Extending a Hand to Attackers: Browser Privilege Escalation Attacks via Extensions

Young Min Kim Byoungyoung Lee

Seoul National University
{ym.kim, byoungyoung}@snu.ac.kr

Abstract

Web browsers are attractive targets of attacks, whereby attackers can steal security- and privacy-sensitive data, such as online banking and social network credentials, from users. Thus, browsers adopt the principle of least privilege (PoLP) to minimize damage if compromised, namely, the multiprocess architecture and site isolation. We focus on browser extensions, which are third-party programs that extend the features of modern browsers (Chrome, Firefox, and Safari). The browser also applies PoLP to the extension architecture; that is, two primary extension components are separated, where one component is granted higher privilege, and the other is granted lower privilege.

In this paper, we first analyze the security aspect of extensions. The analysis reveals that the current extension architecture imposes strict security requirements on extension developers, which are difficult to satisfy. In particular, 59 vulnerabilities are found in 40 extensions caused by violated requirements, allowing the attacker to perform privilege escalation attacks, including UXSS (universal cross-site scripting) and stealing passwords or cryptocurrencies in the extensions. Alarmingly, extensions are used by more than half and a third of Chrome and Firefox users, respectively. Furthermore, many extensions in which vulnerabilities are found are extremely popular and have more than 10 million users.

To address the security limitations of the current extension architecture, we present FISTBUMP, a new extension architecture to strengthen PoLP enforcement. FISTBUMP employs strong process isolation between the webpage and content script; thus, the aforementioned security requirements are satisfied by design, thereby eliminating all the identified vulnerabilities. Moreover, FISTBUMP's design maintains the backward compatibility of the extensions; therefore, the extensions can run with FISTBUMP without modification.

1 Introduction

Web browsers are arguably the most attractive attack targets, primarily owing to their role as a gateway connecting people to cyberspace through websites and web applications. Since the COVID-19 pandemic, work and education have shifted to the Internet in home computers. Consequently, if the attacker can trick the user into visiting their malicious site (the web attacker), they can exploit vulnerabilities in browsers and steal security-critical and private-sensitive data from users (such as online banking or social network credentials) [31].

In response to such security threats, browser vendors have made tremendous efforts to secure their end users. In particular, the architecture of web browsers has evolved to strictly enforce the principle of least privilege (PoLP) [50]. A browser instance is divided into multiple functional components, each of which is granted only the privileges necessary to execute a given task. To implement this, modern browsers employ two techniques: (i) a multiprocess architecture [4, 48, 62, 64] and (ii) site isolation [24, 49].

The multiprocess architecture separates the browser into two types of processes: a renderer process, which processes remote content, and a browser process, which coordinates the renderer processes and interacts with the user. Given these processes, the browser restricts the privileges of the renderer process, essentially granting minimal privileges required to fulfill its task. Contrarily, the browser process is privileged because it requires accessing system resources to manage the renderer processes and interact with the user [7, 34].

Site isolation, recently adopted by Chrome and Firefox, further strengthens PoLP on the renderer processes. Site isolation enforces the fundamental browser security principle—the same-origin policy (SOP)—at the process level. Under site isolation, each renderer process is dedicated to a single website. Thus, two different websites are processed using two different renderer processes. Consequently, the two websites are isolated using the process boundary [49].

These PoLP security techniques have made it difficult for web attackers to successfully compromise web browsers. Because an attacker can only manipulate what the renderer process processes, an attack should begin in the renderer to compromise the browser. However, exploiting the renderer process alone does not grant considerable privilege to attackers

because of PoLP. First, the renderer process is sandboxed using a multiprocess architecture. Thus, attackers' access to security-sensitive system resources (*e.g.*, invoking system calls to access local files) is severely limited [7].

Furthermore, through site isolation, the renderer process handles only the data associated with the attacker-controlled website. Therefore, an attacker cannot steal user data from other websites (*e.g.*, online baking sites or social networks). Accordingly, an attacker must find another vulnerability to bypass these enforcements and escape the sandbox [49].

In this paper, we analyze the security aspects of browser extensions from the perspective of PoLP. Extensions are third-party programs that extend browser features to enrich the browsing experience [58]. For instance, users install an ad-blocker extension to block online advertising or a word dictionary extension to quickly search for a word definition. According to Chrome [60], the Chrome Web Store has more than 180,000 extensions, and nearly half of desktop users actively use extensions. According to Firefox [19, 20], approximately one-third of users have installed an extension or theme, and installations increased by 21 % after the COVID-19 lockdown began.

From a security perspective, extensions have two unique characteristics. First, extensions have access to privileged APIs (application programming interfaces) provided by the browser process, that ordinary web pages do not have. Because the key purpose of an extension is to extend browsing features, extensions are allowed to access various features, such as cookie jars, bookmarks, and browsing history, as well as intercept a network request [58]. Second, extensions handle security-critical data relevant to not only the website currently browsed but also other sites and users. For instance, a password manager extension stores the login credentials for any website, and a cryptocurrency wallet extension stores the user's private key [65].

Considering these security characteristics, the extension architecture is also designed to follow PoLP. Each extension is partitioned into two components: a content script and extension page. The content script interacts with a potentially malicious webpage in the renderer process and is, thus, low-privileged. Conversely, the extension page is a separate page that can be browsed or run in the background. The extension page is high-privileged and has access to privileged APIs. The extension page runs in a separate process known as the extension process. The content script and extension page exchange messages, which the content script can use to request privileged operations [3].

However, in this paper, we show that current browsers (including major browsers such as Chrome, Firefox, and Safari) have critical security limitations in enforcing PoLP for extensions. Particularly, the security architecture of extensions demands third-party extension developers to comply with strict security requirements. These security requirements include: (i) communication between extension components should be authenticated, and (ii) extension data management should not violate the same-origin policy (SOP). Unfortunately, we found 59 vulnerabilities in 40 extensions (the full list is provided in Table 1) where such security requirements are violated. By exploiting these vulnerabilities, attackers can circumvent PoLP enforcements and launch privilege escalation attacks (*e.g.*, perform UXSS [43] or steal passwords and cryptocurrency).

In an attempt to elucidate why these requirements are often violated, we find that the current browser architecture is not a security-by-default architecture; instead, it depends on the extension developers to meet these security requirements. However, extension developers are often not experts in browser architecture and thus may not know how to meet these requirements.

To address the security limitations of the current extension architecture, we present FISTBUMP, a new browser extension architecture to strengthen PoLP. FISTBUMP redesigns the extension architecture such that content scripts are isolated from the renderer process using strong process isolation. As a result, FISTBUMP satisfies all security requirements by design and mitigates all vulnerabilities. In particular, FISTBUMP implements a transparent proxy to delegate all requests between the content script and webpage, preserving the backward compatibility of extensions; thus, extensions can run with FISTBUMP without any modification. Moreover, FISTBUMP integrates a batch-request model to optimize the runtime performance of the transparent proxy. According to our performance evaluation, FISTBUMP shows up to 13 % runtime overhead in execution time.

To summarize, this paper makes the following contributions:

- **Analysis: Security Limitation of Extension Architecture.** We identified and analyzed the security limitation of the extension architecture, particularly when the renderer process is compromised. We discovered that the current extension architecture imposes security responsibilities on extension developers; thus, the security of extensions relies on their developers (§3).
- **Practical Results: Privilege Escalation Vulnerabilities.** We found 59 privilege escalation vulnerabilities in various extensions in modern browsers, including Chrome, Firefox, and Safari. These vulnerabilities are caused by violations of the aforementioned security responsibilities and allow critical privilege escalation attacks (§4).
- **Design and Implementation: New Secure Extension Architecture.** We designed and implemented FISTBUMP, a novel secure extension architecture that fundamentally thwarts the aforementioned attacks. FISTBUMP enforces strong process isolation for extensions, resulting in a secure-by-default architecture that eliminates all the vulnerabilities by design (§5 and §6).

2 Background

This section describes background information on the web browser security (§2.1) and the extension architecture (§2.2).

2.1 Web Browser Security

Modern web browsers follow the principle of the least privilege (PoLP) and separates its functionality into multiple processes, where each process is guarded using a strong isolation boundary provided the operating system (OS) [34].

Multi-Process Architecture. Browsers are partitioned into two types of processes, a browser process and renderer processes. The renderer process renders untrusted, potentially malicious, web contents. Therefore, the renderer process is unprivileged and cannot make a system call or access other processes' memory directly.

On the other hand, the browser process is the privileged process through which the renderer process accesses resources. The browser process also enables the communication between renderer processes, acting as an intermediary for the inter-process communication (IPC). This limits the damage to the user's machine if the renderer process is compromised, i.e., a vulnerability in the rendering engine is exploited [48]. In other words, even one renderer process is compromised, the privileged browser process as well as other renderer processes are still secure.

Threats against Renderer Processes. Despite the rendering itself is sandboxed by the multi-process architecture, it is possible for different sites[1] to be rendered in the same sandbox (renderer process) under certain circumstances. Each origin is logically isolated on the software level, but however, this boundary alone is not strong enough against recent threats posed by web attackers [49].

Specifically, consider attackers who have gained the capability to read or write to the address space of the renderer process. Since different sites are rendered in the same renderer process, these attackers can access data of all websites in the process, violating the fundamental principle of the web security, the same-origin policy (SOP) [40]. Hereafter, we denote these attackers as $\texttt{Attacker}_{\texttt{RW}}$ and $\texttt{Attacker}_{\texttt{R}}$.

First, $\texttt{Attacker}_{\texttt{RW}}$ is the attacker who gained the memory read and write capability to the renderer process. This capability can be achieved by exploiting a memory corruption bug in the rendering engine, which allows $\texttt{Attacker}_{\texttt{RW}}$ to execute arbitrary code and perform universal cross-site scripting (UXSS) attacks. Such bugs are highly common due to the complexity of the rendering engine. In fact, Chromium developers stated "[we] assume that determined attackers will be able to find a way to compromise a renderer process" [59].

[1] A site is defined as the effective top-level domain (eTLD) + 1, which is broader than an origin. For instance, `https://example.com:8443` and `https://sub.example.com` are the same site. The site isolation, explained later, uses the site boundary instead of origin due to backward compatibility.

Second, $\texttt{Attacker}_{\texttt{R}}$ is the attacker who gained the memory read capability to the renderer process. This can be achieved by exploiting a micro-architectural side-channel vulnerability against CPU transient execution, such as Meltdown [37] and Spectre [33]. The unique aspect of these attacks is that they rely on vulnerabilities in the microarchitecture rather than the browser, and thus they are difficult to mitigate [47].

Site Isolation. These attacks motivated Chrome and Firefox to employ strong isolation between sites, called Site Isolation in Chrome [49] and Fission in Firefox [24]. The site isolation ensures that each unique site is loaded in a different renderer process, and filters cross-site data from the network request. For example, `example.org` embedded in `example.com` is loaded in a separate process from the renderer process hosting `example.com`, and attempts to request the data of `example.org` will be blocked by the browser process. The site isolation limits the reach of the compromised renderer process to the very site which the render was hosting, preventing $\texttt{Attacker}_{\texttt{RW}}$ and $\texttt{Attacker}_{\texttt{R}}$ from violating the SOP [49].

These measures have shown to be effective mitigations against these threats. In 2022, in Chrome, there were no UXSS vulnerabilities reported and only eight sandbox escape vulnerabilities (four of which require the victim to install a malicious extension). There were only five vulnerabilities that allow $\texttt{Attacker}_{\texttt{RW}}$ to escalate into a sandbox escape, compared to 196 bugs that potentially grant read and write capabilities [18].

2.2 Browser Extension Architecture

Browser extensions are third-party programs that users install to extend browser functionality. Extensions are connected with the browser using an extension API. Most modern browsers, including Chrome, Chromium-based browsers (e.g., Edge, Opera, Brave), Firefox, and Safari, support the Chrome extension [58] or WebExtensions API [41], which is based on web technologies such as HTML, JavaScript, and CSS. Other plugin interfaces such as ActiveX [44], NPAPI [23, 52], PPAPI [35], and XPCOM [46] have been deprecated and removed, and remaining are Chrome/WebExtensions. It is worth noting that Safari app extension [1] can also be considered as a plugin interface, but since it is a regular macOS application, we do not consider it in this paper.

Following the general security principle of web browsers (§2.1), the web extension architecture is also designed with the PoLP. First, an extension is guarded using the permission-based access control. Each extension should declare the list of required permissions in the `manifest.json` file, which should be confirmed by the user when installing the extension. The permission includes the list of browser APIs (e.g., history, cookies, bookmarks) that the extension can access. The permission also includes the list of websites the extension can be activated on. Since permissions are

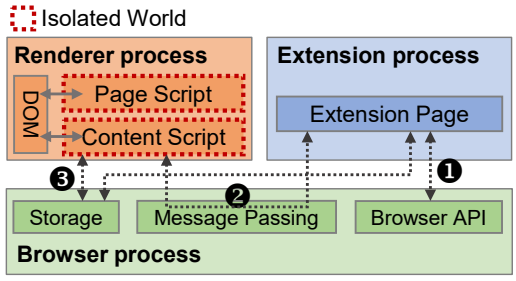

Figure 1: Browser extension architecture. Extensions have two components, an extension page in the extension process and a content script in the renderer process. The browser process provides various APIs for extensions: browser APIs (❶), message passing (❷), and storage (❸).

determined at the installation time and cannot be extended at runtime, the impact of compromised extensions is limited to pre-declared privileges [3].

Second, an extension architecture divides the extension into two parts: high-privileged *extension pages*[2], which run on a dedicated extension process, and low-privileged *content scripts*, which are injected to a renderer process for direct interaction with web pages and thus at a higher risk [38].

Extension Pages. Extension pages have access to aforementioned browser APIs (shown as ❶ in Figure 1) and can make an HTTP request to any origin, as long as the permission is declared in the manifest. To isolate from untrusted web contents, the extension pages run on a dedicated process, which is a special form of renderer process [38].

Moreover, an extension has its own unique ID, which we denote as $\texttt{ID}_{\texttt{EXT}}$. $\texttt{ID}_{\texttt{EXT}}$ is used as the origin of the certain extension page, isolating the extension page from websites and other extensions via the SOP and site isolation [49]. The extension page may run in the background, monitoring and taking action in response to events, e.g., when a message is received, a tab is opened, or a web page is requested [39].

Content Script. Content scripts are unprivileged components of an extension, which let the extension directly interact with the untrusted web page. When a specific web page is loaded, a content script is injected into the page and modifies the page's content via the Document Object Model (DOM). Because content scripts need to access the page's DOM, content scripts run in the same renderer process running the page. As a result, content scripts are at a higher risk of compromise, and thus it is unprivileged [8].

In order to access the browser API, it should rely on the extension pages through message passing, which we explain later. In addition, the content script has the same origin as the page and cannot request cross-origin data, enforcing the site isolation [11].

To isolate the content script from untrusted page scripts running in the same renderer process, the content script runs within an *isolated world*, a software-based isolation mechanism providing a private execution environment. An isolated world has its own JavaScript heap and DOM wrapper. Consequently, page scripts cannot access variables defined by content scripts and even if a built-in object is modified by page scripts, content scripts see its own version. This holds true for the other way around. For example, even if the content script defines `document.foo`, the page script cannot see `document.foo` defined, and vice-versa [3].

Extension Message. Since two extension components, extension page and content script, run on different processes, the communication between is carried out using the message passing (shown as ❷ in Figure 1). By sending a message, the unprivileged content script can request for privileged operations provided by the extension page [9].

An extension message is constructed by the component sending the message, which contains three fields: sender, recipient, and payload. The sender represents which component sent the message, including $\texttt{ID}_{\texttt{EXT}}$, `URL`, and `origin` of the extension page or the content script (i.e., in the case of the content script, the corresponding information of the page is specified). The recipient represents the destination component of the message, which includes component type and $\texttt{ID}_{\texttt{EXT}}$. The payload is the data to be passed, serialized into a JavaScript Object Notation (JSON). The composed message is then relayed through the browser process via IPC and delivered to the specified recipient [21].

Extension Storage. Browsers support persistent data storage for extensions, called the extension storage. Using this storage, the extension is able to store various user data preserved even after the browser restarts. The storage resides in the browser process and each given extension has a separate dedicated storage, i.e., the storage is associated to $\texttt{ID}_{\texttt{EXT}}$. As a result, extension components cannot access another extension's storage. Both extension page and content script can access the storage by making a request to the browser process (shown in ❸ in Figure 1) [10].

Example: Password Manager Extension. Consider an extension with $\texttt{ID}_{\texttt{ext}} = \texttt{E}$ which saves credentials (which includes ID and password) for websites. As visiting any login page, the extension provides an interface for the user to enter the credential. Then such credentials are stored on the extension storage. When the user visits the login page later, the extension automatically fills up the corresponding credential.

The extension also provides a special *admin* page, which shows a full list of saved credentials. More specifically, when the user visits the admin page, the content script (denoted as `C`) is injected to the admin page, which requests the list of saved credentials (shown as ❶ in Figure 2). This request is done through the extension message, indicating the followings: i) it is sent by the content script `C` injected in the admin page; ii) it is destined to the extension page of `E`; and iii) it requests

[2]For the sake of simplicity, we consider other contexts such as workers as a page.

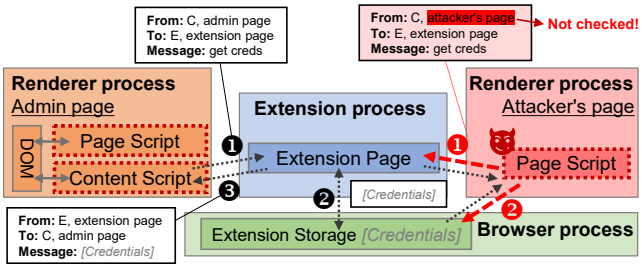

Figure 2: An example of password manager extension. The left-side with admin page illustrates the benign workflows. The right-side with attacker's page illustrates two attacks against the extension.

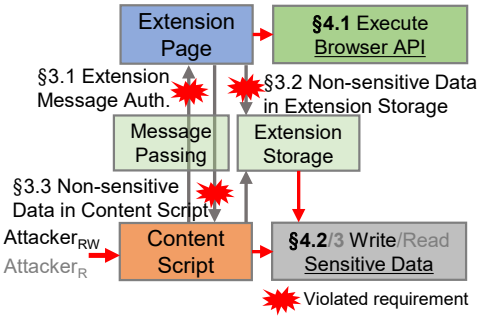

Figure 3: The flow of privilege escalation attacks through violated security requirements.

for getting the credentials.

Once the extension page `E` receives the message, it retrieves the list of credentials from the extension storage in the browser process (❷), and sends it back to the content script `C` (❸).

3 Security Requirements to Protect Against Renderer Attackers

This section analyzes the security requirement of extension design to protect against recent threats of renderer attackers. As the complexity of the rendering engine increases, the number of vulnerabilities have risen up. Furthermore, various defense and mitigation mechanisms such as site isolation have been deployed, preventing the renderer attacker from gaining additional capabilities. As a result, the extension security has become the primary line of defense.

To be specific, the content script runs within the renderer process, so $\texttt{Attacker}_{\texttt{R}}$ and $\texttt{Attacker}_{\texttt{RW}}$ gain capability to read or write the content script, respectively. According to the PoLP, gaining such capability over the content script alone should not grant the attacker additional capability. This is because the content script is an unprivileged component of an extension, running on an unprivileged renderer process. Thus, attackers still cannot access browser APIs provided to privileged extension pages, neither can access system resources provided by the browser process.

However, we found that the current PoLP enforcement over extensions as secure as extension developers strictly keep a certain set of security requirements in developing their extensions. If any of these security requirements is violated, it would render PoLP useless, leading to privilege escalations. Unfortunately, not all extension developers are security experts, and they are less incentivized to write a secure extension. In the following of this section, we introduce three security requirements that extension developers need to follow as well as how each of those can be violated, leading to a critical security vulnerability (illustrated in Figure 3).

3.1 Extension Message Authentication

An extension message can be sent by the content script to request a privileged operation provided by the extension page. In this case, the extension message is sent by the low-privileged component (content script) and delivered to the high-privileged component (extension page). Therefore, it is important for the extension page to thoroughly check if the sender content script is legitimate and can request such operation. This is particularly important in the presence of $\texttt{Attacker}_{\texttt{RW}}$, because $\texttt{Attacker}_{\texttt{RW}}$ can forge IPC message for exchanging extension messages.

Security Requirement 1 *The extension page should authenticate the extension message if the sender content script is legitimate.*

However, we found that extension developers often fail to meet this security requirement. The full list of vulnerable extensions are presented in Table 1. Such failures can be categorized into the following two cases: 1) the extension page does not authenticate the sender; and 2) the extension page improperly authenticates the sender.

First, many extension pages do not authenticate the sender. We suspect this is because many extension developers do not consider the threat model of $\texttt{Attacker}_{\texttt{RW}}$, i.e., the extension message can be forged.

Second, many extension pages improperly authenticate the sender, presumably because it is technically challenging to do so correctly. A representative example would be authenticating the sender using the URL, which is provided as part of the sender information. The URL is tricky to parse, because some URLs need special handling; the URL can be a special URL such as `about:blank`, `about:srcdoc`, `data:`, `blob:` and the origin can be opaque, i.e., `null`.

Another instance is the time-of-check time-of-use (TOCTOU) race condition. For instance, site A requests to run script on the current tab. Since the current tab is showing site A, the extension runs the script. However, site A can navigate to site B and the script is executed in site B.

To make matters worse, the extension API does not provide a straightforward method, e.g., `isTrusted`, to authenticate the sender and has several implementation errors, which we discuss further in §8.

```
// Vulnerable extension's background page
chrome.runtime.onMessage.addListener((message, sender, send) => {
  // Improperly authenticates the URL
  if (sender.url.startsWith("https://admin.com")
      && message == "getCredentials")
    sendResponse(credentials);
});

// AttackerRW on https://admin.com.attacker.com
chrome.runtime.sendMessage("getCredentials")
```

Listing 1: The vulnerable pattern and a PoC exploit.

For instance, recall the previous password manager extension example. The extension message requesting the list of credentials should only come from the content script running in the admin page. However, if the extension page does not check (or does incorrectly check) the sender, the attacker who does not have control of the admin page can also request for credentials (shown as ❶ in Figure 2). Listing 1 shows the vulnerable pattern and a POC exploit. The background page authenticates only the URL prefix, so $\texttt{Attacker}_{\texttt{RW}}$ can register https://admin.com.attacker.com and send a message requesting credentials.

3.2 Non-sensitive Data in Extension Storage

The extension storage is used to store the persistent data of the extension, which is accessible by both extension pages and content scripts. From the security perspective, the renderer process that has been injected with content script should be capable of accessing the extension storage. Therefore, once the renderer process is compromised, $\texttt{Attacker}_{\texttt{RW}}$ can read and modify the extension storage, and only data that are safe for websites to access should be stored on the extension storage.

Security Requirement 2 *The extension should not store security-critical, privacy-sensitive, or cross-site data on the extension storage.*

We found many cases that the security requirement on the extension storage is not adhered. The full list of extensions is presented in Table 1. We suspect this is because extension developers consider that the extension storage can only be accessed by the extension page and content script that they have programmed. As a result, extension developers consider the extension storage cannot be accessed by the attacker, and thus store the sensitive data.

```
// Vulnerable extension's background page
// Stores credentials on the extension storage
chrome.storage.set("credentials", credentials);

// AttackerRW on any page
chrome.storage.get("credentials")
```

Listing 2: The vulnerable pattern and a PoC exploit.

For instance, recall the previous example password manager extension. Credentials are stored on the extension storage, allowing the attacker to illegally access credentials (shown as ❷ in Figure 2). Listing 2 shows the vulnerable pattern and a POC exploit. The background page stores credentials on the extension storage, so $\texttt{Attacker}_{\texttt{RW}}$ can retrieve them on any page.

3.3 Non-sensitive Data in Content Script

Content script runs on the renderer process, as it directly interacts with DOM. In other words, the content script is in the address space of renderer process, and $\texttt{Attacker}_{\texttt{R}}$ and $\texttt{Attacker}_{\texttt{RW}}$ can read it. This is particularly alarming as $\texttt{Attacker}_{\texttt{R}}$ does not depend on bugs in the browser and is hard to mitigate.

Furthermore, Chrome and Firefox attempt to mitigate timing side-channel attacks by restricting availability of high-granularity timers only to cross-origin isolated pages [45, 63]. However, the cross-origin isolation does not affect content script injection, so a web page can perform attacks against content scripts with high-granularity timers.

Therefore, the extension should not store any sensitive data in the content script.

Security Requirement 3 *The extension should not load security-critical or privacy-sensitive data on the content script.*

```
// Vulnerable extension's background page
// Send sensitive data to the content script
chrome.tabs.sendMessage(tabId, sensitiveData);

// The message is enqueued in the renderer process's message queue,
// so AttackerR can read the message
readMemory();
```

Listing 3: The vulnerable pattern and a PoC exploit.

However, we found several extensions do not follow this security requirement (listed in Table 1), similar to the reason of aforementioned cases—i.e., the extension developers do not consider the threat model of $\texttt{Attacker}_{\texttt{R}}$ and $\texttt{Attacker}_{\texttt{RW}}$. In other words, the extension developers think the data placed in the content script can only be accessed by the content script itself. Listing 3 shows the vulnerable pattern and a POC exploit. The background page sends sensitive data to the content script, of which memory $\texttt{Attacker}_{\texttt{R}}$ can read.

4 Privilege Escalation Attacks via Extensions

Given security requirements imposed on extensions, we analyzed whether real-world extensions meet these requirements. Unfortunately, we found that many extensions fail to meet such requirements, resulting in privilege escalation attacks.

Based on our analysis, we devise three new privilege escalation attacks that allow to bypass SOP and execute script on another site, i.e., universal cross-site scripting (UXSS).

Extension Name	Violation	Attack	Impact	Status
Adblock Plus - free ad blocker	3.1, 3.2	4.2	UXSS (limited to predefined scriptlets)	Fixed
AdBlock – best ad blocker	3.1, 3.2	4.2	UXSS (limited to predefined scriptlets)	
AdGuard AdBlocker	3.1, 3.2	4.2	UXSS	Fixed
uBlock Origin	3.1†	4.1	Fetch cross-origin resource, get or create tabs	Fixed
	3.1†, 3.2	4.2	UXSS	
Ghostery – Privacy Ad Blocker	3.1	4.1	Fetch cross-origin resource, get or create tabs	Confirmed
	3.2	4.2	UXSS	
Fair AdBlocker	3.1	4.1	UXSS	Confirmed
	3.2	4.2	UXSS	
AdBlocker Ultimate	3.1	4.1	Get or create tabs	Based on
	3.1, 3.2	4.2	UXSS	uBlock Origin
Honey	3.1	4.1	UXSS, read and modify cookies, get or create tabs	Fixed
	3.3	4.3	UXSS	
Google Translate	3.2	4.2	UXSS	Fixed
Tampermonkey	3.1†	4.1	Intercept network requests, read & modify cookies	Fixed
	3.1†, 3.2	4.2	UXSS	
Adobe Acrobat	3.1, 3.3	4.3	Captured page	Reported
Read&Write for Google Chrome	3.1	4.1	Fetch cross-origin resource	Reported
ClassLink OneClick Extension	3.1	4.1	UXSS	Reported
Cisco Webex Extension	3.1	4.1	Start Cisco Webex Meetings application	Confirmed
Netflix Party is now Teleparty	3.1	4.1	UXSS (under special condition)	Fixed
Amazon Assistant for Chrome	3.1	4.1	Read and modify cookies	Confirmed
Windows Accounts Office	3.1†	4.3	Windows Account and Azure Active Directory account takeover	Confirmed
LastPass: Password Manager	3.1	4.3	Encryption key, saved passwords	Fixed
Avira Password Manager	3.1, 3.2			Confirmed
Keeper® Password Manager	3.1			Fixed
Dashlane - Password Manager	3.1†, 3.2	4.3	Encryption key (if persistent login is used), saved passwords	Fixed
Bitwarden - Password Manager	3.1, 3.2			Fixed
RoboForm Password Manager	3.1, 3.2			Confirmed
Norton Password Manager	3.1†	4.3	Saved passwords	Confirmed
1Password – Password Manager	3.1			Fixed
MetaMask	3.1, 3.2	4.3	Sign a blockchain transaction, steal cryptocurrency	Confirmed
Ronin Wallet	3.1			Reported
Binance Wallet	3.1			Confirmed
Keplr	3.1†			Confirmed
Phantom	3.1†	4.3	Wallet mnemonic and seed, steal cryptocurrency	Confirmed
TronLink	3.1			Confirmed
Kaikas	3.1, 3.2			Confirmed
Stormcrow (Opera)	3.1	4.1	Capture other sites	Confirmed
Background Worker (Opera)	3.1	4.1	Modify browser settings	
Video Handler (Opera)	3.1	4.1	Modify browser settings	
8 Opera component extensions	3.1†	4.1	Modify browser settings	
QuickSearch (Whale)	3.1	4.1	Modify browser settings, access account information, get or create tabs	Fixed
Image Translate (Whale)	3.1	4.1		
Whale WebUI (Whale)	3.1†	4.1	Fetch cross-origin resource	Confirmed
Naver Memo (Whale)	3.1†	4.1	Capture other sites	Confirmed

Table 1: List of vulnerable extensions. Column Violation indicates which requirements are violated and lead to the vulnerability. † indicates the attack requires browser bugs discussed in §8.

With UXSS, the attacker can bypass the SOP, exfiltrate data, e.g., read victim's email, and perform actions on behalf of the victim, e.g., make a bank transfer. In the following, we describe these three attacks, namely executing privilege browser APIs (§4.1), writing sensitive extension data (§4.2), and reading sensitive extension data (§4.3), where the overall attack flow is illustrated in Figure 3.

Methodology. We modified ExtensionCrawlerciteextensioncrawler to collect extensions from the Chrome Web Store and Firefox Add-ons as of April 9, 2022. We excluded Chrome Apps, which were deprecated in 2020, and themes, which have no JavaScript component. We also excluded extensions that are not available for download or are unlisted, i.e., do not appear in the search results. We could not collect extensions for Safari because it does not allow crawling and downloading extensions.

Then, we selected the top 20 extensions with most users in each browser. We also selected extensions bundled with Chrome, Opera, Brave, and Whale. Firefox and Safari did not have bundled extensions. The list of vulnerable extensions is listed in Table, where the full list of analyzed extensions is in Table.

We installed each extension, and examined what messages are exchanged and what data is stored. We developed a DevTools extension to intercept extension messages and browse the extension storage. We then manually inspected the source code of the extension, focusing on how extension messages are handled and how the stored data is used.

4.1 Execute Privileged Browser APIs

Since browser APIs can access another site's data or modify browser behavior, extension messages that call browser APIs should be authenticated (Security Req. 1 described in §3.1). However, we found many extensions fail to meet this requirement—i.e., they either do not authenticate or incorrectly authenticate the extension messages. Specifically, we found 23 extensions, including 15 component extensions, allowing the attacker access privileged browser API without restriction.

Case Study: Honey. The extension `Honey` allows unrestricted access to `executeScript` API, which then allows to execute arbitrary JavaScript in opened tabs, resulting in UXSS. It also broadcasts tab event to all content scripts, leaving other tab information on the content script memory.

Case Study: Tampermonkey. This extension allows invoking browser APIs, such as `fetch`, `tabs`, and `cookies` APIs, thereby allowing to bypass the SOP and read cross-site data.

Case Study: ClassLink OneClick Extension. `tabs` and `executeScript` API uses the tab ID to specify the tab. The tab ID is unique per tab, not per site, i.e., even if the tab is navigated to another site, the ID does not change. By sending the request when the page is unloaded, API calls bound to the current tab will be dispatched to the new site. The attacker can exploit this race by sending the request on `unload` event to execute the script, and the script will be executed on the new site, leading to UXSS.

Case Study: Opera Component Extensions. Opera exposes the `settingsPrivate` API to content script, allowing attackers to modify browser settings. DNS/proxy settings can be manipulated to perform a man-in-the-middle (MITM) attack. Furthermore, some settings have been used to perform UXSS or escape the sandbox.

4.2 Write Sensitive Extension Data

As the extension pages have higher privileges than content scripts, configurations that affect the extension pages behavior should not be modifiable by content scripts. Therefore, they should not be modifiable via extension messages from content scripts (according to Security Req. 1) or should not be stored on the extension storage (according to Security Req. 2).

However, we found many extensions allow modification via extension messages (breaking Security Req. 1) or store sensitive configurations on the extension storage (breaking Security Req. 2), allowing the attacker to manipulate the extension behavior. We focus on configurations that affect injected script on the extension storage, eventually leading to UXSS.

Case Study: Ad Blockers. Ad blockers allow the user to add a custom rule and some rules allow to inject a script to remove dynamically injected ads (with Adblock Plus and AdBlock, only predefined scriptlets can be injected). For example, a filter rule `example.com#%#alert(document.domain)` executes `alert(document.domain)` on `example.com`.

In six ad blockers, the attacker could spoof a request for adding a custom rule and run arbitrary code on web sites. Six ad blockers also stored custom rules on the extension storage, which the attacker could modify.

Case Study: Tampermonkey. Userscript managers Tampermonkey allow to user to add a script, called userscript, that runs on specific pages, just like a content script. In Tampermonkey, the attacker could spoof a request for adding a userscript and run arbitrary userscript on web sites. Tampermonkey also stored userscripts on the extension storage, which the attacker could modify.

Furthermore, userscript can access extension APIs by making a request to the background page via the content script. The attacker could spoof the request and call extension APIs.

Case Study: Google Translate. Google Translate extension translates the page by injecting a script to the page. The user can choose which language to translate to from a list and this configuration is stored on the extension storage. Since the configuration can be only one of predetermined values, the extension injected the value, without validating it first. The attacker could modify a value to an arbitrary value, e.g.,

scripts, and execute script on translated pages (XSS).

4.3 Read Sensitive Extension Data

When handling the security sensitive data, extensions should carefully store and safely control the access to those. Therefore, in order to prevent the renderer attackers from accessing the data, sensitive data should not be accessible via extension messages (Security Req. 1), or it should not be stored either on the extension storage (Security Req. 2) or on the content script memory (Security Req. 3). However, we found 19 extensions fail to meet these requirements, exposing sensitive data to the attacker.

Case Study: Windows Account and Office. Windows Account and Office extension allows the user to sign in with Windows or Azure Active Directory (AAD) accounts on Windows. When the user visits the login page, the content script requests the background page to retrieve the token from the OS. By spoofing the URL as the login page, the attacker could steal the token and takeover the account.

Case Study: Password Managers. Password managers store credentials for web sites. When the user visits the login page, the content script requests the extension page for the saved credential. In LastPass and Bitwarden password manager, by spoofing the URL as the target site, the attacker could steal credentials for that site. Both password managers also stored the encryption key on the extension storage, which the attacker could access.

We performed additional analysis on six more password managers, and found in all password managers, the attacker could steal credentials. In four password managers, the attacker could access the encryption key.

Case Study: Cryptocurrency Wallets. Cryptocurrency wallets store private keys to sign transactions on the blockchain. When the page requests to sign a transaction, the content script forwards the request to the background page and the background page shows a notification to the user to confirm the transaction. If the user confirms the transaction, the notification sends a request to sign the transaction. In MetaMask, the attacker could spoof the confirmation message, signing an arbitrary transaction. MetaMask also stored transaction queue on the extension storage, to which the attacker could add an arbitrary transaction.

We performed additional analysis on seven more cryptocurrency wallets, and found in four wallets, the attacker could sign an arbitrary transaction.

Furthermore, the user can view the mnemonic and private key in the popup. When the user requests to view the mnemonic and private key, the popup requests the background page to send them. In Phantom, TronLink, and Kaikas, the attacker could spoof the request and retrieve the mnemonic and/or private key.

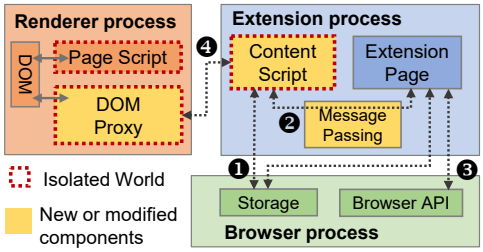

Figure 4: FISTBUMP architecture.

5 Design of FISTBUMP

In the presence of renderer attackers ($\texttt{Attacker}_{\texttt{RW}}$ and $\texttt{Attacker}_{\texttt{R}}$), the current extension architecture demands extension developers to comply with three security requirements presented in §3. However, it is challenging for extension developers to properly separate privileges between extension pages and content scripts, and such security requirements are often violated, leading to critical privilege escalation attacks.

For these reasons, we present FISTBUMP, a new extension architecture which protects content scripts from the renderer attacker using the strong process isolation. To this end, instead of running content scripts in the renderer process, FISTBUMP redesigns the extension architecture to run content scripts in the extension process. As a result, the process running content scripts is isolated from the renderer process, utilizing the process as a protection domain to prevent attackers from gaining the content script capabilities. In other words, the attacker cannot forge an extension message or access the extension storage.

Therefore, by design FISTBUMP satisfies three security requirements and eliminates vulnerabilities presented in §4. From the perspective of the extension developer, FISTBUMP shifts the challenging burden of meeting the security requirements to the browser and the OS.

Design Overview. The overall architecture of FISTBUMP is illustrated in Figure 4. In order to enforce the strong process isolation, FISTBUMP moves content script to the extension process (§5.1). To maintain the functionality and compatibility of content scripts, FISTBUMP introduces $\texttt{DOM}_{\texttt{Proxy}}$ (§5.2). Furthermore, in order to optimize the performance of $\texttt{DOM}_{\texttt{Proxy}}$, FISTBUMP further develops tailored memory management as well as batch processing (§5.3). We note that FISTBUMP can be adopted by extensions as well as browsers, which is further discussed in §8.

5.1 Strong Process Isolation for Content Script

Design Goal 1 *Strongly isolate content scripts from the renderer process.*

The root cause of the privilege escalation attack in §3 is due to the fact that the current isolation mechanism between content scripts and the renderer is not sufficient to thwart

$\texttt{Attacker}_{\texttt{RW}}$ and $\texttt{Attacker}_{\texttt{R}}$. To this end, FISTBUMP employs a stronger notion of isolation mechanism, namely based on the process isolation. As such, a process running the content script should be different from the renderer process, thereby preventing access to the content script by the renderer.

Specifically, FISTBUMP moves the content script to the extension process, where the extension pages are running on. Within the extension process, FISTBUMP runs each content scripts using a dedicated worker thread, so as to preserve the execution characteristics of modern browsers. The modern browsers implement an independent browser tab, so the execution contexts of renderer processes and its associated extensions are independent to each other. The lifecycle of a content script is as follows. First, when a content script is to be injected (e.g., a page is loaded by a renderer process), FISTBUMP a worker and runs the content script in the worker within an extension process. If the page is unloaded later, then FISTBUMP accordingly terminates the content script worker.

The content script worker also runs in an isolated world, a private execution environment, of which privileges are restricted to the same level of the original content script. The content security policy (CSP) is set to prevent the content script worker from executing remote code, e.g., code that is not included in the extension.

As a result, the content script data is kept out of the renderer process, protecting it from $\texttt{Attacker}_{\texttt{R}}$ and $\texttt{Attacker}_{\texttt{RW}}$ by the design.

In addition, the renderer no longer needs privileges of the content script, such as sending extension messages or accessing the extension storage. Following the PoLP, FISTBUMP removes these privileges from the renderer process, so a compromised renderer no longer impersonate a content script.

5.2 Transparent Isolation with DOM Proxy

Design Goal 2 *Provide a transparent isolation of content scripts with backward compatibility.*

It is natural that adding a strong isolation mechanism may entail radical changes in the software architecture, non-trivial engineering costs. For instance, in order to enforce site isolation (which also employed process isolation), browser vendors have invested non-trivial engineering costs [49]. In order to minimize the engineering costs, FISTBUMP aims at providing transparent isolation mechanism with backward compatibility. More precisely, the isolation mechanism of FISTBUMP should not interfere any functional feature of web extensions, and it should be able to run existing browser extensions without manual modification.

Transparent Proxy for DOM with Delegation. The main feature of the content script is to interact with the page's DOM. This raises an issue for FISTBUMP, which requires new mechanism to connect between content scripts and DOM. Specifically, the current browser architecture has all these components (i.e., content scripts, DOM, and page scripts) in the same renderer process, the content scripts can directly access DOM. However, FISTBUMP moves the content script from the renderer process to the extension process. Thus, the content scripts have separate virtual address space and cannot directly access DOM.

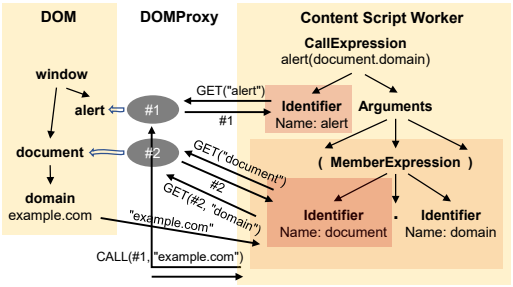

Figure 5: Content Script Execution with $\texttt{DOM}_{\texttt{Proxy}}$.

To address this issue, FISTBUMP proxies DOM access with delegation. Instead of injecting the content script to the renderer process, FISTBUMP inserts the proxy to interact with the DOM, which we call $\texttt{DOM}_{\texttt{Proxy}}$. The content script worker and $\texttt{DOM}_{\texttt{Proxy}}$ communicate using IPC. They exchange JSON-serialized message containing purely DOM operation or event data.

When a DOM object is accessed by the content script in the extension process, the content script worker forwards the operation to $\texttt{DOM}_{\texttt{Proxy}}$. Then $\texttt{DOM}_{\texttt{Proxy}}$ performs the operation as requested, then returns either i) a resulting value or ii) a reference to the resulting object. If a reference is returned, the content script worker creates a *delegate object* for the given reference, and all operations on the delegate object is intercepted and forwarded to $\texttt{DOM}_{\texttt{Proxy}}$.

The content script can also register listener for DOM events. If an event listener is registered by the content script, $\texttt{DOM}_{\texttt{Proxy}}$ registers a corresponding proxy event listener. If the event is fired in the renderer process, the event is forwarded to the content script worker by $\texttt{DOM}_{\texttt{Proxy}}$. Then content script worker raises the delegate event (i.e., a clone of an original event), which will finally be dispatched to the destined event listener in the content script.

Example: Content Script Execution. To clearly show how $\texttt{DOM}_{\texttt{Proxy}}$ operates, we provide an example how $\texttt{DOM}_{\texttt{Proxy}}$ executes the content script code, `alert(document.domain)`. The abstract syntax tree (AST) is shown on the right side of Figure 5.

First, JavaScript evaluates `alert`, which is looking up the identifier named `alert`. The content script worker intercepts and forwards `GET("alert")` request to $\texttt{DOM}_{\texttt{Proxy}}$. Since `alert` is a function, $\texttt{DOM}_{\texttt{Proxy}}$ creates a reference, $\texttt{REF}_{\texttt{alert}}$ and returns the ID (#1) back to the content script worker. Upon receiving the reference, the content script worker creates a delegate function, $\texttt{DEL}_{\texttt{alert}}$.

Second, JavaScript evaluates `document`. Likewise, as

document is an object, $\texttt{DOM}_{\texttt{Proxy}}$ creates a reference, $\texttt{REF}_{\texttt{document}}$ and returns the ID (#2). The content script worker creates a delegate object, $\texttt{DEL}_{\texttt{document}}$.

Then, JavaScript evaluates $\texttt{DEL}_{\texttt{document}}$`.domain`. This is a member expression, retrieving the property named `domain`. The content script worker intercepts and forwards `GET(#2, "domain")` request to $\texttt{DOM}_{\texttt{Proxy}}$. $\texttt{DOM}_{\texttt{Proxy}}$ retrieves the object corresponding to #2, `document`, and looks up the property named `domain` to be `"example.com"`. Since its type is a string primitive, $\texttt{DOM}_{\texttt{Proxy}}$ sends it as is.

Finally, JavaScript evaluates $\texttt{DEL}_{\texttt{alert}}$`("example.com")`, which is calling $\texttt{DEL}_{\texttt{alert}}$ with the argument `"example.com"`. The content script worker forwards `CALL(#1, "example.com")` request to $\texttt{DOM}_{\texttt{Proxy}}$ and $\texttt{DOM}_{\texttt{Proxy}}$ retrieves the function corresponding to #1, `alert` $\texttt{DOM}_{\texttt{Proxy}}$ evaluates `alert("example.com")`, which is equivalent of running `alert(document.domain)` in the content script.

Forwarding Extension API Calls. Another feature of the content script is to call extension APIs. As described in §2, the current browser architecture implements extension API calls with IPC message passing between two different processes. However, because FISTBUMP places a content scripts and a background page in the same process, it no longer requires the IPC message passing. Thus, FISTBUMP implements an in-process extension API call mechanism, which is forwarded by the content script worker. Specifically, all the extension API calls by the content script is first intercepted by the content script worker. Then the content script worker forwards the call to the background page.

5.3 Optimizing Performance of DOM Proxy

Design Goal 3 *Provide an isolation with reasonable performance overhead.*

Since FISTBUMP proxies all DOM accesses of content scripts, it complicates execution behaviors of content scripts, which negatively impacts the performances in terms of memory management and execution speed.

Memory Management. One problem of $\texttt{DOM}_{\texttt{Proxy}}$ is that references to objects and functions in $\texttt{DOM}_{\texttt{Proxy}}$ accumulate, even if they are no longer used in the content script worker, causing memory leaks. $\texttt{DOM}_{\texttt{Proxy}}$ solves this by deleting the reference if the delegate object gets garbage-collected in the content script worker.

Batch Processing and Cache. To reduce the amount of inter-process communication, the content script maintains a queue of proxied operations without side effects, and send them in batch. The content script worker maintains a virtual DOM representation and operations not dependent on the document, e.g., operations on orphan nodes are executed in the content script worker. The content script worker also caches properties that are invariant or of which validity can easily be checked.

6 Implementation

We implemented FISTBUMP to be compatible with the latest Chromium browser (Chromium 105 at the time of writing). FISTBUMP consists of two parts: an extension wrapper written in about 3k lines of JavaScript and Chromium-side modification written in about 100 lines of C++. The extension wrapper is implemented around the extension API, i.e., $\texttt{DOM}_{\texttt{Proxy}}$ is implemented as a content script and the content script worker is implemented in the background page. They communicate using the extension messaging. The extension wrapper uses standard Web APIs and JavaScript (ECMAScript) features, so it is compatible with the latest Firefox and Safari. The extension wrapper can also be easily applied to existing extensions which do not use `"document_start"` content script, so extension developers can also adopt FISTBUMP.

In order to handle `"document_start"` content script, modification on the browser side is necessary. Nevertheless, the modification is small, and we expect modification needed in other browsers to be similarly small.

FISTBUMP and related toolchains will be open sourced and available at https://github.com/compsec-snu/exthand.

7 Evaluation

In this section, we evaluate various aspects of FISTBUMP, particularly focusing on its security (§7.1), compatibility (§7.2), and performance (§7.3).

Experimental Setting. We tested our implementation on a machine with Intel i7-10700K and 32 GB RAM. We built the Chromium browser based on the tag 101.0.4951.41. For comparison, we prepared two sets of browsers, with and without FISTBUMP. We used a DOM Fuzzer, Domato [26] for generating test HTML and used `html.txt` grammar provided by Domato.

7.1 Security

The foremost goal of FISTBUMP is to strengthen the PoLP of the current extension architecture. Thus, we evaluate the security aspect of FISTBUMP in this subsection.

Security Analysis. As described in §3, the current extension architecture demands from extension developers to keep three security requirements. This in fact motivated FISTBUMP, which attempts to satisfy all such security requirements by design. In the following, we describe and reason about how the design of FISTBUMP indeed meets each security requirement.

First, extension messages under FISTBUMP cannot be sent by the renderer process, because the content script is moved to the extension process in FISTBUMP. Therefore, the message can only come from the extension process (which is secure against $\texttt{Attacker}_{\texttt{RW}}$), and thus all sender information can be

trusted. Thus, FISTBUMP does not impose the responsibility of Security Req. 1 on extension developers, and accordingly eliminate all the corresponding vulnerabilities.

Second, the extension storage cannot be accessed by the renderer process under FISTBUMP, because the extension storage is only accessible from the extension process. Therefore, all the data saved in the extension storage is secure against $\texttt{Attacker}_{\texttt{RW}}$, eliminating all the vulnerabilities corresponds to Security Req. 2.

Lastly, the memory footprints of content scripts cannot be accessed by the renderer process, because the content script runs in separate virtual address space in FISTBUMP. Therefore, $\texttt{Attacker}_{\texttt{R}}$ and $\texttt{Attacker}_{\texttt{RW}}$ cannot fetch any data from the content script, and thus FISTBUMP does not impose Security Req. 3 and mitigate all corresponding vulnerabilities.

Moving the content script to the extension process may expose additional attack surfaces. $\texttt{Attacker}_{\texttt{RW}}$ can spoof a message from $\texttt{DOM}_{\texttt{Proxy}}$, but messages between $\texttt{DOM}_{\texttt{Proxy}}$ and content script worker use existing extension message implementation and contain purely structured data. They do not exchange JavaScript code or pointer and $\texttt{DOM}_{\texttt{Proxy}}$ cannot directly alter the control flow of content script worker.

Furthermore, the content script worker runs with content script privileges and cannot run remote code, e.g., code provided by the attacker. Therefore, for the attacker to exploit the extension process, they need to find either:

- an arbitrary code execution vulnerability in the content script and a CSP-bypass vulnerability, or
- a gadget in content script to trigger and a gadget to exploit a memory corruption.

We believe these vulnerabilities are difficult to be found, and if found, they would allow more serious attacks without the need of the extension. There were no CSP bypass vulnerabilities reported in 2022 and all memory corruption bugs required passing invalid argument or specific user interactions [18], which are highly unlikely in normal content scripts.

Exploit Mitigation. In order to check if FISTBUMP stops the concrete attacks that we demonstrated in §4, we tried to reproduce the attack while running the Chromium browser with FISTBUMP. For each extension, we first installed it in FISTBUMP-enabled Chrome, and then launched a renderer remote code execution (RCE) attack based on a type confusion vulnerability CVE-2022-1134 [42]. Then we executed privilege escalation attacks PoC we created in §4. The result showed that the same privilege escalation attack against all tested extensions are no longer working, indicating that the security design and its implementation of FISTBUMP is effective as expected.

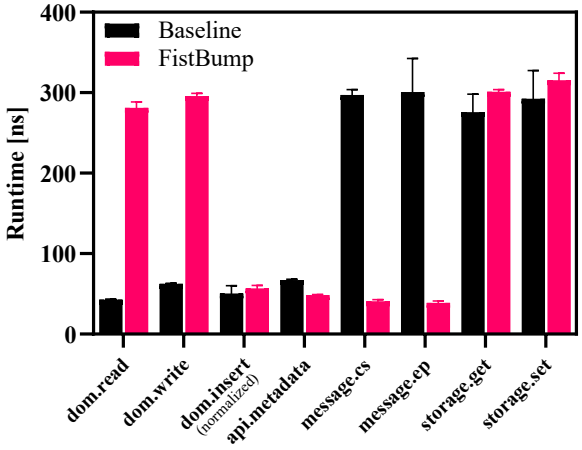

Figure 6: Mean runtime of each operation in original Chrome (baseline) and FISTBUMP-enabled Chrome. Runtime of `dom.insert` is normalized to a unit operation. Error bars represent one standard deviation.

7.2 Backward Compatibility

Unit Test. We ran a Chromium unit test suite, which includes 75 unit test on the content script API [17]. We found all 75 unit tests passed, showing the content script implementation of FISTBUMP is correct and backward compatible.

Limitation. JavaScript uses an event loop, which waits and processes a message on a message queue. Each message is a function call and runs to completion, i.e., is processed completely before processing the next message. In other word, the message processing blocks the runtime. However, with $\texttt{DOM}_{\texttt{Proxy}}$, each expression or statement is evaluated in separate messages. This allows other message such as an event handler to be processed in the middle of running a code block and cause inconsistency.

We tested all analyzed extensions how this limitation affect their operation. We installed the extension on both FISTBUMP-enabled browser and original browser and compared their behavior, while performing their functionality. For example, with a password manager extension, we saved a password and checked that it gets successfully filled onto the login form.

We did not find instances where this limitation alters the extension behavior. Nonetheless, to ensure consistency, $\texttt{DOM}_{\texttt{Proxy}}$ can be implemented synchronously to block the event loop until the content script worker runs to completion.

7.3 Performance

The runtime performance of browsers is always critical, as it significantly impacts user experiences. Because the content script worker uses the same JavaScript engine, there is no additional overhead running pure JavaScript. However,

since FISTBUMP introduces non-trivial changes in the extension architecture, we measured the performance overhead of DOM operations and extension API calls in FISTBUMP. Figure 6 shows mean runtime of each operation compared to the original browser.

The IPC overhead between $\texttt{DOM}_{\texttt{Proxy}}$ and content script introduces approximately 235 ns latency in a single DOM operation such as reading a content (`dom.read`) or setting the value of an element (`dom.write`). However, common DOM operations such as inserting elements (`dom.insert`) consist of multiple operation without side effects. With batch processing, these operations can be processed with a single IPC, effectively reducing the overhead to approximately 7 ns (13 %).

We saw approximately 28 % performance improvement in calls to basic extension APIs such as retrieving metadata (`api.metadata`), as the extension process can process the request directly. We also observed approximately 87 % improvement in extension messaging (`message.cs/ep`) as it is processed in the extension process rather than via IPC. However, there is approximately 9 % overhead in storage access (`storage.get/set`), presumably because the content script worker has to go through the background page to access the extension storage.

As a result, FISTBUMP shows up to 13 % runtime overhead in extensions with heavy DOM or storage operations and may show performance improvement in message-heavy extensions.

Memory. To measure the memory overhead, we took a snapshot of memory using Chrome DevTools. $\texttt{DOM}_{\texttt{Proxy}}$ and a content script worker add approximately 3.4 MB memory overhead with additional 1.2 KB each time the delegate object and the corresponding reference is created. We observed references of delegate objects that are no longer used are successfully garbage-collected, when the memory pressure is high.

8 Discussion

Extension Vulnerabilities due to Browser Implementations. During our research, we found several vulnerabilities on the browser extension implementation. For example, in Chrome, Firefox, and Safari, $\texttt{Attacker}_{\texttt{RW}}$ could spoof a message from or access the storage of extension that has not injected a content script. This was an independently known issue in Chrome, and they have deployed the ContentScriptTracker in Chrome 103 for message passing and 105 for storage [13, 15]. Safari has confirmed and fixed the issue, and assigned CVE-2022-32784 [2]. The issue is confirmed by Firefox but not yet fixed.

In Chrome, Firefox, and Safari, $\texttt{Attacker}_{\texttt{RW}}$ could also spoof the URL or origin of the sender. This was also an independently known issue in Chrome but not yet fixed. The issue has been confirmed by Firefox but not yet fixed. Firefox also does not provide the origin of the sender, and we submitted a patch to add support for the origin. The issue has been confirmed and fixed by Safari, and assigned with CVE-2022-32784 [2].

In Chrome and Firefox, there are several instances $\texttt{Attacker}_{\texttt{RW}}$ could send message to other extension components which content scripts normally cannot send message to. Finally, we found several bugs where the sender information is not available or incorrect (some bugs were known issues) [12, 14, 16].

Responsible Disclosure and Vendor Response. We responsibly disclosed our findings to extension developers and browser vendors through the vulnerability reporting process or email. However, many extension developers did not acknowledge or fix the issue. In these cases, we reported the issue to Chrome Web Store. Furthermore, some cases were impossible to fix, and we learned that most extension developers do not have (or do not need) deep understanding of the browser security architecture. This supports our analysis in §3 and rationalizes our approach to redesign the extension architecture in §5, which satisfies the security requirements by design.

Motivated by our findings, the Chromium team is discussing to limit the access to extension storage from content scripts in their next extension API version. We note that compared to FISTBUMP, this approach only mitigates the vulnerabilities due to Security Req. 2.

Large-Scale Analysis. We find that 18,432 (about 15 %) out of collected extensions insert a content script to all pages and use the messaging or storage API. We could not automate the analysis as JavaScript supports dynamic function invocation, and files are usually bundled and minimized, making static code analysis difficult. Furthermore, understanding high-level functionality is needed to properly identify vulnerabilities. We believe a large-scale analysis would help identify more vulnerabilities and leave it for future work.

Potential Bypass Attacks against FISTBUMP. FISTBUMP relies on the process isolation boundary provided by the browser and operating system. Such cross-process or kernel attacks are out of scope of this paper, and FISTBUMP must be combined with mitigation on the hardware and OS level.

Improving Performance of FISTBUMP JavaScript engine and rendering engine have multiple optimizations for DOM operations. However, FISTBUMP separates DOM code from where the operations actually happen, reducing opportunities for optimizations. Making JavaScript engine and rendering engine aware of $\texttt{DOM}_{\texttt{Proxy}}$ and optimizing them for $\texttt{DOM}_{\texttt{Proxy}}$ may improve performance.

9 Related work

The security of browser plugins and extensions has been major concern in web security. Research works can be classified into categories: (i) protecting browser from malicious extension and (ii) protecting browser from web pages exploiting benign-but-buggy extensions. Both goals are orthogonal, but protecting from malicious extension may also help protecting from benign-but-buggy extensions, as if what the extension can do is limited in the first place, the compromised extension is also limited.

Earlier works focused on execution monitoring. Janus [25] proposed a sandbox environment for browser helper applications. Louw et al. [57] proposed code integrity check and runtime policy monitoring for Firefox add-ons. Many research prototype browsers including OP [27], Gazelle [61], IBOS [56], and OP2 [28] also proposed isolating web principals using the OS process domain.

Barth et al. [3] found most Firefox extensions request excess permissions and designed the Chrome extension architecture with the PoLP and privilege separation. However, Felt et al. [22], Guha et. al [29], and Carlini et al. [6] showed many extensions still request excess permissions, rendering privilege separation useless and increasing the attack surface. They concluded that privilege separation is rarely needed (but effective) and developers accidentally or intentionally make privilege separation ineffective, which supports our finding.

Guha et. al [29] proposed IBEX, a static verification and fine-grained access control using Datalog for provably secure extension.

Calzavara et al. [5] performed a formal analysis on privilege escalation via message passing interface. Some [54] and Fass et al. [21] performed data-flow analysis to detect message flows that attackers can exploit to elevate their privilege. However, they focused messaging interfaces which normal web page can access, not considering renderer attacker. To the best of our knowledge, our work is the first to comprehensively analyze threats against real-world extensions by renderer attackers.

Furthermore, since the extension storage was introduced fairly recently in 2014, to the best of our knowledge, our work is the first to analyze the security implication of the extension storage.

Extension Fingerprinting. There are also several works on fingerprinting, i.e., identifying which extensions are installed. Sjösten et al. [53], Sanchez-Rola et al. [51], and Gulyás et al. [30] used web accessible resources (WAR) to detect the presence of specific browser extensions. XHOUND [55] and Laperdrix et al. [36] proposed using DOM modifications and stylesheets injected by extension, respectively. Carnus [32] suggested a behavior-based fingerprinting by monitoring communication patterns.

Identifying which extensions are installed would help launch the attack described in our paper, as it allows targeted attacks.

Principle of Least Privilege. PoLP and privilege separation are fundamental principle in software engineering [50]. The web browser is analogous to operating system where the browser process is kernel, web pages are normal application and extensions are privileged application or kernel extensions.

10 Conclusion

This paper identified the design issues of the current extension architecture, which imposes strict security requirements from extension developers. We further demonstrated the criticality of these issues through analyzing popular extensions, which discovered 59 vulnerabilities from 40 extensions. Recognizing the pressing security needs on this problem, we further present FISTBUMP, the new extension architecture to eliminate all such vulnerabilities by design.

Acknowledgments

The authors would like to thank our anonymous reviewers and shepherd for their insightful and valuable feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). This work was supported by National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] Apple Developer Documentation. Safari app extensions. URL https://developer.apple.com/documentation/safariservices/safari_app_extensions.

[2] Apple Support. About the security content of safari 15.6, 2022. URL https://support.apple.com/en-us/HT213341.

[3] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010. URL https://www.ndss-symposium.org/ndss2010/protecting-browsers-extension-vulnerabilities/.

[4] D. Callahan. The "why" of electrolysis, 2016. URL https://blog.mozilla.org/addons/2016/04/11/the-why-of-electrolysis/.

[5] S. Calzavara, M. Bugliesi, S. Crafa, and E. Steffinlongo. Fine-grained detection of privilege escalation attacks on browser extensions. In J. Vitek, editor, *Programming Languages and Systems*, pages 510–534, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8.

[6] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 97–111, Bellevue, WA, Aug. 2012. USENIX Association. ISBN 978-931971-95-9. URL

https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/carlini.

[7] Chrome Developers. Sandbox, . URL https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md.

[8] Chrome Developers. Content scripts, . URL https://developer.chrome.com/docs/extensions/mv3/content_scripts/.

[9] Chrome Developers. Message passing, . URL https://developer.chrome.com/docs/extensions/mv3/messaging/.

[10] Chrome Developers. chrome.storage, . URL https://developer.chrome.com/docs/extensions/reference/storage/.

[11] Chrome Developers. Cross-origin xmlhttprequest, . URL https://developer.chrome.com/docs/extensions/mv3/xhr/.

[12] Chromium Bug Tracker. Issue 626926: sender.url is undefined when tabs.sendmessage sends to an extension page, 2016. URL https://crbug.com/626926.

[13] Chromium Bug Tracker. Issue 982361: Compromised web renderer should be unable to spoof messagesender.id if it never run a content script from the given extension, 2019. URL https://crbug.com/982361.

[14] Chromium Bug Tracker. Issue 1050254: Messagesender.origin might not be available in messages from service workers, 2020. URL https://crbug.com/1050254.

[15] Chromium Bug Tracker. Issue 1183604: Compromised web renderer that *hasn't* run any content scripts can spoof chrome.storage (and other api calls) for any extension, 2021. URL https://crbug.com/1183604.

[16] Chromium Bug Tracker. Issue 1197803: Messages sent from an extension context have an incorrect ‘"null"‘ origin, 2021. URL https://crbug.com/1197803.

[17] Chromium Contributors. chrome/browser/extensions/content_script_apitest.cc. URL https://source.chromium.org/chromium/chromium/src/+/refs/tags/101.0.4951.41:chrome/browser/extensions/content_script_apitest.cc.

[18] cvedetails.com. Google chrome : Security vulnerabilities published in 2022, 2022. URL https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-15031/year-2022/Google-Chrome.html.

[19] S. DeVaney. Firefox's most popular and innovative browser extensions of 2021, 2021. URL https://addons.mozilla.org/blog/firefoxs-most-popular-innovative-browser-extensions-of-2021/.

[20] S. DeVaney. The pandemic changed everything – even the way we use browser extensions, 2022. URL https://addons.mozilla.org/blog/the-pandemic-changed-everything-even-the-way-we-use-browser-extensions/.

[21] A. Fass, D. F. Somé, M. Backes, and B. Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 1789–1804, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384544. doi: 10.1145/3460120.3484745. URL https://doi.org/10.1145/3460120.3484745.

[22] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *2nd USENIX Conference on Web Application Development (WebApps 11)*, Portland, OR, June 2011. USENIX Association. URL https://www.usenix.org/conference/webapps11/effectiveness-application-permissions.

[23] Firefox Site Compatibility. Plug-in support has been dropped other than flash. URL https://www.fxsitecompat.com/en-CA/docs/2016/plug-in-support-has-been-dropped-other-than-flash/.

[24] A. Gakhokidze. Introducing firefox's new site isolation security architecture, 2021. URL https://hacks.mozilla.org/2021/05/introducing-firefox-new-site-isolation-security-architecture/.

[25] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, page 1, USA, 1996. USENIX Association.

[26] Google Inc. Domato, 2017. URL https://github.com/googleprojectzero/domato.

[27] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 402–416, 2008. doi: 10.1109/SP.2008.19.

[28] C. Grier, S. Tang, and S. T. King. Designing and implementing the op and op2 web browsers. *ACM Trans. Web*, 5(2), may 2011. ISSN 1559-1131. doi: 10.1145/1961659.1961665. URL https://doi.org/10.1145/1961659.1961665.

[29] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *2011 IEEE Symposium on Security and Privacy*, pages 115–130, 2011. doi: 10.1109/SP.2011.36. URL https://ieeexplore.ieee.org/document/5958025.

[30] G. G. Gulyas, D. F. Some, N. Bielova, and C. Castelluccia. To extend or not to extend: On the uniqueness of browser extensions and web logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, WPES'18, pages 14–27, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359894. doi: 10.1145/3267323.3268959. URL https://doi.org/10.1145/3267323.3268959.

[31] J. Jang-Jaccard and S. Nepal. A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences*, 80(5):973–993, 2014. ISSN 0022-0000. doi: https://doi.org/10.1016/j.jcss.2014.02.005. URL https://www.sciencedirect.com/science/article/pii/S0022000014000178. Special Issue on Dependable and Secure Computing.

[32] S. Karami, P. Ilia, K. Solomos, and J. Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *NDSS*, 2020. URL https://www.ndss-symposium.org/ndss-paper/carnus-exploring-the-privacy-threats-of-browser-extension-fingerprinting/.

[33] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002. URL https://ieeexplore.ieee.org/document/8835233.

[34] M. Kosaka. Inside look at modern web browser (part 1), 2018. URL https://developer.chrome.com/blog/inside-browser-part1/.

[35] A. Laforge. Moving forward from chrome apps. URL https://blog.chromium.org/2020/01/moving-forward-from-chrome-apps.html.

[36] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2507–2524. USENIX Association, Aug. 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/laperdrix.

[37] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/

usenixsecurity18/presentation/lipp.

[38] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012. URL https://www.ndss-symposium.org/ndss2012/ndss-2012-programme/chrome-extensions-threat-analysis-and-countermeasures/.

[39] MDN Web Doc. Background scripts, . URL https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Background_scripts.

[40] MDN Web Doc. Same-origin policy, . URL https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[41] MDN Web Doc. Webextensions. Technical report, . URL https://developer.mozilla.org/ko/docs/Mozilla/Add-ons/WebExtensions.

[42] M. Y. Mo. The chromium super (inline cache) type confusion, 2022. URL https://github.blog/2022-06-29-the-chromium-super-inline-cache-type-confusion/.

[43] M. Moroz and S. Glazunov. Analysis of uxss exploits and mitigations in chromium. Technical report, 2019.

[44] C. Morris and J. Rossi. A break from the past, part 2: Saying goodbye to activex, vbscript, attachevent... URL https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent/.

[45] Mozilla Security Blog. Mitigations landing for new class of timing attack. URL https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/.

[46] K. Needham. The future of developing firefox add-ons. URL https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/.

[47] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813708. URL https://doi.org/10.1145/2810103.2813708.

[48] C. Reis. Multi-process architecture, 2008. URL https://blog.chromium.org/2008/09/multi-process-architecture.html.

[49] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/reis.

[50] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. doi: 10.1109/PROC.1975.9939.

[51] I. Sanchez-Rola, I. Santos, and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 679–694, Vancouver, BC, Aug. 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/sanchez-rola.

[52] J. Schuh. Saying goodbye to our old friend npapi. URL https://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html.

[53] A. Sjösten, S. Van Acker, and A. Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 329–336, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450345231. doi: 10.1145/3029806.3029820. URL https://doi.org/10.1145/3029806.3029820.

[54] D. F. Somé. Empoweb: Empowering web applications with browser extensions. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 227–245, 2019. doi: 10.1109/SP.2019.00058. URL https://ieeexplore.ieee.org/document/8835286.

[55] O. Starov and N. Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 941–956, 2017. doi: 10.1109/SP.2017.18. URL https://ieeexplore.ieee.org/document/7958618.

[56] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 17–31, USA, 2010. USENIX Association.

[57] M. Ter Louw, J. S. Lim, and V. N. Venkatakrishnan. Extensible web browser security. In *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '07, page 1–19, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540736134. doi: 10.1007/978-3-540-73614-1_1. URL https://doi.org/10.1007/978-3-540-73614-1_1.

[58] The Chromium Authors. What are extensions?, 2013. URL https://developer.chrome.com/docs/extensions/mv3/overview/.

[59] The Chromium Projects. Site isolation. URL https://sites.google.com/a/chromium.org/dev/Home/chromium-security/site-isolation.

[60] J. Wagner. Trustworthy chrome extensions, by default, 2018. URL https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html.

[61] H. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, H. Venter, and S. King. The multi-principal os construction of the gazelle web browser. Technical Report MSR-TR-2009-16, February 2009. URL https://www.microsoft.com/en-us/research/publication/the-multi-principal-os-construction-of-the-gazelle-web-browser/. MSR Technical Report.

[62] S. Weinig, M. Stachowiak, D. Bates, S. Fraser, A. Roben, A. Kling, and C. A. L. Perez. Webkit2, 2010. URL https://trac.webkit.org/wiki/WebKit2.

[63] Y. Weiss and E. Kitamura. Aligning timers with cross origin isolation restrictions. URL https://developer.chrome.com/blog/cross-origin-isolated-hr-timers/.

[64] A. Zeigler. IE8 and loosely-coupled IE (LCIE), 2008. URL https://learn.microsoft.com/en-us/archive/blogs/ie/ie8-and-loosely-coupled-ie-lcie.

[65] R. Zhao, C. Yue, and Q. Yi. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1384–1394, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee. ISBN 9781450334693. doi: 10.1145/2736277.2741134. URL https://doi.org/10.1145/2736277.2741134.

A List of Analyzed Extensions

Table 2 lists analyzed extensions on Chrome Web Store.

Extension ID	Name	Version	Users
aapbdbdomjkkjkaonfhkkikfgjllcleb	Google Translate	2.0.12	10M+
bgnkhhnnamicmpeenaelnjfhikgbkllg	AdGuard AdBlocker	4.0.161	10M+
bmnlcjabgnpnenekpadlanbbkooimhnj	Honey	14.8.0	10M+
cfhdojbkjhnklbpkdaibdccddilifddb	Adblock Plus - free ad blocker	3.12	10M+
cjpalhdlnbpafiamejdnhcphjbkeiagm	uBlock Origin	1.42.4	10M+
cmedhionkhpnakcndndgjdbohmhepckk	Adblock for Youtube™	5.1.7	10M+
dhdgffkkebhmkfjojejmpbldmpobfkfo	Tampermonkey	4.16	10M+
ecnphlgnajanjnkcmbpancdjoidceilk	Kami for Google Chrome™	2.0.15049	10M+
efaidnbmnnnibpcajpcglclefindmkaj	Adobe Acrobat: PDF edit & convert & sign tools	15.1.3.10	10M+
gighmmpiobklfepjocnamgkkbiglidom	AdBlock – best ad blocker	4.44.0	10M+
hdokiejnpimakedhajhdlcegeplioahd	LastPass: Free Password Manager	4.92.0.1	10M+
inoeonmfapjbbkmdafoankkfajkcphgd	Read&Write for Google Chrome™	2.0.1	10M+
inomeogfingihgjfjlpeplalcfajhgai	Chrome Remote Desktop	1.5	10M+
jgfbgkjjlonelmpenhpfeeljjlcgnkpe	ClassLink OneClick Extension	10.6	10M+
jlhmfgmfgeifomenelglieieghnjghma	Cisco Webex Extension	1.17.0	10M+
kbfnbcaeplbcioakkpcpgfkobkghlhen	Grammarly for Chrome	14.1056.0	10M+
mmeijimgabbpbgpdklnllpncmdofkcpn	Screencastify - Screen Video Recorder	2.67.0.4291	10M+
nkbihfbeogaeaoehlefnkodbefgpgknn	MetaMask	10.12.4	10M+
nopfnnpnopgmcnkjchnlpomggcdjfepo	Clever	1.17.1	10M+
oocalimimngaihdkbihfgmpkcpnmlaoa	Netflix Party is now Teleparty	3.4.0	10M+
ppnbnpeolgkicgegkbkbjmhlideopiji	Windows Accounts	1.0.6	10M+
pbjikboenpfhbbejgkoklgkhjpfogcam	Amazon Assistant for Chrome	10.2107.7.11654	8M+
caljgklbbfbcjjanaijlacgncafpegll	Avira Password Manager	2.18.5.3877	5M+
fdjamakpfbbddfjaooikfcpapjohcfmg	Dashlane - Password Manager	6.2212.2	5M+
admmjipmmciaobhojoghlmleefbicajg	Norton Password Manager	7.5.1.48	4M+
ndjpnladcallmjemlbaebfadecfhkepb	Office	2.2.9	4M+
aeblfdkhhhdcdjpifhhbdiojplfjncoa	1Password – Password Manager	2.3.2	2M+
bfnaelmomeimhlpmgjnjophhpkkoljpa	Phantom	22.3.29	2M+
fnjhmkhhmkbjkkabndcnnogagogbneec	Ronin Wallet	1.7.0	2M+
hnmpcagpplmpfojmgmnngilcnanddlhb	Windscribe - Free Proxy and Ad Blocker	3.4.0	2M+
mlomiejdfkolichcflejclcbmpeaniij	Ghostery – Privacy Ad Blocker	8.6.3	2M+
nngceckbapebfimnlniiiahkandclblb	Bitwarden - Free Password Manager	1.57.0	2M+
fhbohimaelbohpjbbldcngcnapndodjp	Binance Wallet	2.12.2	1M+
hnfanknocfeofbddgcijnmhnfnkdnaad	Coinbase Wallet extension	2.12.2	1M+
kacljcbejojnapnmiifgckbafkojcncf	Ad-Blocker	1.5	1M+
lgblnfidahcdcjddiepkckcfdhpknnjh	Fair AdBlocker	1.524	1M+
pkehgijcmpdhfbdbbnkijodmdjhbjlgp	Privacy Badger	2021.11.23.1	1M+
ohahllgiabjaoigichmmfljhkcfikeof	AdBlocker Ultimate	3.7.16	0.9M+
aiifbnbfobpmeekipheeijimdpnlpgpp	Terra Station Wallet	2.7.0	0.7M+
pnlccmojcmeohlpggmfnbbiapkmbliob	RoboForm Password Manager	9.3.2.0	0.6M+
bfogiafebfohielmmehodmfbbebbbpei	Keeper® Password Manager & Digital Vault	16.4.0	0.5M+
dmkamcknogkgcdfhhbddcghachkejeap	Keplr	0.10.0	0.5M+
ibnejdfjmmkpcnlpebklmnkoeoihofec	TronLink	3.26.4	0.5M+
ffnbelfdoeiohenkjibnmadjiehjhajb	Yoroi	4.11.500	0.4M+
fhmfendgdocmcbmfikdcogofphimnkno	Sollet	0.3.1	0.2M+
jblndlipeogpafnldhgmapagcccfchpi	Kaikas	1.10.1	0.2M+

Table 2: List of analyzed extensions on Chrome Web Store.