An Evaluation of the Google Chrome Extension Security Architecture

Nicholas Carlini, Adrienne Porter Felt, and David Wagner
University of California, Berkeley
nicholas.carlini@berkeley.edu, apf@cs.berkeley.edu, daw@cs.berkeley.edu

Abstract

Vulnerabilities in browser extensions put users at risk by providing a way for website and network attackers to gain access to users' private data and credentials. Extensions can also introduce vulnerabilities into the websites that they modify. In 2009, Google Chrome introduced a new extension platform with several features intended to prevent and mitigate extension vulnerabilities: strong isolation between websites and extensions, privilege separation within an extension, and an extension permission system. We performed a security review of 100 Chrome extensions and found 70 vulnerabilities across 40 extensions. Given these vulnerabilities, we evaluate how well each of the security mechanisms defends against extension vulnerabilities. We find that the mechanisms mostly succeed at preventing direct web attacks on extensions, but new security mechanisms are needed to protect users from network attacks on extensions, website metadata attacks on extensions, and vulnerabilities that extensions add to websites. We propose and evaluate additional defenses, and we conclude that banning HTTP scripts and inline scripts would prevent 47 of the 50 most severe vulnerabilities with only modest impact on developers.

1 Introduction

Browser extensions can introduce serious security vulnerabilities into users' browsers or the websites that extensions interact with [20, 32]. In 2009, Google Chrome introduced a new extension platform with several security mechanisms intended to prevent and mitigate extension vulnerabilities. Safari and Mozilla Firefox have since adopted some of these mechanisms for their own extension platforms. In this paper, we evaluate the security of the widely-deployed Google Chrome extension platform with the goal of understanding the practical successes and failures of its security mechanisms.

Most extensions are written by well-meaning developers who are not security experts. These non-expert developers need to build extensions that are robust to attacks originating from malicious websites and the network. Extensions can read and manipulate content from websites, make unfettered network requests, and access browser userdata like bookmarks and geolocation. In the hands of a web or network attacker, these privileges can be abused to collect users' private information and authentication credentials.

Google Chrome employs three mechanisms to prevent and mitigate extension vulnerabilities:

- *Privilege separation.* Chrome extensions adhere to a privilege-separated architecture [23]. Extensions are built from two types of components, which are isolated from each other: *content scripts* and *core extensions*. Content scripts interact with websites and execute with no privileges. Core extensions do not directly interact with websites and execute with the extension's full privileges.
- *Isolated worlds.* Content scripts can read and modify website content, but content scripts and websites have separate program heaps so that websites cannot access content scripts' functions or variables.
- *Permissions.* Each extension comes packaged with a list of permissions, which govern access to the browser APIs and web domains. If an extension has a core extension vulnerability, the attacker will only gain access to the permissions that the vulnerable extension already has.

In this work, we provide an empirical analysis of these security mechanisms, which together comprise a state-of-the-art least privilege system. We analyze 100 Chrome extensions, including the 50 most popular extensions, to determine whether Chrome's security mechanisms successfully prevent or mitigate extension vulnerabilities. We find that 40 extensions contain at least one type of vulnerability. Twenty-seven extensions contain core extension vulnerabilities, which give an attacker full control over the extension.

Based on this set of vulnerabilities, we evaluate the effectiveness of each of the three security mechanisms. Our primary findings are:

- The isolated worlds mechanism is highly successful at preventing content script vulnerabilities.
- The success of the isolated worlds mechanism renders privilege separation unnecessary. However, privilege separation would protect 62% of extensions if isolated worlds were to fail. In the remaining 38% of extensions, developers either intentionally or accidentally negate the benefits of privilege separation. This highlights that forcing developers to divide their software into components does not automatically achieve security on its own.
- Permissions significantly reduce the severity of half of the core extension vulnerabilities, which demonstrates that permissions are effective at mitigating vulnerabilities in practice. Additionally, dangerous permissions do not correlate with vulnerabilities: developers who write vulnerable extensions use permissions the same way as other developers.

Although these mechanisms reduce the rate and scope of several classes of attacks, a large number of high-privilege vulnerabilities remain.

We propose and evaluate four additional defenses. Our extension review demonstrates that many developers do not follow security best practices if they are optional, so we propose four mandatory bans on unsafe coding practices. We quantify the security benefits and functionality costs of these restrictions on extension behavior. Our evaluation shows that banning inline scripts and HTTP scripts would prevent 67% of the overall vulnerabilities and 94% of the most dangerous vulnerabilities at a relatively low cost for most extensions. In concurrent work, Google Chrome implemented Content Security Policy (CSP) for extensions to optionally restrict their own behavior. Motivated in part by our study [5], future versions of Chrome will use CSP to enforce some of the mandatory bans that we proposed and evaluated.

Contributions. We contribute the following:

- We establish the rate at which extensions contain different types of vulnerabilities, which should direct future extension security research efforts.
- We perform the first large-scale study of the effectiveness of privilege separation when developers who are not security experts are required to use it.
- Although it has been assumed that permissions mitigate vulnerabilities [12, 14, 10], we are the first to evaluate the extent to which this is true in practice.
- We propose and evaluate new defenses. This study partially motivated Chrome's adoption of a new mandatory security mechanism.

2 Extension Security Background

2.1 Threat Model

In this paper, we focus on non-malicious extensions that are vulnerable to external attacks. Most extensions are written by well-meaning developers who are not security experts. We do not consider malicious extensions; preventing malicious extensions requires completely different tactics, such as warnings, user education, security scans of the market, and feedback and rating systems. Benign-but-buggy extensions face two types of attacks:

- *Network attackers.* People who use insecure networks (e.g., public WiFi hotspots) may encounter network attackers [26, 21]. A network attacker's goal is to obtain personal information or credentials from a target user. To achieve this goal, a network attacker will read and alter HTTP traffic to mount man-in-the-middle attacks. (Assuming that TLS works as intended, a network attacker cannot compromise HTTPS traffic.) Consequently, data and scripts loaded over HTTP may be compromised.

  If an extension adds an *HTTP script* – a JavaScript file loaded over HTTP – to itself, a network attacker can run arbitrary JavaScript within the extension's context. If an extension adds an HTTP script to an HTTPS website, then the website will no longer benefit from the confidentiality, integrity, and authentication guarantees of HTTPS. Similarly, inserting HTTP data into an HTTPS website or extension can lead to vulnerabilities if the untrusted data is allowed to execute as code.

- *Web attackers.* Users may visit websites that host malicious content (e.g., advertisements or user comments). A website can launch a cross-site scripting attack on an extension if the extension treats the website's data or functions as trusted. The goal of a web attacker is to gain access to browser userdata (e.g., history) or violate website isolation (e.g., read another site's password).

Extensions are primarily written in JavaScript and HTML, and JavaScript provides several methods for converting strings to code, such as `eval` and `setTimeout`. If used improperly, these methods can introduce code injection vulnerabilities that compromise the extension. Data can also execute if it is written to a page as HTML instead of as text, e.g., through the use of `document.write` or `document.body.innerHTML`. Extension developers need to be careful to avoid passing unsanitized, untrusted data to these execution sinks.

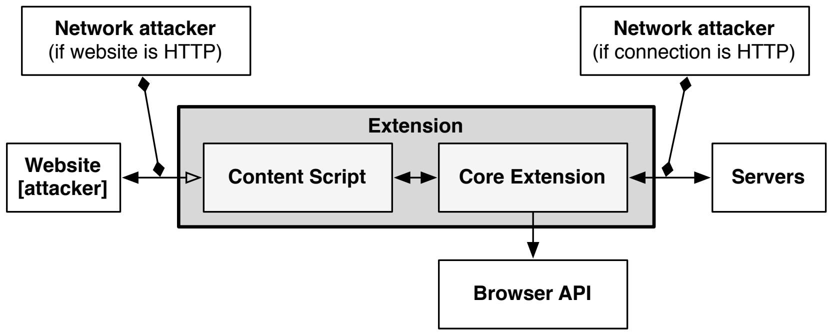

Figure 1: The architecture of a Google Chrome extension.

2.2 Chrome Extension Security Model

Many Firefox extensions have publicly suffered from vulnerabilities [20, 32]. To prevent this, the Google Chrome extension platform was designed to protect users from vulnerabilities in benign-but-buggy extensions [4]. It features three primary security mechanisms:

- *Privilege separation.* Every Chrome extension is composed of two types of components: zero or more *content scripts* and zero or one *core extension*. Content scripts read and modify websites as needed. The core extension implements features that do not directly involve websites, including browser UI elements, long-running background jobs, an options page, etc. Content scripts and core extensions run in separate processes, and they communicate by sending structured clones over an authenticated channel. Each website receives its own separate, isolated instance of a given content script. Core extensions can access Chrome's extension API, but content scripts cannot. Figure 1 illustrates the relationship between components in a Chrome extension.

  The purpose of this architecture is to shield the privileged part of an extension (i.e., the core extension) from attackers. Content scripts are at the highest risk of attack because they directly interact with websites, so they are low-privilege. The sheltered core extension is higher-privilege. As such, an attack that only compromises a content script does not pose a significant threat to the user unless the attack can be extended across the message-passing channel to the higher-privilege core extension.

  1.4% of extensions also include binary plugins in addition to content scripts and core extensions [12]. Binary plugins are native executables and are not protected by any of these security mechanisms. We do not discuss the security of binary plugins in this paper because they are infrequently used and must undergo a manual security review before they can be posted in the Chrome Web Store.

- *Isolated worlds.* The isolated worlds mechanism is intended to protect content scripts from web attackers. A content script can read or modify a website's DOM, but the content script and website have separate JavaScript heaps with their own DOM objects. Consequently, content scripts and websites never exchange pointers. This should make it more difficult for websites to tamper with content scripts.[1]

- *Permissions.* By default, extensions cannot use parts of the browser API that impact users' privacy or security. In order to gain access to these APIs, a developer must specify the desired permissions in a file that is packaged with the extension. For example, an extension must request the `bookmarks` permission to read or alter the user's bookmarks. Permissions also restrict extensions' use of cross-origin `XMLHttpRequests`; an extension needs to specify the domains that it wants to interact with. Only the core extension can use permissions. Content scripts cannot invoke browser APIs or make cross-origin XHRs.[2] A content script has only two privileges: it can access the website it is running on, and send messages to its core extension.

  Permissions are intended to mitigate core extension vulnerabilities.[3] An extension is limited to the permissions that its developer requested, so an attacker cannot request new permissions for a compromised extension. Consequently, the severity of a vulnerability in an extension is limited to the API calls and domains that the permissions allow.

[1]Although isolated worlds separates websites from content scripts, it not a form of privilege separation; privilege separation refers to techniques that isolate parts of the same application from each other.

[2]In newer versions of Chrome, content scripts can make cross-origin XHRs. However, this was not permitted at the time of our study.

[3]Extension permissions are shown to users during installation, so they may also have a role in helping users avoid malicious extensions; however, we focus on benign-but-buggy extensions in this work.

Google Chrome was the first browser to implement privilege separation, isolated worlds, and permissions for an extension system. These security mechanisms were intended to make Google Chrome extensions safer than Mozilla Firefox extensions or Internet Explorer browser helper objects [4]. Subsequently, Safari adopted an identical extension platform, and Mozilla Firefox's new Add-on SDK (Jetpack) privilege-separates extension modules. All of our study findings are directly applicable to Safari's extension platform, and the privilege separation evaluation likely translates to Firefox's Add-on SDK.

Contemporaneously with our extension review, the Google Chrome extension team began to implement a fourth security mechanism: *Content Security Policy (CSP)* for extensions. CSP is a client-side HTML policy system that allows website developers to restrict what types of scripts can run on a page [29]. It is intended to prevent cross-site scripting attacks by blocking the execution of scripts that have been inserted into pages. By default, CSP disables inline scripts: JavaScript will not run if it is in a link, between `<script>` tags, or in an event handler. The page's policy can specify a set of trusted servers, and only scripts from these servers will execute. Consequently, any attacker that were to gain control of a page would only be able to add code from the trusted servers (which should not lead to harm). CSP can also restrict the use of `eval`, XHR, and iframes. In Chrome, CSP applies to extensions' HTML pages [28].

3 Extension Security Review

We reviewed 100 Google Chrome extensions from the official directory. This set is comprised of the 50 most popular extensions and 50 randomly-selected extensions from June 2011.[4] Section 3.1 presents our extension review methodology. Our security review found that 40% of the extensions contain vulnerabilities, and Section 3.2 describes the vulnerabilities. Section 3.3 presents our observation that 31% of developers do not follow even the simplest security best practices. We notified most of the authors of vulnerable extensions (Section 3.4).

3.1 Methodology

We manually reviewed the 100 selected extensions, using a three-step security review process:

1. *Black-box testing.* We exercised each extension's user interface and monitored its network traffic to observe inputs and behavior. We looked for instances of network data being inserted into the DOM of a page. After observing an extension, we inserted malicious data into its network traffic (including the websites it interacts with) to test potential vulnerabilities.

2. *Source code analysis.* We examined extensions' source code to determine whether data from an untrusted source could flow to an execution sink. After manually reviewing the source code, we used `grep` to search for any additional sources or sinks that we might have missed. For sources, we looked for static and dynamic script insertion, `XMLHttpRequests`, cookies, bookmarks, and reading websites' DOMs. For sinks, we looked for uses of `eval`, `setTimeout`, `document.write`, `innerHTML`, etc. We then manually traced the call graph to find additional vulnerabilities.

3. *Holistic testing.* We matched extensions' source code to behaviors we identified during black-box testing. With our combined knowledge of an extension's source code, network traffic, and user interface, we attempted to identify any additional behavior that we had previously missed.

We then verified that all of the vulnerabilities could occur in practice by building attacks. Our goal was to find all vulnerabilities in every extension.

During our review, we looked for three types of vulnerabilities: vulnerabilities that extensions add to websites (e.g., HTTP scripts on HTTPS websites), vulnerabilities in content scripts, and vulnerabilities in core extensions. Some content script vulnerabilities may also be core extension vulnerabilities, depending on the extensions' architectures. Core extension vulnerabilities are the most severe because the core is the most privileged extension component. We do not report vulnerabilities if the potential attacker is a trusted website (e.g., `https://mail.google.com`) and the potentially malicious data is not user-generated; we do not believe that well-known websites are likely to launch web attacks.

After our manual review, we applied a well-known commercial static analysis tool to six extensions, with custom rules. However, our manual review identified significantly more vulnerabilities, and the static analysis tool did not find any additional vulnerabilities because of limitations in its ability to track strings. Prior research has similarly found that a manual review by experts uncovers more bugs than static analysis tools [30]. Our other alternative, VEX [3], was not built to handle several of the types of attacks that we reviewed. Consequently, we did not pursue static analysis further.

[4]We excluded four extensions because they included binary plugins; they were replaced with the next popular or random extensions. The directory's popularity metric is primarily based on the number of users.

Vulnerable Component	Web Attacker	Network Attacker
Core extension	5	50
Content script	3	1
Website	6	14

Table 1: 70 vulnerabilities, by location and threat model.

Vulnerable Component	Popular	Random	Total
Core extension	12	15	27
Content script	1	2	3
Website	11	6	17
Any	22	18	40

Table 2: The number of extensions with vulnerabilities, of 50 popular and 50 randomly-selected extensions.

3.2 Vulnerabilities

We found 70 vulnerabilities across 40 extensions. The appendix identifies the vulnerable extensions. Table 1 categorizes the vulnerabilities by the location of the vulnerability and the type of attacker that could exploit it. More of the vulnerabilities can be leveraged by a network attacker than by a web attacker, which reflects the fact that two of the Chrome extension platform's security measures were primarily designed to prevent web attacks. A bug may be vulnerable to both web and network attacks; we count it as a single vulnerability but list it in both categories in Table 1 for illustrative purposes.

The vulnerabilities are evenly distributed between popular and randomly-selected extensions. Table 2 shows the distribution. Although popular extensions are more likely to be professionally written, this does not result in a lower vulnerability rate in the set of popular extensions that we examined. We hypothesize that popular extensions have more complex communication with websites and servers, which increases their attack surface and neutralizes the security benefits of having been professionally developed. The most popular vulnerable extension had $768,154$ users in June 2011.

3.3 Developer Security Effort

Most extension developers are not security experts. However, there are two best practices that a security-conscious extension developer can follow without any expertise. First, developers can use HTTPS instead of HTTP when it is available, to prevent a network attacker from inserting data or code into an extension. Second, developers can use `innerText` instead of `innerHTML` when adding untrusted, non-HTML data to a page; `innerText` does not allow inline scripts to execute. We evaluate developers' use of these best practices in order to determine how security-conscious they are.

We find that 31 extensions contain at least one vulnerability that was caused by not following these two simple best practices. This demonstrates that a substantial fraction of developers do not make use of optional security mechanisms, even if the security mechanisms are very simple to understand and use. As such, we advocate mandatory security mechanisms that force developers to follow best security practices (Section 7).

3.4 Author Notification

We disclosed the extensions' vulnerabilities to all of the developers that we were able to contact. We found contact information for 80% of the vulnerable extensions.[5] Developers were contacted between June and September 2011, depending on when we completed each review. We sent developers follow-up e-mails if they did not respond to our initial vulnerability disclosure within a month.

Of the 32 developers that we contacted, 19 acknowledged and fixed the vulnerabilities in their extensions, and 7 acknowledged the vulnerabilities but have not completely fixed them as of February 7, 2012. Two of the un-patched extensions are official Google extensions. As requested, we provided guidance on how the security bugs could be fixed. None of the developers disputed the legitimacy of the vulnerabilities, although one developer argued that a vulnerability was too difficult to fix.

The appendix identifies the extensions that have been fixed. However, the "fixed" extensions are not necessarily secure despite our review. While checking on the status of vulnerabilities, we discovered that developers of several extensions have introduced new security vulnerabilities that were not present during our initial review. We do not discuss the new vulnerabilities in this paper.

4 Evaluation of Isolated Worlds

The isolated worlds mechanism is intended to protect content scripts from malicious websites, including otherwise-benign websites that have been altered by a network attacker. We evaluate whether the isolated worlds mechanism is sufficient to protect content scripts from websites. Our security review indicates that isolated worlds largely succeeds: only 3 of the 100 extensions have content script vulnerabilities, and only 2 of the vulnerabilities allow arbitrary code execution.

Developers face four main security challenges when writing extensions that interact with websites. We discuss whether and how well the isolated worlds mechanism helps prevent these vulnerability classes.

[5]For the remaining 20%, contact information was unavailable, the extension had been removed from the directory, or we were unable to contact the developer in a language spoken by the developer.

Data as HTML. One potential web development mistake is to insert untrusted data as HTML into a page, thereby allowing untrusted data to run as code. The isolated worlds mechanism mitigates this type of error in content scripts. When a content script inserts data as HTML into a website, any scripts in the data are executed within the website's isolated world instead of the extension's. This means that an extension can read data from a website's DOM, edit it, and then re-insert it into the page without introducing a content script vulnerability. Alternately, an extension can copy data from one website into another website. In this case, the extension will have introduced a vulnerability into the edited website, but the content script itself will be unaffected.

We expect that content scripts would exhibit a higher vulnerability rate if the isolated worlds mechanism did not mitigate data-as-HTML bugs. Six extensions' content scripts contained data-as-HTML errors that resulted in web site vulnerabilities, instead of the more-dangerous content script vulnerabilities. Furthermore, we found that 20 of the 50 (40%) core extension vulnerabilities are caused by inserting untrusted data into HTML; core extensions do not have the benefit of the isolated worlds mechanism to ameliorate this class of error. Since it is unlikely that developers exercise greater caution when writing content scripts than when writing core extensions, we conclude that the isolated worlds mechanism reduces the rate of content script vulnerabilities by mitigating data-as-HTML errors.

Eval. Developers can introduce vulnerabilities into their extensions by using `eval` to execute untrusted data. If an extension reads data from a website's DOM and `eval`s the data in a content script, the resulting code will run in the content script's isolated world. As such, the isolated worlds mechanism does not prevent or mitigate vulnerabilities due to the use of `eval` in a content script.

We find that relatively few developers use `eval`, possibly because its use has been responsible for well-known security problems in the past [8, 27]. Only 14 extensions use `eval` or equivalent constructs to convert strings to code in their content scripts, and most of those use it only once in a library function. However, we did find two content script vulnerabilities that arise because of an extension's use of `eval` in its content script. For example, the *Blank Canvas Script Handler* extension can be customized with supplemental scripts, which the extension downloads from a website and `eval`s in a content script. Although the developer is intentionally running data from the website as code, the integrity of the HTTP website that hosts the supplemental scripts could be compromised by a network attacker.

Click Injection. Extensions can register event handlers for DOM elements on websites. For example, an extension might register a handler for a button's `onClick` event. However, extensions cannot differentiate between events that are triggered by the user and events that are generated by a malicious web site. A website can launch a click injection attack by invoking an extension's event handler, thereby tricking the extension into performing an action that was not requested by the user. Although this attack does not allow the attacker to run arbitrary code in the vulnerable content script, it does allow the website to control the content script's behavior.

The isolated worlds mechanism does not prevent or mitigate click injection attacks at all. However, the attack surface is small because relatively few extensions register event handlers for websites' DOM elements. Of the 17 extensions that register event handlers, most are for simple buttons that toggle UI state. We observed only one click injection vulnerability, in the *Google Voice* extension. The extension changes phone numbers on websites into links. When a user clicks a phone number link, Google Voice inserts a confirmation dialog onto the DOM of the website to ensure that the user wants to place a phone call. Google Voice will place the call following the user's confirmation. However, a malicious website could fire the extension's event handlers on the link and confirmation dialog, thereby placing a phone call from the user's Google Voice account without user consent.

Prototypes and Capabilities. In the past, many vulnerabilities due to prototype poisoning and capability leaks have been observed in bookmarklets and Firefox extensions [20, 32, 2]. The isolated worlds mechanism provides heap separation, which prevents both of these types of attacks. Regardless of developer behavior, these attacks are not possible in Chrome extensions as long as the isolation mechanism works correctly.

Based on our security review, the isolated worlds mechanism is highly effective at shielding content scripts from malicious websites. It mitigates data-as-HTML errors, which we found were very common in the Chrome extensions that we reviewed. Heap separation also prevents prototype poisoning and capability leaks, which are common errors in bookmarklets and Firefox extensions. Although the isolated worlds mechanism does not prevent click injection or `eval`-based attacks, we find that developers rarely make these mistakes. We acknowledge that our manual review could have missed some content script vulnerabilities. However, we find it unlikely that we could have missed many, given our success at finding the same types of vulnerabilities in core extensions. We therefore conclude that the isolated worlds mechanism is effective, and other extension platforms should implement it if they have not yet done so.

5 Evaluation of Privilege Separation

Privilege separation is intended to shield the privileged core extension from attacks. The isolated worlds mechanism serves as the first line of defense against malicious websites, and privilege separation is supposed to protect the core extension when isolated worlds fails. We evaluate the effectiveness of extension privilege separation and find that, although it is unneeded, it would be partially successful at accomplishing its purpose if the isolated worlds mechanism were to fail.

5.1 Cross-Component Vulnerabilities

Some developers give content scripts access to core extension permissions, which removes the defense-in-depth benefits of privilege separation. We evaluate the impact of developer behavior on the effectiveness of extension privilege separation.

Vulnerable Content Scripts. The purpose of privilege separation is to limit the impact of content script vulnerabilities. Even if a content script is vulnerable, privilege separation should prevent an attacker from executing code with the extension's permissions. We identified two extensions with content script vulnerabilities that permit arbitrary code execution; these two extensions could benefit from privilege separation.

Despite privilege separation, both of the vulnerabilities yield access to some core extension privileges. The vulnerable content scripts can send messages to their respective core extensions, requesting that the core extensions exercise their privileges. In both extensions, the core extension makes arbitrary XHRs on behalf of the content script and returns the result to the content script. This means that the two vulnerable content scripts could trigger arbitrary HTTP XHRs even though content scripts should not have access to a cross-origin `XMLHttpRequest` object. These vulnerable extensions represent a partial success for privilege separation because the attacker cannot gain full privileges, but also a partial failure because the attacker can gain the ability to make cross-origin XHRs.

Hypothetical Vulnerabilities. Due to the success of the isolated worlds mechanism, our set of vulnerabilities only includes two extensions that need privilege separation as a second line of defense. To expand the scope of our evaluation of privilege separation, we explore a hypothetical scenario: if the currently-secure extensions' content scripts had vulnerabilities, would privilege separation mitigate these vulnerabilities?

Of the 98 extensions that do not have content script vulnerabilities, 61 have content scripts. We reviewed the message passing boundary between these content scripts and their core extensions. We determined that 38% of content scripts can leverage communication with their core extensions to abuse some core extension privileges: 4 extensions' content scripts can use all of their cores' permissions, and 19 can use some of their cores' permissions. Table 3 shows which permissions attackers would be able to obtain via messages if they were able to compromise the content scripts. This demonstrates that privilege separation could be a relatively effective layer of defense, if needed: we can expect that privilege separation would be effective at limiting the damage of a content script vulnerability 62% of the time.

Permissions	**Number of Scripts**
All of the extension's permissions	4
Partial: Cross-origin XHRs[2]	9
Partial: Tab control	5
Partial: Other	5

Table 3: 61 extensions have content scripts that do not have code injection vulnerabilities. If an attacker were hypothetically able to compromise the content scripts, these are the permissions that the attacker could gain access to via the message-passing channel with the cores.

Example. The *AdBlock* extension allows its content script to execute a set of pre-defined functions in the core extension. To do this, the content script sends a message to the core extension. A string in the message is used to index the `window` object, allowing the content script to select a pre-defined function to run. Unfortunately, this also permits arbitrary code execution because the `window` object provides access to `eval`. As such, a compromised content script would have unfettered access to the core extension's permissions.

Example. A bug in the *Web Developer* extension unintentionally grants its content script full privileges. Its content script can post small notices to the popup page, which is part of the core extension. The notices are inserted using `innerHTML`. The notices are supposed to be text, but a compromised content script could send a notice with an inline script that would execute in the popup page with full core extension permissions.

5.2 Web Site Metadata Vulnerabilities

The Chrome extension platform applies privilege separation with the expectation that malicious website data will first enter an extension via a vulnerable content script. However, it is possible for a website to attack a core extension without crossing the privilege separation boundary. Website-controlled metadata such as titles and URLs can be accessed by the core extension through browser

Type	Vulnerabilities
Website content	2
Website metadata	5
HTTP XHR	16
HTTP script	28
Total	50

Table 4: The types of core extension vulnerabilities.

managers (e.g., the history, bookmark, and tab managers). This metadata may include inline scripts, and mishandled metadata can lead to a core extension vulnerability. Website metadata does not flow through content scripts, so privilege separation does not impede it. We identified five vulnerabilities from metadata that would allow an attacker to circumvent privilege separation.

Example. The *Speeddial* extension replicates Chrome's built-in list of recently closed pages. Speeddial keeps track of the tabs opened using the tabs manager and does not sanitize the titles of these pages before adding them to the HTML of one of its core extension pages. If a title were to contain an inline script, it would execute with the core extension's permissions.

5.3 Direct Network Attacks

Privilege separation is intended to protect the core extension from web attackers and HTTP websites that have been compromised by network attackers. However, the core extension may also be subject to direct network attacks. Nothing separates a core extension from code in HTTP scripts or data in HTTP `XMLHttpRequests`. HTTP scripts in the core extension give a network attacker the ability to execute code with the extension's full permissions, and HTTP XHRs cause vulnerabilities when extensions allow the HTTP data to execute.

Direct network attacks comprise the largest class of core extension vulnerabilities, as Table 4 illustrates. Of the 50 core extension vulnerabilities, 44 vulnerabilities (88%) stem from HTTP scripts or HTTP `XMLHttpRequests`, as opposed to website data. For example, many extensions put the HTTP version of the Google Analytics script in the core extension to track which of the extensions' features are used.

Example. Google Dictionary allows a user to look up definitions of words by double clicking on a word. The desired definition is fetched by making a HTTP request to `google.com` servers. The response is inserted into one of the core extension's pages using `innerHTML`. A network attacker could modify the response to contain malicious inline scripts, which would then execute as part of the privileged core extension page.

5.4 Implications

The isolated worlds mechanism is so effective at protecting content scripts from websites that privilege separation is rarely needed. As such, privilege separation is used to address a threat that almost does not exist, at the cost of increasing the complexity and performance overhead of extensions. (Privilege separation requires an extra process for each extension, and communication between content scripts and core extensions is IPC.) We find that network attackers are the real threat to core extension security, but privilege separation does not mitigate or prevent these attacks. This shows that although privilege separation can be a powerful security mechanism [23], its placement within an overall system is an important determining factor of its usefulness.

Our study also has implications for the use of privilege separation in other contexts. All Chrome extension developers are required to privilege separate their extensions, which allows us to evaluate how well developers who are not security experts use privilege separation. We find that privilege separation would be fairly effective at preventing web attacks in the absence of isolated worlds: privilege separation would fully protect 62% of core extensions. However, in more than a third of extensions, developers created message passing channels that allow low-privilege code to exploit high-privilege code. This demonstrates that forcing developers to privilege separate their software will improve security in most cases, but a significant fraction of developers will accidentally or intentionally negate the benefits of privilege separation. Mandatory privilege separation could be a valuable line of defense for another platform, but it should not be relied on as the only security mechanism; it should be coupled with other lines of defense.

6 Evaluation of the Permission System

The Chrome permission system is intended to reduce the severity of core extension vulnerabilities. If a website or network attacker were to successfully inject malicious code into a core extension, the severity of the attack would be limited by the extension's permissions. However, permissions will not mitigate vulnerabilities in extensions that request many dangerous permissions. We evaluate the extent to which permissions mitigate the core extension vulnerabilities that we found.

Table 5 lists the permissions that the vulnerable extensions request. Ideally, each permission should be requested infrequently. We find that 70% of vulnerable extensions request the `tabs` permission; an attacker with access to the `tabs` API can collect a user's browsing history or redirect pages that a user views. Fewer than half of extensions request each of the other permissions.

Permissions	**Times Requested**	**Percentage**
tabs (browsing history)	19	70%
all HTTP domains	12	44%
all HTTPS domains	12	44%
specific domains	10	37%
notifications	5	19%
bookmarks	4	15%
no permissions	4	15%
cookies	3	11%
geolocation	1	4%
context menus	1	4%
unlimited storage	1	4%

Table 5: The permissions that are requested by the 27 extensions with core extension vulnerabilities.

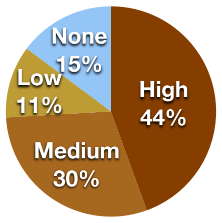

Figure 2: The 27 extensions with core vulnerabilities, categorized by the severity of their worst vulnerabilities.

To summarize the impact of permissions on extension vulnerabilities, we categorized all of the vulnerabilities by attack severity. We based our categorization on the Firefox Security Severity Ratings [1], which has been previously used to classify extension privileges [4]:

- *Critical*: Leaks the permission to run arbitrary code on the user's system
- *High*: Leaks permissions for the DOM of all HTTP(S) websites
- *Medium*: Leaks permissions for private user data (e.g., history) or the DOM of specific websites that contain financial or important personal data (e.g., `https://*.google.com/*`)
- *Low*: Leaks permissions for the DOM of specific websites that do not contain sensitive data (e.g., `http://*.espncricinfo.com`) or permissions that can be used to annoy the user (e.g., fill up storage or make notifications)
- *None*: Does not leak any permissions

We did not find any critically-vulnerable extensions. This is a consequence of our extension selection methodology: we did not review any extensions with binary plugins, which are needed to obtain critical privileges.

Figure 2 categorizes the 27 vulnerable extensions by their most severe vulnerabilities. In the absence of a permission system, all of the vulnerabilities would give an attacker access to all of the browser's privileges (i.e., critical privileges). With the permission system, less than half of the vulnerable extensions yield access to high-severity permissions. As such, our study demonstrates that the permission system successfully limits the severity of most vulnerabilities.

We hypothesized that permissions would positively correlate with vulnerabilities. Past work has shown that many extensions are over-permissioned [12, 14], and we thought that developers who are unwilling to follow security best practices (e.g., use HTTPS) would be unwilling to take the time to specify the correct set of permissions. This would result in vulnerable extensions requesting dangerous permissions at a higher rate. However, we do not find any evidence of a positive correlation between vulnerabilities and permissions. The 27 extensions with core vulnerabilities requested permissions at a lower rate than the other 73 extensions, although the difference was not statistically significant. Our results show that developers of vulnerable extensions can use permissions well enough to reduce the privileges of their insecure extensions, even though they lack the expertise or motivation required to secure their extensions.

Permissions are not only used by the Google Chrome extension system. Android implements a similar permission system, and future HTML5 device APIs will likely be guarded with permissions. Although it has been assumed that permissions mitigate vulnerabilities [10, 12, 14], our study is the first to evaluate whether this is true for real-world vulnerabilities or measure quantitatively how much it helps mitigate these vulnerabilities in practice. Our findings indicate that permissions can have a significant positive impact on system security and are worth including in a new platform as a second line of defense against attacks. However, they are not effective enough to be relied on as the only defense mechanism.

7 Defenses

Despite Google Chrome's security architecture, our security review identified 70 vulnerabilities in 40 extensions. Based on the nature of these vulnerabilities, we propose and evaluate four additional defenses. The defenses are bans on unsafe coding practices that lead to vulnerabilities. We advocate mandatory bans on unsafe coding practices because many developers do not follow security best practices when they are optional (Section 3.3). We quantify the security benefits and compatibility costs of each of these defenses to determine whether they should be adopted. Our main finding is that a combination of banning HTTP scripts and banning inline scripts would prevent 94% of the core extension vulnerabilities, with only a small amount of developer effort to maintain full functionality in most cases.

In concurrent work, Google Chrome implemented Content Security Policy (CSP) for extensions. CSP can be used to enforce all four of these defenses. Initially, the use of CSP was wholly optional for developers. As of Chrome 18, extensions that take advantage of new features will be subject to a mandatory policy; this change was partially motivated by our study [5].

7.1 Banning HTTP Scripts

Scripts fetched over HTTP are responsible for half of the vulnerabilities that we found. All of these vulnerabilities could be prevented by not allowing extensions to add HTTP scripts to their core extensions [15] or to HTTPS websites. Extensions that currently violate this restriction could be easily modified to comply by packaging the script with the extension or using a HTTPS URL. Only vulnerable extensions would be affected by the ban because any extension that uses HTTP scripts will be vulnerable to man-in-the-middle attacks.

Core Extension Vulnerabilities. Banning HTTP scripts from core extensions would remove 28 core extension vulnerabilities (56% of the total core extension vulnerabilities) from 15 extensions. These 15 extensions load HTTP scripts from 13 domains, 10 of which already offer the same script over HTTPS. The remaining 3 scripts are static files that could be downloaded once and packaged with the extensions.

Website Vulnerabilities. Preventing extensions from adding HTTP scripts to HTTPS websites would remove 8 website vulnerabilities from 8 extensions (46% of the total website vulnerabilities). These vulnerabilities allow a network attacker to circumvent the protection that HTTPS provides for websites. The extensions load HTTP scripts from 7 domains, 3 of which offer an HTTPS option. The remaining 4 scripts are static scripts that could be packaged with the extensions.

7.2 Banning Inline Scripts

Untrusted data should not be added to pages as HTML because it can contain inline scripts (e.g., inline event handlers, links with embedded JavaScript, and `<script>` tags). For example, untrusted data could contain an image tag with an inline event handler: `<img onload="doEvil();" ...>`. We find that 40% of the core extension vulnerabilities are caused by adding untrusted data to pages as HTML. These vulnerabilities could be prevented by not allowing any inline scripts to execute: the untrusted data will still be present as HTML, but it would be static. JavaScript will only run on a page if it is in a separate `.js` file that is stored locally or loaded from a trusted server that the developer has whitelisted.

Banning inline scripts from extension HTML would eliminate 20 vulnerabilities from 15 extensions. All of these vulnerabilities are core extension vulnerabilities. Content script vulnerabilities cannot be caused by inline scripts, and we cannot prevent extensions from adding inline scripts to HTTPS websites because existing enforcement mechanisms cannot differentiate between a website's own inline scripts and extension-added scripts.

However, banning inline scripts has costs. Developers use legitimate inline scripts for several reasons, such as to define event handlers. In order to maintain functionality despite the ban, all extensions would need to delete their inline scripts from HTML and move them to separate `.js` files. Inline event handlers (e.g., `onclick`) cannot simply be copied and pasted; they need to be rewritten as programmatically using the DOM API.

We reviewed the 100 extensions to determine what changes would be needed to comply with a ban on inline scripts. Applying this ban breaks 79% of the extensions. However, all of the extensions could be retrofitted to work without inline scripts without significant changes to the extension. Most of the compatibility costs pertain to moving the extensions' inline event handlers. The extensions contain an average of 7 event handlers, with a maximum of 98 and a minimum of 0 event handlers.

7.3 Banning Eval

Dynamic code generation converts strings to code, and its use can lead to vulnerabilities if the strings are untrusted data. Disallowing the use of dynamic code generation (e.g., `eval` and `setTimeout`) would eliminate three vulnerabilities: one core extension vulnerability, and two vulnerabilities that are both content script and core extension vulnerabilities.

We reviewed the 100 extensions and find that dynamic code generation is primarily used in three ways:

1. Developers sometimes pass static strings to `setTimeout` instead of functions. This coding pattern cannot be exploited. It would be easy to alter instances of this coding pattern to comply with a ban on dynamic code generation; the strings simply need to be replaced with equivalent functions.
2. Some developers use `eval` on data instead of `JSON.parse`. We identified one vulnerability that was caused by this practice. In the absence of dynamic code generation, developers could simply use the recommended `JSON.parse`.
3. Two extensions use `eval` to run user-specified scripts that extend the extensions. In both cases, their error is that they fetch the extra scripts over HTTP instead of HTTPS. For these two extensions, a ban on `eval` would prevent the vulnerabilities but irreparably break core features of the extensions.

Restriction	Security Benefit	Broken, But Fixable	Broken And Unfixable
No HTTP scripts in core	15%	15%	0%
No HTTP scripts on HTTPS websites	8%	8%	0%
No inline scripts	15%	79%	0%
No `eval`	3%	30%	2%
No HTTP XHRs	17%	29%	14%
All of the above	35%	86%	16%
No HTTP scripts and no inline scripts	32%	80%	0%
Chrome 18 policy	27%	85%	2%

Table 6: The percentage of the 100 extensions that would be affected by the restrictions. The "Security Benefit" column shows the number of extensions that would be fixed by the corresponding restriction.

Richards et al. present additional uses of `eval` in a large-scale study of web applications [24].

We find that 32 extensions would be broken by a ban on dynamic code generation. Most instances can easily be replaced, but 2 extensions would be permanently broken. Overall, a ban on `eval` would fix three vulnerabilities at the cost of fundamentally breaking two extensions.

7.4 Banning HTTP XHR

Network attacks can occur if untrusted data from an HTTP `XMLHttpRequest` is allowed to flow to a JavaScript execution sink. 30% of the 70 vulnerabilities are caused by allowing data from HTTP XHRs to execute. One potential defense is to disallow HTTP XHRs; all XHRs would have to use HTTPS. This ban would remove vulnerabilities from 17 extensions.

However, banning HTTP XHRs would have a high compatibility cost. The only way to comply with an HTTPS-only XHR policy is to ensure that the server supports HTTPS; unlike scripts, remote data cannot be packaged with extensions. Developers who do not control the servers that their extensions interact with will not be able to adapt their extensions. Extension developers who also control the domains may be able to add support for HTTPS, although this can be a prohibitively expensive and difficult process for a novice developer.

We reviewed the 100 extensions and found that 29% currently make HTTP XHRs. All of these would need to be changed to use HTTPS XHRs. However, not all of the domains offer HTTPS. Ten extensions request data from at least one HTTP-only domain. Additionally, four extensions make HTTP XHRs to an unlimited number of domains based on URLs provided by the user; these extensions would have permanently reduced functionality. For example, *Web Developer* lets users check whether a website is valid HTML. It fetches the user-specified website with an XHR and then validates it. Under a ban on HTTP XHRs, the extension would not be able to validate HTTP websites. In total, 14% of extensions would have some functionality permanently disabled by the ban.

7.5 Recommendations

Table 6 summarizes the benefits and costs of the defenses. If the set of 100 extensions were subject to all four bans, only 5 vulnerable extensions would remain, and 16 extensions would be permanently broken. Based on this evaluation, we conclude:

- We strongly recommend banning HTTP scripts and inline scripts; together, they would prevent 47 of the 50 core extension vulnerabilities, and no extension would be permanently broken. The developer effort required to comply with these restrictions is modest.
- Banning `eval` would have a neutral effect: neither the security benefits nor the costs are large. Consequently, we advise against banning `eval`.
- We do not recommend banning HTTP XHRs, given the number of extensions that would be permanently disabled by the ban. Of the 20 vulnerabilities that the ban on HTTP XHRs would prevent, 70% could also be prevented by banning inline scripts. We do not feel that the ban on HTTP XHRs adds enough value to justify breaking 14% of extensions.

Starting with Chrome 18, extensions will be subject to a CSP that enforces some of these bans [13]. Our study partially motivated their decision to adopt the bans [5], although the policy that they adopted is slightly stricter than our recommendations. The mandatory policy in Chrome 18 will ban HTTP scripts in core extensions, inline scripts, and dynamic code generation. Due to technical limitations, they are not adopting a ban on adding HTTP scripts to HTTPS websites. The policy will remove all of the core extension vulnerabilities that we found. The only extensions that the policy will permanently break are the two extensions that rely on `eval`.

8 Related Work

Extension vulnerabilities. To our knowledge, our work is the first to evaluate the efficacy of the Google Chrome extension platform, which is widely deployed and explicitly designed to prevent and mitigate extension vulnerabilities. Vulnerabilities in other extension platforms, such as Firefox, have been investigated by previous researchers [20, 3]. We found that 40% of Google Chrome extensions are vulnerable, which is in contrast to a previous study that found that 0.24% of Firefox extensions contain vulnerabilities [3]. This does not necessarily imply that Firefox extensions are more secure; rather, our scopes and methodologies differ. Unlike the previous study, we considered network attackers as well as web attackers. We find that 5% of Google Chrome extensions have the types of web vulnerabilities that the previous study covered. The remaining discrepancy could be accounted for by our methodology: we employed expert human reviewers whereas previous work relied on a static analysis tool that does not model dynamic code evaluation, data flow through the extension API, data flow through DOM APIs, or click injection attacks.

Privilege separation. Privilege separation is a fundamental software engineering principle proposed by Saltzer and Schroeder [25]. Numerous works have applied this concept to security, such as OpenSSH [23] and qmail [6]. Recently, researchers have built several tools and frameworks to help developers privilege separate their applications [7, 11, 17, 18, 22]. Studies have established that privilege separation has value in software projects that employ security experts (e.g., browsers [9]). However, we focus on the effectiveness of privilege separation in applications that are not written by security experts.

In concurrent and independent work, Karim et al. studied the effectiveness of privilege separation in Mozilla Jetpack extensions [16]. Like Chrome extensions, Jetpack extensions are split into multiple components with different permissions. They statically analyzed Jetpack extensions and found several capability leaks in modules. Although none of these capability leaks are tied to known vulnerabilities, the capability leaks demonstrate that developers can make errors in a privilege-separated environment. Their findings support the results of our analysis of privilege separation in Chrome extensions.

Extension permissions. Previous researchers have established that permissions can reduce the privileges of extensions without negatively impacting the extensions' functionality [4, 12]. Studies have also shown that some extensions request unnecessary permissions, which is undesirable because it unnecessarily increases the scope of a potential vulnerability [12, 14]. All of these past studies asserted that the correct usage of permissions could reduce the severity of attacks on extensions. However, they did not study whether this is true in practice or quantify the benefit for deployed applications. To our knowledge, we are the first to test whether permissions mitigate vulnerabilities in practice.

CSP compatibility. Adapting websites to work with CSP can be a challenging undertaking for developers, primarily due to the complexities associated with server-side templating languages [31]. However, extensions do not use templating languages. Consequently, applying CSP to extensions is easier than applying it to websites in most cases. We expect that our CSP compatibility findings for extensions will translate to packaged JavaScript and packaged web applications.

Malicious extensions. Extension platforms can be used to build malware (e.g., FFsniFF and Infostealer.Snifula [33]). Mozilla and Google employ several strategies to prevent malicious extensions, such as domain verification, fees, and security reviews. Liu et al. propose changes to Chrome to make malware easier to identify [19]. Research on extension malware is orthogonal to our work, which focuses on external attackers that leverage vulnerabilities in benign-but-buggy extensions.

9 Conclusion

We performed a security review on a set of 100 Google Chrome extensions, including the 50 most popular, and found that 40% have at least one vulnerability. Based on this set of vulnerabilities, we evaluated the effectiveness of Chrome's three extension security mechanisms: isolated worlds, privilege separation, and permissions.

We found that the isolated worlds mechanism is highly effective because it prevents common developer errors (i.e., data-as-HTML errors). The effectiveness of isolated worlds means that privilege separation is rarely needed. Privilege separation's infrequent usefulness may not justify the complexity and communication overhead that it adds to extensions. However, our study shows that privilege separation would improve security in the absence of isolated worlds. We also found that permissions can have a significant positive impact on system security; developers of vulnerable extensions can use permissions well enough to reduce the scope of their vulnerabilities.

Although we demonstrated that privilege separation and permissions can mitigate vulnerabilities, developers do not always use them optimally. We identified several instances in which developers accidentally negated the benefits of privilege separation or intentionally circumvented the privilege separation boundary to implement features. Similarly, extensions sometimes ask for more permissions than they need [12]. Automated tools for privilege separation and permission assignment could help developers better use these security mechanisms, thereby rendering them even more effective.

Despite the successes of these security mechanisms, extensions are widely vulnerable. The vulnerabilities occur because the system was designed to address only one threat: websites that attack extensions through direct interaction. There are no security mechanisms to prevent direct network attacks on core extensions, website metadata attacks, or attacks on websites that have been altered by extensions. This finding should serve as a reminder that multiple threats should be considered when initially designing a system. We propose to prevent these additional threats by banning insecure coding practices that commonly lead to vulnerabilities; bans on HTTP scripts and inline scripts would remove 94% of the most serious attacks with a tractable developer cost.

Acknowledgements

We would like to thank Prateek Saxena and Adam Barth for their insightful comments. This material is based upon work supported by Facebook and National Science Foundation Graduate Research Fellowships. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of Facebook or the National Science Foundation. This work is also partially supported by National Science Foundation grant CCF-0424422, a gift from Google, and the Intel Science and Technology Center for Secure Computing.

References

[1] L. Adamski. Security severity ratings. `https://wiki.mozilla.org/Security_Severity_Ratings`.

[2] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript Environments. In *Web 2.0 Security and Privacy (W2SP)*, 2009.

[3] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting Browser Extensions For Security Vulnerabilities. In *USENIX Security*, 2010.

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[5] Adam Barth. More secure extensions, by default. `http://blog.chromium.org/2012/02/more-secure-extensions-by-default.html`, February 2012.

[6] D. J. Bernstein. The qmail security guarantee. `http://cr.yp.to/qmail/guarantee.html`.

[7] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[8] B. Chess, Y. T. O'Neil, and J. West. JavaScript Hijacking. Technical report, Fortify, 2007.

[9] J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith, and C. Valasek. Browser Security Comparison: A Quantitative Approach. Technical report, Accuvant Labs, 2011.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *ACM Conference on Computer and Communication Security (CCS)*, 2011.

[11] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner. Diesel: Applying Privilege Separation to Database Access. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2011.

[12] A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. In *USENIX Conference on Web Application Development (WebApps)*, 2011.

[13] Google Chrome Extensions. Content Security Policy (CSP). `http://code.google.com/chrome/extensions/trunk/contentSecurityPolicy.html`.

[14] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, 2011.

[15] C. Jackson. Block chrome-extension:// pages from importing script over non-https connections. `http://code.google.com/p/chromium/issues/detail?id=29112`.

[16] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung chiech Shan. An Analysis of the Mozilla Jetpack Extension Framework. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[17] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-grained privilege separation for web applications. In *International Conference on World Wide Web (WWW)*, 2010.

[18] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make Least Privilege a Right (Not a Privilege). In *Conference on Hot Topics in Operating Systems*, 2005.

[19] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[20] R. S. Liverani and N. Freeman. Abusing Firefox Extensions. Defcon17.

[21] A. Mikhailovsky, K. V. Gavrilenko, and A. Vladimirov. The Frame of Deception: Wireless Man-in-the-Middle Attacks and Rogue Access Points Deployment. `http://www.informit.com/articles/article.aspx?p=353735&seqNum=7`, 2004.

[22] D. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *European Workshop on System Security (EUROSEC)*, 2008.

[23] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *USENIX Security Symposium*, 2003.

[24] G. Richards, C.Hammer, B. Burg, and J. Vivek. The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications. In *European Conference on Object-Oriented Programming*, 2012.

[25] J. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *IEEE 63*, 1975.

[26] R. Saltzman and A. Sharabani. Active Man in the Middle Attacks: A Security Advisory. Technical report, IBM, 2009.

[27] StackOverflow. Why is using JavaScript eval function a bad idea? `http://stackoverflow.com/questions/86513/why-is-using-javascript-eval-function-a-bad-idea`.

[28] B. Sterne and A. Barth. Content security policy. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.

[29] Brandon Sterne and Adam Barth. Content security policy 1.1. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`, May 2012.

[30] S. Wagner, J. Jurgens, C. Koller, and P. Trischberger. Comparing Bug Finding Tools with Reviews and Tests. *Lecture Notes in Computer Science*, 2005.

[31] J. Weinberger, A. Barth, and D. Song. Towards Client-side HTML Security Policies. In *Workshop on Hot Topics on Security (HotSec)*, 2011.

[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. Technical report, Symantec, 2009.

A. List of Extensions

We selected 100 extensions from the official Chrome extension directory. We have coded extensions as follows: vulnerable and fixed (†), vulnerable but not fixed (‡), and created by Google (*). We last checked whether extensions are still vulnerable on February 7, 2012.

Most Popular Extensions

The 50 most popular extensions (and versions) that we reviewed are as follows: AdBlock 2.4.6, FB Photo Zoom 1.1105.7.2, FastestChrome - Browse Faster 4.0.6†, Adblock Plus for Google Chrome? (Beta) 1.1.3†, Google Translate 1.2.3.1*‡, Google Dictionary (by Google) 3.0.0*†, Downloads 1, Turn Off the Lights 2.0.0.7, Google Chrome to Phone Extension 2.3.0*, Firebug Lite for Google Chrome 1.3.2.9761†, Docs PDF/PowerPoint Viewer (by Google) 3.5*, RSS Subscription Extension (by Google) 2.1.3*‡, Webpage Screenshot 5.2†, Mail Checker Plus for Google Mail 1.2.3.3, Awesome Screenshot: Capture & Annotate 3.0.4‡, Google Voice (by Google) 2.2.3.4*†, Speed Dial 2.1‡, Smooth Gestures 0.15.2, Xmarks Bookmark Sync 1.0.14, Send from Gmail (by Google) 1.12*, SocialPlus! 2.5.4‡, FlashBlock 0.9.31, AddThis - Share & Bookmark (new) 2.1†, WOT 1.1, Add to Amazon Wish List 1.0.0.4†, StumbleUpon 3.5.18.1†, Google Calendar Checker (by Google) 1.2.1*, Clip to Evernote 5.0.14.9248, Google Quick Scroll 1.8*, Stylish 0.7, Silver Bird 1.9.7.9†, SmoothScroll 1.0.1, Browser Button for AdBlock 0.0.13, TV 2.0.5, Fast YouTube Search 1.2‡, Slideshow 1.2.9†, bit.ly — a simple URL shortener 1.2.1.9, Web Developer 0.3.1, LastPass 1.73.2, SmileyCentral 1.0.0.3‡, Select To Get Maps 1.1.1‡, TooManyTabs for Chrome 1.6.5, Blog This! (by Google) 0.1.1*, TinEye Reverse Image Search 1.1, ESPN Cricinfo 1.8.3†, MegaUpload DownloadHelper 1.2, Forecastfox 2.0.10‡, PanicButton

0.13.1†, AutoPager Chrome 0.6.2.12, RapidShare DownloadHelper 1.1.1.

Randomly Selected Extensions

The 50 randomly selected extensions (and versions) that we reviewed are as follows: The Independent 1.7.0.3†, Deposit Files Download Helper 1.2, The Huffington Post 1.0.5‡, Bookmarks Menu 3.4.6, X-notifier (Gmail, Hotmail, Yahoo, AOL ...) 0.8.2‡, SmartVideo For YouTube 0.94, PostRank Extension 0.1.7, Bookmark Sentry 1.6.5†, Print Plus 1.0.5.0‡, 4chan 4chrome 9001.47‡, HootSuite Hootlet 1.5, Cortex 1.8.3, ScribeFire 1.7‡, Chrome Dictionary Lite 0.2.6†, Taberareloo 2.0.17, SEO Status Pagerank/Alexa Toolbar 1.6, ChatVibes Facebook Video Chat! 1.0.7†, PHP Console 2.1.4, Blank Canvas Script Handler 0.0.17‡, Reddit Reveal 0.2, Greplin 1.7.3, DropBox 1.1.5, Speedtest.or.th 1, Happy Status 1.0.1‡, New Tab Favorites 0.1, Ricks Domain Cleaner for Chrome 1.1.1, Fazedr 1.6†, LL Bonus Comics First! 2.2, Better Reddit 0.0.4, (non-English characters) 1, turl.im url shortener 1.1, Wooword Bounce 1.2, ntust Library 0.7, me2Mini 0.0.81‡, Back to Top 1.1, Favstar Tally by @paul_shinn 1.0.0.0, ChronoMovie 0.1.0, AutoPagerize 0.3.1, Rlweb's Bitcoin Generator 0.1, Nooooo button 1‡, The Bass Buttons 1.95, Buttons 1.4, OpenAttribute 0.6†, Nu.nl TV gids 1.1.3‡, Hide Sponsored Links in Gmail? 1.4, Short URL 4, Smart Photo Viewer on Facebook 1.3.0.1‡, Airline Checkin (mobile) 1.2102, Democracy Now! 1.1‡, Coworkr.net Chrome 0.9.