Protecting Browsers from Extension Vulnerabilities

Adam Barth
Adrienne Porter Felt
Prateek Saxena
Aaron Boodman

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-185
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-185.html

December 18, 2009

Acknowledgement

We would like to thank Nick Baum, Erik Kay, Collin Jackson, Matt Perry, Dawn Song, David Wagner, and the Google Chrome Team. This work is partially supported by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170.

Protecting Browsers from Extension Vulnerabilities

Adam Barth, Adrienne Porter Felt, Prateek Saxena
University of California, Berkeley
{abarth, afelt, prateeks}@eecs.berkeley.edu

Aaron Boodman
Google, Inc.
aa@google.com

Abstract

Browser extensions are remarkably popular, with one in three Firefox users running at least one extension. Although well-intentioned, extension developers are often not security experts and write buggy code that can be exploited by malicious web site operators. In the Firefox extension system, these exploits are dangerous because extensions run with the user's full privileges and can read and write arbitrary files and launch new processes. In this paper, we analyze 25 *popular Firefox extensions and find that* 88% *of these extensions need less than the full set of available privileges. Additionally, we find that* 76% *of these extensions use unnecessarily powerful APIs, making it difficult to reduce their privileges. We propose a new browser extension system that improves security by using least privilege, privilege separation, and strong isolation. Our system limits the misdeeds an attacker can perform through an extension vulnerability. Our design has been adopted as the Google Chrome extension system.*

1 Introduction

Web browser extensions are phenomenally popular: roughly one third of Firefox users have at least one browser extension [22]. 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 that launch the eponymous IP-telephony application [5]. Although there have been several recent proposals for new web browser architectures [18, 11, 32], little attention has been paid to the architecture of browser extension systems.

Many extensions interact extensively with arbitrary web pages, creating a large attack surface that attackers can scour for vulnerabilities. In this paper, we focus on *benign-but-buggy* extensions. Most extensions are not written by security experts, and vulnerabilities in benign extensions are worrisome because Firefox extensions run with the browser's full privileges. If an attacker can exploit an extension vulnerability, the attacker can usurp the extension's broad privileges and install malware on the user's machine. At this year's DEFCON, Liverani and Freeman presented attacks against a number of popular Firefox extensions [23]. In one example, if the user dragged an image from a malicious web page into the extension, the web site operator could install a remote desktop server on the user's machine and take control of the user's mouse and keyboard.

These attacks raise the question of whether browser extensions require such a high level of privilege. To investigate this question, we examine 25 popular Firefox extensions to determine how much privilege each one requires. We find that only 3 of the 25 extensions require full system access. The remainder are over-privileged, needlessly increasing the severity of extension vulnerabilities. An extension system that narrows this *privilege gap* would reduce the severity of extension exploits, but the Firefox extension platform does not provide sufficiently fine-grained privileges. For example, many extensions store settings with an interface that can read and write arbitrary files.

We propose a new extension system, built with security in mind. In particular, we aim to protect users from benign-but-buggy extensions by designing *least privilege*, *privilege separation*, and *strong isolation* into our extension system. Instead of running with the user's full privileges, extensions in our system are limited to a set of privileges chosen at install time. If an extension later becomes compromised, the extension will be unable to increase this set of privileges. In particular, our case studies of Firefox extensions suggest that most extensions do not require the privilege to execute arbitrary code; consequently, the privilege to execute arbitrary code will often be unavailable to an attacker who compromises an extension in our system.

In addition to limiting the overall privileges of each extension, our system further reduces the attack surface of extensions by forcing developers to divide their extensions into three components: content scripts, an extension core, and a native binary (see Figure 1):

- Each *content script* has direct access to the DOM of a single web page and is thereby exposed to poten-

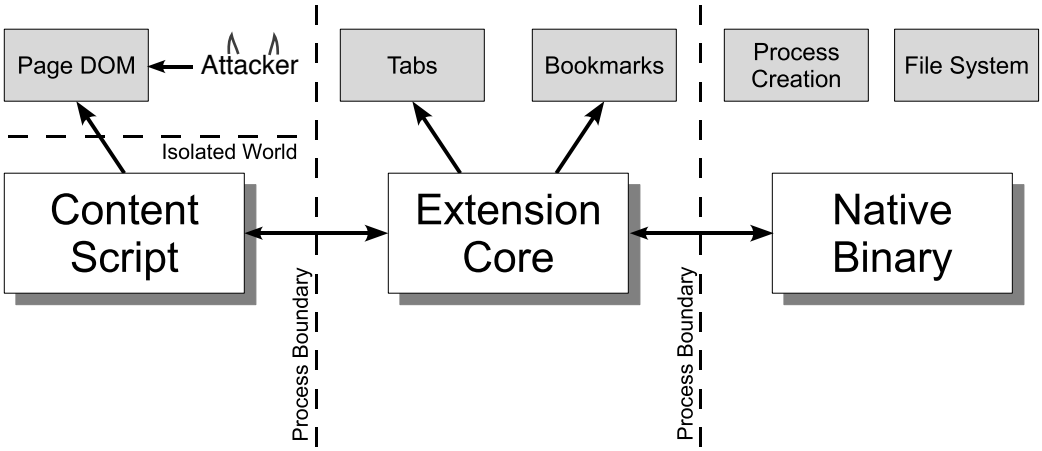


Figure 1. Extensions are divided into three components, each with progressively more privileges and less exposure to malicious web content.

tially malicious input. However, content scripts have no other privileges except for the ability to send messages to the extension core.

- The *extension core* contains the bulk of the extension privileges, but the extension core can only interact with web content via `XMLHttpRequest` and content scripts. Even the extension core does not have direct access to the host machine.

- An extension can optionally include a *native binary* that can access the host machine with the user's full privileges. The native binary interacts with the extension core via the standard NPAPI interface used by Flash and other browser plug-ins.

To gain the user's full privileges, an attacker would need to convince the extension to forward malicious input from the content script to the extension core and from the extension core to the native binary, where the input would need to exploit a vulnerability. We argue that exploiting such a multi-layer vulnerability is more difficult than exploiting a simple cross-site scripting hole in a Firefox extension.

Finally, the different components of an extension are isolated from each other by strong protection boundaries: each component runs in a separate operating system process. The content script and the extension core run in sandboxed processes that cannot use most operating system services. As a first layer of defense, the content script is isolated from its associated web page by running in a separate JavaScript heap. Although both the content script and the web page have access to the same underlying DOM, the two never exchange JavaScript pointers, helping prevent JavaScript capability leaks [12].

Our extension system design has been adopted by Google Chrome and is available in Google Chrome 4. Although it is difficult to predict how developers will use the extension system, we believe that this architecture will provide a solid foundation for building more secure extensions.

2 Attacks on Extensions

A browser extension is a third-party software module that extends the functionality of a web browser, letting users customize their browsing experience. Because extensions interact directly with untrusted web content, extensions are at risk of attack from malicious web site operators and active network attackers. In this section, we present a generic threat model for extension security that applies to both the Firefox extension system and the new extension system we introduce in this paper. We then focus our attention on the Firefox extension system, providing background material and examples of real attacks.

2.1 Threat Model

We focus on *benign-but-buggy* extensions: we assume the extension developer is well-intentioned but not a security expert. We assume attacker attempts to corrupt the extension and usurp its privileges. For example, the attacker might be able to install malware on the user's machine if the extension has arbitrary file access. We assume the attacker is unable to entice the user into downloading or running native executables. We further assume the browser itself is vulnerability-free, letting us focus on the additional attack surface provided by extensions.

We consider two related threat models: a web attacker and 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.

Plug-ins. In this paper, we focus on browser extensions, which differ from browser plug-ins. Plug-ins render specific media types (such as PDF and Flash) or expose additional APIs to web content (such as the Gears APIs). 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. Although plug-in security is an important area of research [18, 17], securing browser extensions requires different techniques.

2.2 Exploiting Firefox Extensions

In Firefox, browser extensions run with the same privileges as the browser itself. Firefox extensions have full access to browser internals and the user's operating system. Extensions can change the functionality of the browser, modify the behavior of web sites, run arbitrary code, and access the file system. Firefox extensions combine two dangerous qualities: high privilege 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 [23], 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.

- **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 [9] and trick an extension into performing some misdeed. 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, this 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 [23].

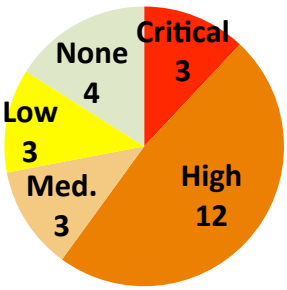
- **JavaScript Capability Leaks.** JavaScript capability leaks [12] are another avenue for exploiting extensions. If an extension leaks one of its own objects to a malicious web page, the attacker can often access other JavaScript objects, including powerful extension APIs. For example, an early version of Greasemonkey exposed a privileged version of `XMLHttpRequest` to every web page [33], 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 developers of the extension on August 12, 2009, and the developers released a fixed version that operates only on the non-HTTPS version of Gmail.)

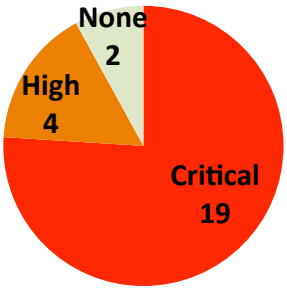
Even though we might be able to design defenses for each of these attack classes, we argue that the underlying issue is that Firefox extensions interact directly with untrusted content while possessing a high level of privilege.

3 Limiting Firefox Extension Privileges

A natural approach to mitigating extension vulnerabilities is to reduce the privileges granted to extensions. To evaluate the feasibility of this approach, we studied 25 popular Firefox extensions to determine how much privilege



(a) Most powerful behavior.



(b) Most powerful interface.

Figure 2. The chart on the left shows the severity ratings of the most dangerous behaviors exhibited by each extension. The chart on the right shows the security ratings of the extension interfaces used to implement these behaviors.

each needs to implement its features. In addition to presenting our case studies, we also present an algorithm for finding methods in the Firefox extension API that lead from a less-privileged interface to a more-privileged interface.

3.1 Case Studies

We review 25 extensions manually to determine their privilege requirements:

1. We analyze the behavior of an extension to determine how much privilege an extension *needs* to realize its functionality, letting us compare its required privileges to its actual privileges.

2. We analyze the implementation of an extension to determine how much power the extension *receives*, given the set of interfaces it uses to realize its functionality. This lets us evaluate how much we could reduce its privileges if we limited access to interfaces.

We find that most extensions do not require arbitrary file system access (the most powerful privilege), meaning that most extensions are over-privileged. We also find that extensions commonly use powerful interfaces to accomplish simple tasks because the Firefox APIs are coarse-grained.

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. 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 also located usage 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. This process could not be automated because understanding high-level functionality requires human judgement.

To compare the set of interfaces with extension functionality, we assigned one of five ratings (critical, high, medium, low, and none) to each interface and functionality. These ratings are 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) or privileges limited to the extension itself

Results. Of the 25 subject extensions, only 3 require critical privileges (see Figure 2(a)). Therefore, 22 of the subject extensions are over-privileged because all extensions have the privilege to perform critical tasks. Despite the fact that only 3 need critical privileges, 19 use a critical-rated interface (see Figure 2(b)). An additional 3 use high-rated interfaces despite needing only medium or less privileges, meaning that a total of 19 extensions use interfaces that have broader privileges than they require. Figure 3 shows the detailed results. We summarize these results below:

- Three extensions, all download managers, require the ability to create new processes. (These are the only

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%)

Figure 3. 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.

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 network data) and/or web page access (e.g., manipulating a 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.

- 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).

Discussion. Although every Firefox extension runs with the user's full privileges, only three of the extensions we analyze actually require such a high level of privilege. The remaining 22 extensions exhibit a *privilege gap*: they run with more privileges than required. Moreover, none of the subject extensions require arbitrary file access and only 70% require network or web page access. The extension system can reduce the privileges of these extensions without impacting functionality.

Unfortunately, reducing the privileges of extensions in the Firefox extensions system is difficult because the Firefox extension API bundles many privileges into a single interface. This is evidenced by the 19 extensions that use excessively powerful interfaces: 16 use critical-rated interfaces and 3 use high-rated interfaces without needing that level of privilege. For example, most extensions use the preference service to store extension-local preferences. This service can also change browser-wide preferences and preferences belonging to other extensions.

We identified the file system interface as 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.

3.2 The Security Lattice

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

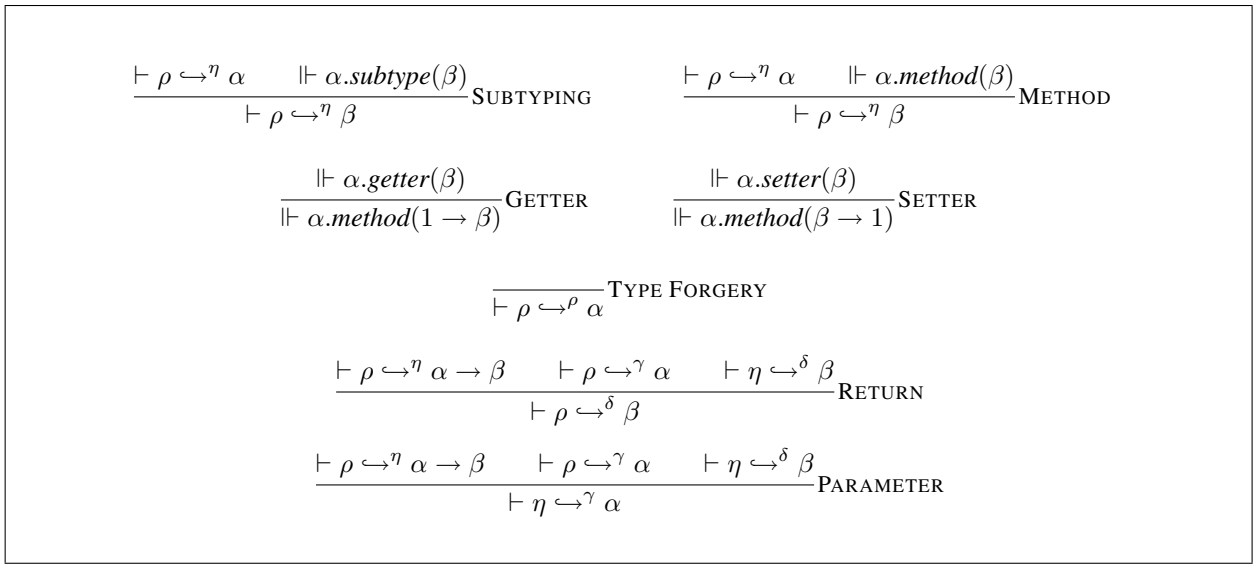


Figure 4. Inference rules for reachability in a type system with type forgery, such as the Firefox extension API.

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.

In Firefox, extensions and internal browser components use the same interfaces (known as XPCOM interfaces). These strictly typed interfaces are defined in a CORBA-like Interface Description Language (IDL). We analyzed the XPCOM interfaces from Firefox 3.5 by adding a Datalog back-end to the Firefox IDL compiler. By default, these interfaces are implemented internally by the browser. However, extensions can (and do) replace these implementations. For example, the SafeCache [21] browser extension replaces the HTTP cache. Regardless of the implementation of an XPCOM interface, the browser enforces the return and parameter types declared in the interface description.

We analyze the API for escalation points by organizing the XPCOM interfaces into a *security lattice*. We manually label the severity of 613 interfaces (of 1582 total), including all the interfaces used by the subject extensions. 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 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, type `foo` has a method that accepts type `bar` as a parameter. Type `bar` has a method `getFile` that returns a file type. We do not know whether an implementation of `foo` actually ever calls `bar.getFile`, but we know it is possible.

Deductive System. Our deductive system (see Figure 4) computes which additional interfaces a principal (the browser or an extension) can obtain from one interface. Along with the interface name, the rules track which principal implements each concrete instance of the interface. We write $\rho \hookrightarrow^{\eta} \alpha$ when principal ρ has a reference to an interface α implemented by principal η. The deduction rules then describe various ways a reference to one interface can lead to a reference to another interface. For example, if ρ possesses both a method of type $\alpha \to \beta$ implemented by η and an object with interface α implemented by γ, then ρ can give the α object to η by calling the method. Afterwards, η will have a reference to an object with interface α implemented by δ.

One subtle rule in the deductive system is the type forgery rule. This rule states that every principal can create an object that implements an arbitrary interface. This rule is appropriate for XPCOM (and, in fact, most CORBA-like component systems) 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. This technique is useful to attacks because an attacker can use a "forged" object to call a method the attacker could not call otherwise.

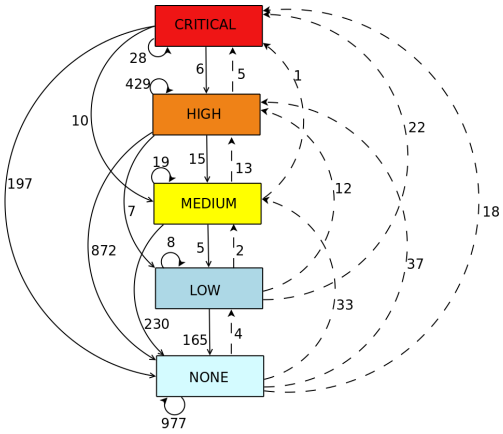


Figure 5. The Firefox extension API reachability graph, from our deductive system. Upward edges could lead to privilege escalation.

Reachability. We computed the security lattice for the Firefox extension interfaces by implementing our rules in Datalog. We add an edge from one interface to another if our deductive system computes that a reference to an object with the first interface implemented by one principal could lead to an object with the second interface implemented by the same principal. Notice that the type forgery rule permits us to reason about each interface individually instead of requiring us to build a lattice over sets of interfaces. Figure 5 summarizes the lattice by coalescing all the interfaces with the same security rating into a single vertex and contracting the unlabeled interfaces.

Of the 2920 edges in the lattice, 147 edges go "up" the lattice. 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. When designing a new extension system, we suggest not introducing escalation points into the security lattice.

4 Google Chrome Extension System

In this section, we describe and evaluate the security architecture of the Google Chrome extension system. We focus on the aspects of the design related to protecting users from benign-but-buggy extensions. The security model for the extension system is based on least privilege, privilege separation, and strong isolation.

4.1 Least Privilege

Instead of running with the user's full privileges, extensions run with a restricted set of privileges. The browser grants an extension access only to those privileges explicitly requested in the extension's *manifest*. By requiring extensions to declare their privileges at install time, an attacker who compromises an extension is limited to these privileges at runtime. For example, consider the manifest for the sample Gmail Checker extension [13]:

```
{
  "name": "Google Mail Checker",
  "description": "Displays the number of unread
     messages...",
  "version": "1.2",
  "background_page": "background.html",
  "permissions": [
    "tabs",
    "http://*.google.com/",
    "https://*.google.com/"
  ],
  "browser_action": {
      "default_title": ""
  },
  "icons": {
    "128": "icon_128.png"
  }
}
```

In the example, Gmail Checker needs access to subdomains of `google.com` and the `tabs` API. An extension can request a number of different privileges in its manifest:

- **Execute Arbitrary Code.** Although our case study suggests that a majority of extensions do not require the privilege to execute arbitrary code, some extensions do require this privilege. To request the privilege to execute arbitrary code, an extension lists a native binary in its manifest.

- **Web Site Access.** Extensions can also request the privilege to interact with web sites. Instead of receiving access to all web sites, extensions designate which web sites they would like to access by origin. For example, Gmail Checker requests access to subdomains of Google by listing `http://*.google.com` and `https://*.google.com` in its manifest. If the extension were later compromised, the attacker would not have the privilege to access `https://bank.com`.

- **API Access.** In addition to the usual web platform APIs, extensions can also request access to extension APIs, which are grouped according to functionality.

Behavior	Interface	Disparity?
User chooses a file (N)	User chooses a file (N)	No
Extension-specific files (N)	HTML5 storage (N)	No
Extension-specific SQLite (N)	HTML5 storage (N)	No
Specific-domain network access (M)	Restricted domains (M)	No
Page for display only (L)	Page for display only (L)	No
DOM of specific sites (M)	Restricted domains (M)	No
Highlighted text/images (L)	Gleam API (L)	No
Same-extension prefs (N)	HTML5 storage (N)	No
Language preferences (M)	Browser settings (M)	No

Figure 6. The proposed Google Chrome extension interfaces closely match the privileges required by the 25 extensions in our case study.

For example, the extension system contains an API group called `tabs` for interacting with the browser's tab strip (creating tabs, moving tabs, etc.). An extension is granted access to an API group only if that group appears in the extension's manifest.

The API groups closely match the privileges needed by the extensions we studied in Section 3.1. Figure 6 shows how those extensions could implement their functionality using the extension APIs. For example, an extension that requires access only to user-selected files can use the `<input type="file">` element, which grants the extension access to a file chosen by the user (and not to a mutable file handle).

Without additional encouragement, developers are likely to request the maximum possible privileges for their extensions, reducing the benefits of least privilege. To encourage developers to request the minimum required privileges, we alter the user experience for installing an extension from the Google-controlled extension gallery based on the maximum privilege level the extension requests. The most dangerous class of extensions (extensions with the privilege to execute arbitrary code) are not permitted in the gallery unless the developer signs a contract with Google. Another approach is to review extensions manually, as in the `addons.mozilla.org` gallery. In this approach, the manifests make it easier for reviewers to prioritize reviews of low-privilege extensions. This incentivizes developers to request fewer privileges to reduce review latency. Whether these incentives are sufficient to encourage least privilege will depend largely on whether developers can gain more exposure for their extensions by appearing in the gallery sooner or more prominently.

Extensions can also be installed from arbitrary web sites. This install experience is different from installing an extension from the gallery. When installing an extension from outside the gallery, the user experience is the same as the user experience for downloading and running a native executable. An attacker who can trick a user into installing a malicious extension this way can likely already trick the user into running an arbitrary executable, giving the attacker little additional leverage.

4.2 Privilege Separation

To make it more difficult for a malicious web site operator to usurp an extension's privileges, the extension platform forces developers to divide their extensions into multiple components: *content scripts*, the *extension core*, and a *native binary* (see Figure 1):

- **Content Scripts.** Content scripts let extensions interact directly with untrusted web content. If the manifest limits the extension's access to origins, the browser blocks the extension from injecting content scripts into unauthorized origins. Each content script, written in JavaScript, has direct access to the DOM of a single web page via the standard DOM APIs. Content scripts do not have access to the powerful extension APIs provided by the browser. The only other privilege granted to content scripts is the privilege to send JSON [2] messages to the extension core via a `postMessage`-like API.

- **Extension Core.** The extension core, written in HTML and JavaScript, controls the extension's user interface (e.g., browser actions, pop-ups) and has access to the extension APIs declared in the extension's manifest. The extension core contains the majority of the extension's privileges, but it is insulated from direct interaction with untrusted web content. To interact with untrusted content, the extension core can either (1) communicate with a content script or (2) issue an `XMLHttpRequest`. Both of these mechanisms require the extension author to take explicit action and restrict the interaction to plain data.

- **Native Binary.** Only native binaries can run arbitrary code or access arbitrary files. To gain these privileges, the extension developer must supply a native Netscape Plug-in API (NPAPI) binary. For example, on Windows such a binary consists of a dynamically linked library (DLL) with certain entry points. By default, the native binary can interact only with the extension core (e.g., not with content scripts). Furthermore, the interaction is typically limited to the interfaces defined when the native binary was compiled, but, of course, the native binary can re-compile itself because it can run arbitrary code. Similarly, the manifest lets developers expose their native binaries to web pages because there are no technical means for stopping an extension that can run arbitrary code from installing a regular browser plug-in.

Content scripts, which have the largest attack surface, do not have a direct channel to the component with critical privileges. By dividing the extension's privileges among three components, the extension system makes it harder for an attacker to exploit the user's machine. To run arbitrary code, the attacker first convinces the extension's content script to forward malicious input to the extension core. The attacker then convinces the extension core to forward the malicious input to the native binary (assuming one even exists). Finally, the attacker exploits a vulnerability in the native binary.

4.3 Isolation Mechanisms

The extension system uses three mechanisms to isolate extension components from each other and from web content. First, we leverage the same-origin web sandbox by running the extension core in a unique origin designated by a public key. Second, we run the extension core and the native binaries in their own processes. Finally, content scripts run in a separate JavaScript heap from untrusted web content.

Origin. In the web platform, the authority of a script is derived from its origin (in particular, the scheme, host, and port of the URL from which the browser obtained the script). However, extension scripts are not loaded from the network; extensions are stored in the user's file system. Consequently, extensions do not have an origin in the usual sense. We assign an "origin" to an extension by including a public key in the extension's URL as follows:

```
chrome-extension://
ilpnegfhimflflifcnmgpeihglhedbnn/
```

When loading an extension, the browser verifies that the extension package is "self-signed" by the public key in its URL. Placing the extension's public key in its URL frees the extension system from depending on a central naming authority (like a public-key infrastructure or DNS), reducing the attack surface of the platform and simplifying extension signing. By using this approach, we can reuse the web's same-origin machinery to isolate extensions from browser internals, web pages, and each other.

This approach to extension identity also makes updating extensions easy. If the browser encounters a newer extension package signed with the same public key, the browser can replace the installed version of the extension (unless the new manifest requests critical privileges and changes the install experience). When the browser reloads the extension, the updated version inhabits the same security context as the old version, analogous to re-visiting a web site. In particular, the updated extension still has access to its previous persistent state because `localStorage` is segregated by origin and its origin remains the same.

Process Isolation. Each component of the extension runs in a different process. The extension core and the native binary each receive dedicated processes. Content scripts run in the same process as their associated web pages. This process isolation has two benefits: it defends against browser errors and low-level exploits. Process isolation helps protect the extension core from browser implementation errors, such as cross-origin JavaScript capability leaks [12], because JavaScript objects cannot leak from one process to another. Process isolation also defends against low-level exploits in the browser. For example, if a malicious web site operator manages to corrupt the renderer process [11] (e.g., via a buffer overflow), the attacker will not be granted access to the extension APIs because the extension core resides in another process.

Isolated Worlds. We provide an additional layer of isolation between the content script and the untrusted web site's JavaScript environment by running the content script in an *isolated world*. Instead of accessing the underlying DOM data structures via the same JavaScript objects used by the page, each content script accesses the DOM with its "own" JavaScript objects. Content scripts and web pages therefore never exchange JavaScript pointers, making it more difficult for a malicious web page to confuse the content script (e.g., with a JavaScript rootkit [9]).

This design changes the normal one-to-one relation between DOM implementation objects and their JavaScript representations (see Figure 7) into a one-to-many relation (see Figure 8). For example, both the page and the content script have a global variable named `document`, but these variables refer to two distinct JavaScript objects. Consistency is still maintained: when either script calls a DOM method, such as `appendChild`, both objects are

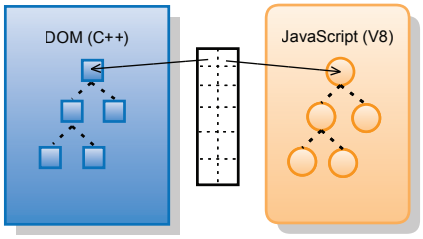


Figure 7. The normal one-to-one relation between DOM implementation objects and JavaScript representations.

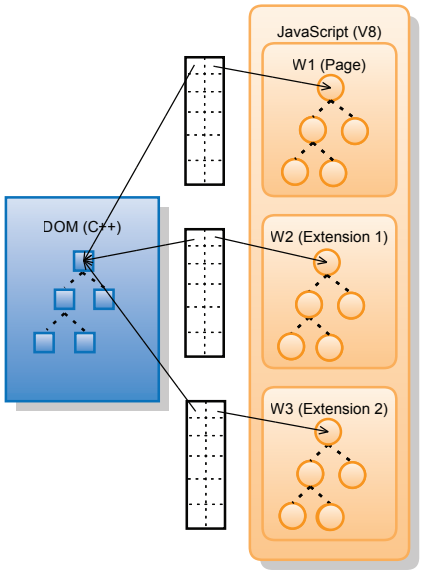


Figure 8. The one-to-many relation caused by running content scripts in isolated worlds.

updated to reflect the modified document tree. However, when a script modifies a non-standard DOM property, such as `document.foo`, the modification is not reflected in the other object. These rules help maintain the invariant that JavaScript objects (i.e., non-primitive values) are never transmitted between worlds.

The standard one-to-one relation is implemented using a hash table keyed by DOM implementation object (depicted as black rectangles in the figures). For isolated worlds, we create a hash table for each world and choose which hash table to use based on the currently executing world. When entering the JavaScript virtual machine (e.g., when invoking a callback function registered via `setTimeout`), the browser must start executing the function in the proper world. If the browser executes the function in the wrong world, we risk leaking a JavaScript pointer between worlds. To select the correct world with reasonable assurance, we cache a reference to the appropriate world on the function object itself at the time the callback is registered.

4.4 Performance

Separating extensions into components could potentially add overhead to operations that involve multiple components. For example, if a content script needs to use privileges held by the extension core, the content script needs to send a message to the core process instead of simply calling a function in its own address space. Similarly, DOM access from content scripts requires crossing from the extension's isolated world to the page's world, incurring an additional hash table lookup on some execution paths.

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 and back 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 [27] incurs this overhead once per page load to issue a cross-origin XMLHttpRequest, increasing the load time by an unnoticeable 0.8 ms.

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 but nothing to layout and rendering time.

5 Related Work

In addition to the Firefox extension system we analyze in this paper, Firefox has a second, experimental extension system: Jetpack [26]. Similar to the extension system we propose, 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 generativity [28].

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 [24] 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 3.2.

A number of papers [15, 34, 7, 16, 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.

Our work is also related to *mashups*, which are web pages that result from sophisticated communication and data sharing between multiple parties (e.g., plotting data from one source on a map from another source). In a sense, a browser is a mashup combining extension code and web content into a personalized browsing experience. Our design draws inspiration from MashupOS [31] and OMash [14], albeit taking into account subsequent attacks and design recommendations [10]. In addition, the isolated worlds heap-segregation mechanism is an outgrowth of the perspective expressed in [12]. Finally, placing the extension's public-key in the URL was suggested in [20] to remedy a vulnerability in Firefox's signed JAR mechanism.

Browser extensions are also analogous to kernel modules in operating systems. Buggy kernel modules have long been a major cause of failures and security vulnerabilities in operating systems. Nooks [29] and SafeDrive [35] employ memory access confinement to limit the privileges of kernel modules. Although the two problems are analogous, the techniques used are quite different.

6 Conclusion

Browser extensions are often not written by security experts, and many extensions contain security vulnerabilities. Every cross-site scripting vulnerability in a Firefox extension is an avenue for malicious web site operators to install malware onto the user's machine because Firefox extensions run with 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. However, reducing the privileges of existing Firefox extensions is difficult because many Firefox APIs are more powerful than required to implement extension features.

Although one could imagine restructuring the Firefox extension interface, we instead recommend building a new extension platform with security in mind. In our proposed system, extensions enumerate which privileges they desire at install-time and are limited to those privileges at runtime. If an extension does not include a native binary (which most do not require), then an attacker who compromises the extension will not gain the privilege to run arbitrary code.

In addition to least privilege, we separate privileges by dividing extensions into three components: content scripts, the extension core, and a native binary. Content scripts are exposed directly to web content but have few privileges. Native binaries are powerful but (by default) have no direct contact with web content. The three components interact via narrow interfaces, reducing the attack surface for the privileged components. We expect vulnerabilities to exist, of course, but we hope they will be harder to exploit than a single cross-site scripting hole.

Acknowledgments

We would like to thank Nick Baum, Erik Kay, Collin Jackson, Matt Perry, Dawn Song, David Wagner, and the Google Chrome Team. This work is partially supported by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170.

References

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

[2] JSON. `http://www.json.org`.

[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. http://www.skype.com.
[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, and W. Li. Attacks on JavaScript Mashup Communication. In *Proceedings of the Web 2.0 Security and Privacy 2009*.
[11] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, Google, 2008.
[12] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security Symposium*, 2009.
[13] A. Boodman and E. Kay. Google Mail Checker. http://code.google.com/chrome/extensions/samples.html.
[14] S. Crites, F. Hsu, and H. Chen. Omash: Enabling secure web mashups via object abstractions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108. ACM, 2008.
[15] 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.
[16] 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.
[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] C. Jackson and A. Barth. Beware of finer-grained origins. In *Web 2.0 Security and Privacy*, 2008.
[21] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th International World Wide Web Conference (WWW)*, May 2006.
[22] kkovash. How Many Firefox Users Customize Their Browser? *Blog of Metrics*, 2009.
[23] R. S. Liverani and N. Freeman. Abusing Firefox Extensions. Defcon17, July 2009.
[24] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. In *Journal in Computer Virology*, August 2008.
[25] Microsoft Developer Network. Introduction of the Protected Mode API. http://msdn.microsoft.com/en-us/library/ms537319(VS.85).aspx.
[26] Mozilla Labs. Jetpack. https://wiki.mozilla.org/Labs/Jetpack.
[27] D. Pupius. Fittr Flickr Extension for Chrome. http://code.google.com/p/fittr/.
[28] A. Raskin. Jetpack FAQ. http://www.azarask.in/blog/post/jetpack-faq/, 2009.
[29] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.
[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, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.
[32] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazell Web Browser. In *USENIX Security Symposium*, 2009.
[33] S. Willison. Understanding the Greasemonkey vulnerability. http://simonwillison.net/2005/Jul/20/vulnerability/.
[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.
[35] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniquesXFI. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

A Extension Survey

Our extension survey (Section 3.1) 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 JSConsole 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.