege.

Privilege	# of extensions
Content script DOM access	
`file` scheme	1
All sites	14
Limited number of sites	3
No sites	8
Core extension network/DOM access	
All sites	12
Limited number of sites	8
No sites	5
Other APIs	
Tabs	20
Bookmarks	2
Plugin	2

(b) Privilege use breakdown.

Figure 6. Results of our survey of 25 popular Google Chrome extensions.

5.3 Performance

Separating extensions into components has the benefit of reducing the severity of a bug in one component, but it could potentially add overhead to inter-component operations. For example, if a content script needs to use privileges held by the extension core, the content script has to send a message to the core process instead of simply calling a function in its own address space.

To evaluate the run-time overhead of inter-process communication, we measured the round-trip latency for sending a message from a content script to the extension core in Google Chrome 4.0.249.22 on Mac OS X. We observe an average round-trip latency of 0.8 ms ($n = 100$, $\sigma = 0.0079$ ms), where each trial is the average of 1000 inter-process round-trips. Of course, an extension incurs this added latency only for operations that require coordination between multiple components. For example, an extension that adds additional EXIF metadata to Flickr [28] incurs this overhead once per page load to issue a cross-origin XMLHttpRequest, increasing the load time by an unnoticeable 0.8 ms.

Google Chrome shields buggy content scripts from malicious website operators by running content scripts in *isolated worlds*. With isolated worlds, content scripts and web pages have different JavaScript objects for the same underlying DOM nodes. This prevents content scripts and web pages from ever exchanging JavaScript pointers, making it more difficult for a malicious web page to confuse the content script. Consequently, DOM access from a content script incurs an additional hash table lookup on some execution paths.

To evaluate the run-time overhead of the isolated words mechanism, we ran a DOM core performance benchmark [19] in Chromium 4.0.266.0 on Mac OS X. The benchmark measures the total speed of a set of append, prepend, insert, index, and remove DOM operations. In the main world, the benchmark required an average of 231 ms ($n = 100$, $\sigma = 5.46$ ms) to complete. When run in an isolated world, the benchmark took an average of 309 ms ($n = 100$, $\sigma = 6.33$ ms). The use of isolated worlds adds 33.3% to DOM access time, which we expect would be a small fraction of overall run and load time.

6 Related Work

In addition to the Firefox extension system we analyze in this paper, Firefox has a second, experimental extension system: Jetpack [27]. Jetpack exposes browser functionality via narrow interfaces. Currently, however, each Jetpack extension runs with the user's full privileges and has access to the complete Firefox extension API. As Jetpack matures, we expect the Firefox developers to restrict the privileges of Jetpack extensions, but the designers of Jetpack have chosen to focus first on usability and developer productivity [29].

Internet Explorer has a combined plug-in and extension system known as Browser Helper Object (BHO) modules. For example, the Yahoo Toolbar for Internet Explorer is implemented as a BHO. These extensions are written in native code and have direct access to the win32 API. If a BHO has a vulnerability (such as a buffer overflow), a malicious web site can issue arbitrary win32 API calls by exploiting the vulnerability. Recent versions of Internet Explorer run these BHOs in "protected mode," [25] reducing their privileges. However, a compromised BHO still has full access to web pages (including passwords and cookie) and read access to the file system.

One recent paper [22] considers limiting the privileges of Firefox extensions. They propose a mechanism for sandboxing extensions by intercepting various events in the XPCOM object marshaling layer, incurring a performance overhead of 19% for a particular policy. Unlike our work, this paper focuses entirely on mechanism, and the authors do not determine which policies their mechanism ought to enforce. We could imagine reducing the privileges of Firefox extensions by using this mechanism to restrict extension behavior at the escalation points we identify in Section 4.2.

A number of papers [14, 34, 7, 15, 30] consider the problem of running native plug-in code securely using fault isolation and system call interposition. These techniques focus on isolating untrusted native code, whereas we focus on code written in JavaScript, letting us use the standard same-origin JavaScript sandbox. We are chiefly concerned with the privileges afforded to extensions via explicit APIs, a topic that has not been studied in much detail. Their techniques for plug-in confinement are complimentary to our work and could be used to monitor native binaries distributed with extensions.

7 Summary

Browser extension systems face two threats: malicious extensions and benign extensions with security vulnerabilities. By default, all Mozilla Firefox extensions receive the user's full privileges. To evaluate whether extensions actually require such a high level of privilege to implement their feature set, we analyze 25 "recommended" extensions from the Firefox extension gallery. We find that the majority of these extensions do not require full privileges. This motivates a new extension system that would reduce the privilege levels of extensions.

We examine the feasibility of retrofitting the Firefox extension system with the ability to contain extensions to only the parts of the API needed to function. However, reducing the privileges of existing Firefox extensions is difficult because many Firefox APIs are more powerful than required to implement extension features. We provide recommendations for altering the Firefox API.

The new Google Chrome extension system was partially motivated by the findings of this thesis's extension survey. Its design requires developers to ask for specific permissions at install time. This alerts users to extensions with suspiciously high privilege requirements and, if used correctly by developers, reduces the severity of extension vulnerabilities. We survey 25 Google Chrome extensions and find that most of the developers understand how to ask for the correct amount of privilege.

Acknowledgments

I would like to thank my colleagues, Adam Barth and Prateek Saxena, for their collaboration. This work was partially supported by National Science Foundation TRUST Grant No. CCF-0424422.

References

[1] Arbitrary code execution using bug 459906.
https://bugzilla.mozilla.org/show_bug.cgi?id=460983.

[2] Gleam API.
http://www.chromium.org/developers/design-documents/extensions/gleam-api.

[3] Mozilla Security Advisory 2009-19.
http://www.mozilla.org/security/announce/2009/mfsa2009-19.html.

[4] Mozilla Security Advisory 2009-39.
http://www.mozilla.org/security/announce/2009/mfsa2009-39.html.

[5] Skype. https://developer.skype.com/SkypeToolbars.

[6] Zemanta. http://www.zemanta.com.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[8] L. Adamski. Security Severity Ratings. https://wiki.mozilla.org/Security_Severity_Ratings.

[9] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript Environments. In *3rd USENIX Workshop on Offensive Technologies*, 2009.

[10] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, Google, 2008.

[11] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security Symposium*, 2009.

[12] BitDefender. Trojan.pws.chromeinject.b:
http://www.bitdefender.com/VIRUS-1000451-en--Trojan.PWS.ChromeInject.B.html.

[13] D. Caraig. Firefox Add-On Spies on Google Search Results. http://blog.trendmicro.com/firefox-addo-spies-on-google-search-results/, 2009.

[14] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *USENIX Operating System Design and Implementation*, 2008.

[15] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[16] Google. Google Chrome Extensions: Most popular gallery. https://chrome.google.com/extensions/list/popular.

[17] C. Grier, S. T. King, and D. S. Wallach. How I Learned to Stop Worrying and Love Plugins. In *Web 2.0 Security and Privacy*, 2009.

[18] C. Grier, S. Tang, and S. T. King. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, 2008.

[19] I. Hickson. DOM Core Performance, Test 1.
http://www.hixie.ch/tests/adhoc/perf/dom/artificial/core/001.html.

[20] kkovash. How Many Firefox Users Customize Their Browser? http://blog.mozilla.com/metrics/2009/08/11/how-many-firefox-users-customize-their-browser/, 2009.

[21] R. S. Liverani and N. Freeman. Abusing Firefox Extensions. Defcon17, July 2009.

[22] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. In *Journal in Computer Virology*, August 2008.

[23] McAfee. Form spy: http://vil.nai.com/vil/content/v_140256.htm.

[24] L. Meyerovich, A. P. Felt, and M. S. Miller. Object Views: Fine-Grained Sharing in Browsers. In *WWW*, 2010.

[25] Microsoft Developer Network. Introduction of the Protected Mode API. http://msdn.microsoft.com/en-us/library/ms537319(VS.85).aspx.

[26] Mozilla Add-ons Blog. Security Issue on AMO. http://blog.mozilla.com/addons/2010/02/04/please-read-security-issue-on-amo/, February 2010.

[27] Mozilla Labs. Jetpack.
https://wiki.mozilla.org/Labs/Jetpack.

[28] D. Pupius. Fittr Flickr Extension for Chrome.
http://code.google.com/p/fittr/.

[29] A. Raskin. Jetpack FAQ. http://www.azarask.in/blog/post/jetpack-faq/, 2009.

[30] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1994.

[31] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security Symposium*, 2009.

[32] S. Willison. Understanding the Greasemonkey vulnerability. `http://simonwillison.net/2005/Jul/20/vulnerability/`.

[33] C. Wuest and E. Florio. Firefox and Malware: When Browsers Attack. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/firefox_and_malware.pdf`, 2009.

[34] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

A Firefox Extension Survey

Our Firefox extension survey (Section 3) examines extensions from the Firefox Add-on "recommended" directory. We selected two from each category in the directory. The thirteen categories are: Alerts & Updates, Appearance, Bookmarks, Download Management, Feeds News & Blogging, Language Support, Photos Music & Videos, Privacy & Security, Search Tools, Social & Communication, Tabs, Toolbars, and Web Development.

The twenty-five extensions in our extension survey are: Adblock Plus 1.0.2, Answers 2.2.48, AutoPager 0.5.0.1, Auto Shutdown (InBasic) 3.1.1B, Babel Fish 1.84, CoolPreviews 2.7.4, Delicious Bookmarks 4.3, docked JS-Console 0.1.1, DownloadHelper 4.3, Download Statusbar 2.1.018, File and Folder Shortcuts 1.3, Firefox Showcase 0.3.2009040901, Fission 1.3, Glue 4.2.18, GoogleEnhancer 1.70, Image Tweak 0.18.1, Lazarus: Form Recovery 1.0.5, Mouseless Browsing 0.5.2.1, Multiple Tab Handler 0.9.5, Quick Locale Switcher 1.6.9, Shareaholic 1.7, Status-bar Scientific Calculator 4.5, TwitterFox 1.7.7.1, WeatherBug 2.0.0.4, and Zemanta 0.5.4.

B Google Chrome Extension Survey

Our Google Chrome extension survey (Section 5.1) examines extensions from the Google Chrome "most popular" directory. There are no official categories for Google Chrome extensions. Note that 9 of the extensions are made by Google-affiliated developers, as denoted with an asterisk.

The twenty-five extensions in our Google Chrome extension survey are: Google Mail Checker 1.2*, AdThwart 0.4.1, Google Translate 1.1.4*, IE Tab, Google Wave Notifier 2.2, RSS Subscription Extension 1.8.1*, Xmarks bookmark sync 0.5.24, Docs PDF/PowerPoint Viewer 1.5.3*, AdBlock 1.1.91, Google Quick Scroll 0.5.4*, CoolIris, Chromed Bird 1.2.0, Facebook for Google Chrome 1.3, Google Reader Notifier 1.1*, Google Calendar Checker 1.0.3*, SmoothScroll 0.6.1, Speed Tracer 0.6*, Evernote Web Clipper 1.1, Send from Gmail 1.11*, Bubble Translate 1.2, Chrome Gesture 1.8.0, AniWeather 0.6.19.2, FlashBlock 1.2.11.11, Select to Get Maps 1.1.1, StumbleUpon 1.0.11208.1.

C Popular Interfaces

As part of the Firefox extension survey in Section 4.1, we compiled a list of interfaces. There were 228 interfaces in total, and 735 references to those 228 interfaces. The majority of interfaces (nearly a hundred) are only used by one extension. Only ten are used by twelve or more extensions. Figure 7 shows how many interfaces were used a given number of times.

The most popular interfaces are particularly interesting, since they are responsible for the majority of the privilege gap. They are also a good indication of what functionality is important for a new API. However, a popularity ranking by interface is not a perfect gauge of important features (for example, there are four different interfaces relating to preferences, and they are scored separately).

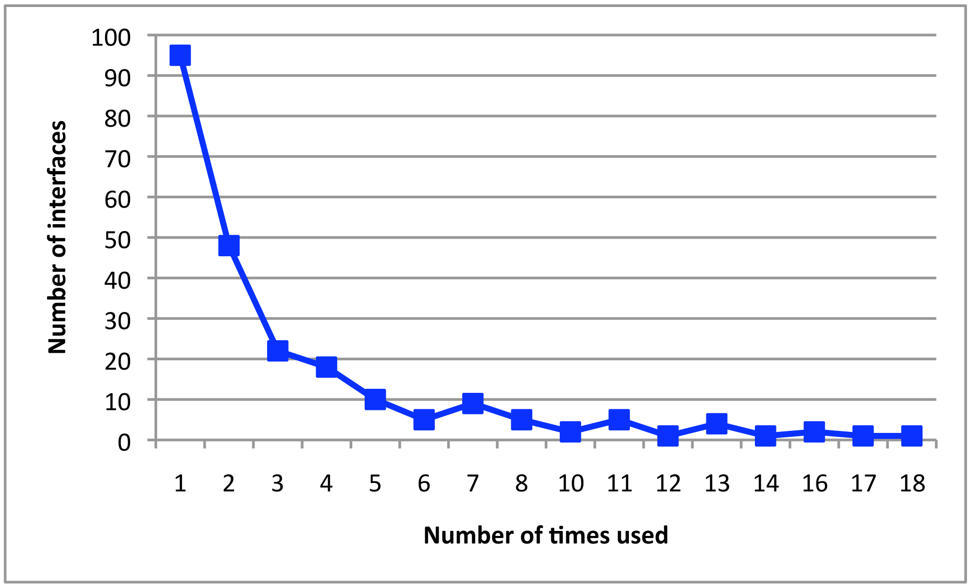


Figure 7. A frequency chart for interfaces. A large number of interfaces were only used once; a small number were used 10 or more times.

1. `nsIWindowMediator` (18) keeps track of open windows and can be used to get window objects. Severity: None.

2. `nsIIOService` (17) provides I/O helper functions, like URI parsing and getting protocol schemes. Severity: None.

3. `nsIPrefBranch` (16) is used for preferences. Severity: High. It can be used to edit any preference settings, whether they belong to other extensions or the browser itself. We recommend the creation of an extension-specific preferences store.

4. `nsIPrefService` (16) is also used for preferences. Severity: High.

5. `nsISupports` (14) is the base class from which all XPCOM interfaces inherit. Extensions' components must explicitly inherit from it in their definitions. It also provides the commonly used `QueryInterface` method, which provides runtime type discovery. Severity: None.

6. `nsIConsoleService` (13) provides logging services. Severity: None.

7. `nsIExtensionManager` (13) provides general extension management functions, such as checking for upgrades and incompatible extensions. Severity: Critical. It can add things to the download manager or install a new extension. Simple extension management functions (e.g., checking for an upgrade) could be separated from the more dangerous ones.

8. `nsIFile` (13) represents a file and has related methods. Severity: Critical. We suggest it could be replaced with immutable file handles returned by a file picker, or limited to extension-specific directories.

9. `nsILocalFile` (13) is a platform-independent file handler and its related methods. Severity: Critical.

10. `nsIPromptService` (12) displays dialogs similar to `window.alert` and `window.confirm`. Severity: None.