Chrowned by an Extension: Abusing the Chrome DevTools Protocol through the Debugger API

José Miguel Moreno
Universidad Carlos III de Madrid
Madrid, Spain
josemore@pa.uc3m.es

Narseo Vallina-Rodriguez
IMDEA Networks Institute
Leganés, Spain
narseo.vallina@imdea.org

Juan Tapiador
Universidad Carlos III de Madrid
Madrid, Spain
jestevez@inf.uc3m.es

***Abstract*—The Chromium open-source project has become a fundamental piece of the Web as we know it today, with multiple vendors offering browsers based on its codebase. One of its most popular features is the possibility of altering or enhancing the browser functionality through third-party programs known as browser extensions. Extensions have access to a wide range of capabilities through the use of APIs exposed by Chromium. The Debugger API—arguably the most powerful of such APIs—allows extensions to use the Chrome DevTools Protocol (CDP), a capability-rich tool for debugging and instrumenting the browser. In this paper, we describe several vulnerabilities present in the Debugger API and in the granting of capabilities to extensions that can be used by an attacker to take control of the browser, escalate privileges, and break context isolation. We demonstrate their impact by introducing six attacks that allow an attacker to steal user information, monitor network traffic, modify site permissions (*e.g.*, access to camera or microphone), bypass security interstitials without user intervention, and change the browser settings. Our attacks work in all major Chromium-based browsers as they are rooted at the core of the Chromium project. We reported our findings to the Chromium Development Team, who already fixed some of them and are currently working on fixing the remaining ones. We conclude by discussing how questionable design decisions, lack of public specifications, and an overpowered Debugger API have contributed to enabling these attacks, and propose mitigations.**

1. Introduction

Modern browsers expose powerful APIs [25], [36] to enable the development of third-party browser extensions [35], [54]. These are small programs, built and released by third-party developers, that extend or enhance the default features offered by the browser like cookie management and ad-blocking. Because of their potential for abuse, these APIs are permission-protected and not all of them are automatically granted by default to extensions [25]. In the case of Chromium, its permission system is quite limited and does not offer a comprehensive set of protections as implemented, for example, in Android [4]. Anecdotal evidence has already shown how these APIs have been abused for nefarious purposes [5], [17].

One overlooked Chromium feature is the `debugger` permission [37], which grants access to a limited version of the powerful Chrome DevTools Protocol (CDP), a core Chromium component for debugging and instrumenting the browser through a command passing interface. CDP is widely used for running End-to-End (E2E) tests on web-based applications through popular tools like Selenium, Puppeteer and Playwright, and for building crawlers. CDP exposes a WebSocket server to which external applications can connect to. Chromium extensions may also communicate with this component using the Debugger API, which is protected by the `debugger` permission. The Debugger API is a general substitute of virtually any other extension API as it grants total control over tabs, windows and critical browser resources. These powerful capabilities are expected to be found in a debugging tool, but are also an obvious candidate for abuse if they are insecurely exposed to potentially malicious actors.

Despite the risks of granting third-party extensions access to such a powerful component, no previous work has systematically analyzed the robustness of the Debugger API implementation and its security implications. In fact, Chromium's Debugger API is already being used by at least 434 extensions published on the Chrome Web Store according to a permission measurement that we performed in June 2022. Furthermore, no official specification detailing the design and purposes of this component can be publicly found. In this paper, we describe the results of a systematic security analysis done over the Debugger API and related components in the Chromium codebase. Our analysis focuses on finding violations of a set of security requirements that we derive from Chromium's CRX API Security Checklist [13]. Through a systematic code review, we find multiple vulnerabilities that can be exploited by a third-party extension to (i) circumvent the permission model to elevate privileges and gain control over more capabilities than expected, including key browser features; and (ii) break the isolation principles implemented by Chromium to prevent an attacker from accessing third-party targets. Specifically, we present the following six attacks:

- *Listing active targets* (§4.1). This is a privacy attack that can be used to track the list of active running extensions and user's browsing history, including URLs visited in an incognito window.
- *Running on regular tabs* (§4.2). This attack allows an extension to steal any user information contained inside most browser tabs, including those in an incognito window, and evaluate arbitrary code inside them to alter their behavior.

- *Running on security interstitial tabs* (§4.3). We show how extensions can abuse the Debugger API to modify the contents of interstitial messages, including critical security messages such as TLS error dialogs. It also can be used to skip interstitials completely with no user interaction.
- *Running on WebUI tabs* (§4.4). This attack extends the capabilities of the second attack (§4.2). It allows extensions to run on internal tabs (*e.g.*, settings page), thus allowing the modification of critical browser settings, including security ones.
- *Running on other extensions* (§4.5). This attack allows extensions to debug any other running extension to steal sensitive information (such as plaintext passwords and credit card details stored in password managers) or modify its normal operation (*e.g.*, change the receiving wallet of a cryptocurrency transaction).
- *Attaching to the browser target* (§4.6). This attack interacts with a special high-privileged target from the CDP that allows it to run on virtually any tab or extension and take full control of the browser.

We confirm the feasibility of the proposed attacks on every major Chromium-based browser, including more privacy-focused solutions like Brave and Ungoogled Chromium. All of our attacks share the same root cause, which we attribute to a mix of questionable design decisions and excessive functionality granted to extensions through CDP and the `debugger` permission. Overall, our attacks exemplify the inherent tensions when reconciling a debugging tool, which by definition has powerful capabilities, with an API exposed to untrusted third-party code. This is a challenging design area and we discuss the causes, impact, and potential mitigations in §5.

Coordinated disclosure. We disclosed all our findings to the Chromium Development Team along with a practical Proof-of-Concept for each attack. The attack from §4.1 and the vulnerable behavior from §4.2 were reported on March 2022[1] and acknowledged to be a bug by Google. It was fixed in May 2022, when it landed in Chromium Stable 102. The attack from §4.3 was reported on November 2021[2] and was fixed in Chromium Stable 103,[3] when it was assigned CVE-2022-2164. The attack from §4.4 was reported on December 2021,[4] marked as a duplicate and merged with §4.5. As far as we know, this is the only bug that was previously known by the Chromium Team. The attack from §4.5 was reported on December 2021[5] and is still awaiting a fix from the Chromium Development Team. The attack from §4.6 was reported on December 2021[6] and again for a different vulnerability on March 2022.[7] The first report was marked as duplicated and merged with §4.5. The second report, while initially flagged as a "high-severity bug," was then classified as "not-a-security-bug." We discuss this point in detail in §5.

1. Reported in https://crbug.com/1301950.
2. Reported in https://crbug.com/1268445.
3. See https://crrev.com/c/3594083.
4. Reported in https://crbug.com/1276500.
5. Reported in https://crbug.com/1276497.
6. Reported in https://crbug.com/1276503.
7. Reported in https://crbug.com/1301966.

Research artifacts. Proof-of-Concepts for all our attacks are available at https://github.com/josemmo/chrowned. All of them use the Manifest V3 specification to account for the approaching phase out of Manifest V2 [39].

2. Background

Chromium is an open-source web browser project mostly developed and maintained by Google. Its codebase is the foundation for the Google Chrome browser, and it is widely reused by many other popular browsers, including Microsoft Edge, Brave, Samsung Internet or Opera. This section provides technical background on Chromium extensions (§2.1), the Chrome DevTools Protocol (§2.2), and Chromium's Debugger API (§2.3).

2.1. Chromium Extensions

Chromium extensions are programs consisting of a set of files and assets similar to those found in traditional web applications. They are typically packaged into a custom file format known as "CRX" and then submitted to the Chrome Web Store—the official distribution platform from Google—for distribution. CRX packages are signed by extension developers using public-key cryptography to provide integrity and authenticity. As §3 explains, extensions can also be distributed outside the Chrome Web Store without the need for signatures through *sideloading* (*e.g.*, using ZIP archives).

Extensions must have a mandatory JSON file named `manifest.json`. The manifest file contains basic metadata about the extension (*e.g.*, name, version) and other properties, such as a background script (for running tasks or receiving events independently of any opened tab) and content scripts (for injecting code onto pages that match a given URL).

Chromium implements several security features to protect its users from malicious extensions. One of such features is isolation, which prevent websites and extensions alike from running code on other execution contexts outside their scope. Additionally, extensions must declare permissions in their manifest file to access sensitive resources (like the browsing history) or break isolation and run on tabs with a given URL (*i.e.*, through the use of host permissions). Some permissions are considered more dangerous than others because of the potential harm that their abuse or misuse can cause to users. For this reason, an extension containing one or more of these permissions will display a prompt during install with brief warnings for each of them [27].

Chromium's permission model still presents important shortcomings, many of which have already been fixed on platforms like Android [4]. For example, once an extension is installed, it keeps unrestricted access to all declared permissions and there is no simple way to revoke access afterwards. Another limitation is transparency: the user warnings shown in the install dialog communicate what a permission enables at a technical level but it does not clearly convey how impactful it can be for users' privacy and security if abused.

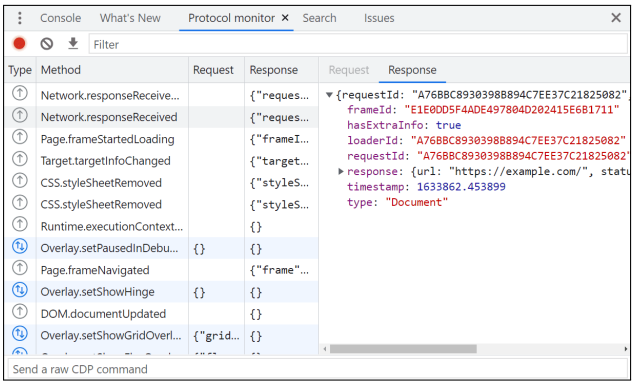

Figure 1. Protocol Monitor tab from DevTools UI window.

2.2. Chrome DevTools Protocol

Chromium has a built-in component known as the Chrome DevTools Protocol (CDP) [29] which allows web developers to instrument and debug the browser to the fullest extent. Although this feature can be abused as we demonstrate in this paper, it offers useful features for software developers and researchers. Selenium [69], Puppeteer [51] and Playwright [24] are three tools powered by CDP which are widely used for End-to-End (E2E) testing of web applications or automating web scraping.

Any application using the CDP communicates with the browser by sending JSON-encoded messages to a WebSocket endpoint hosted by the Chromium process, which is also used to receive events triggered by the browser. For security and performance reasons, this endpoint has to be enabled by adding the `--remote-debugging-port` command line flag when launching the browser, followed by the port number the CDP server will be bound to (port 9222 by convention) [29].

The Chrome DevTools UI [8] uses the CDP under the hood to inspect and debug a target (*e.g.*, a tab). Messages exchanged by the browser (*host*) and the DevTools UI (*client*) can too be inspected using the "Protocol monitor" drawer tab (see Figure 1). This feature is initially turned off but it can be manually enabled from the "Experiments" section in the DevTools UI settings menu.

To account for the isolation between tabs, background pages, service workers, and other worlds from the browser, CDP introduces the concept of *target*, which can be any of the former. When a client wants to interact with a target, it first has to *attach* to it using the `Target.attachToTarget` command. It will then receive a session ID to forward CDP commands to the desired target. This ensures that, for example, if we want to evaluate JavaScript code on a particular tab, it will get executed only on our target and nowhere else. By default, when opening a WebSocket to the CDP server, the client attaches to what is known as the *browser target*. This is a special type of target that can list targets (*i.e.*, `Target.getTargets`), attach to active ones and perform other instrumentation actions affecting the entire browser.

The number of capabilities the protocol offers varies from version to version. In a nutshell, the CDP can perform virtually any action a user will be able to achieve with manual interaction (*e.g.*, clicking on a link or button). Additionally, it gives finer control over browser internals that a regular user cannot access, such as traffic interception or placing debugging breakpoints. Specific capabilities provided by CDP include: (i) evaluate JavaScript code on the global object of a target (*e.g.*, `window`); (ii) traverse, manipulate and add event listeners to the DOM tree of a target; (iii) intercept and modify network flows; (iv) list, edit, create and delete any entry from the browser cookie store; (v) grant or deny site permissions (*e.g.*, camera, geolocation) without requesting user consent; (vi) take image screenshots of focused targets; and (vii) capture profiling snapshots of CPU, GPU and memory, as well as metrics like code coverage [7]. While most of these capabilities are truly sensitive, this is expected from a development and debugging tool, and is not necessarily concerning by itself. However, carelessly exposing these capabilities to other parts of the browser (*e.g.*, websites, extensions) might be a source of vulnerabilities.

TABLE 1. SUMMARY OF FEATURES PROVIDED THROUGH CDP COMMANDS AND THEIR EQUIVALENTS IN EXTENSION APIS.

Feature	CDP domain	Extension API
Evaluate JS code	`Runtime`	`scripting`
Manipulate cookie store	`Network`	`cookies`
Intercept network traffic	`Network`	`webRequest`
Take screenshots	`Page`	`desktopCapture`
Modify site permissions	`Browser`	N/A
Place breakpoints	`Debugger`	N/A
Record execution trace	`Tracing`	N/A

2.3. The Debugger extension API

Extensions cannot control the command line arguments the browser has been launched with, which is a necessary step to activate the CDP WebSocket endpoint (§2.2). Yet, extensions can still communicate with a version of the CDP with limited capabilities through the use of the Debugger API [37], which is a substitute for virtually any other extension API: by just requesting the `debugger` permission, extensions can perform a wide range of sensitive actions (§2.2) without having to declare any further permissions in the manifest. In addition, it gives extensions access to exclusive functionality (*i.e.*, that no other extension API offers). Table 1 provides a list of noteworthy Debugger API features alongside their counterparts using other extension APIs, if any.

The Debugger API differs from the regular CDP in that it makes critical protocol domains inaccessible. For example, the `Target` domain or even some cherry-picked methods from other partially allowed domains. The rationale behind this decision is presumably to prevent a rogue extension from taking full control of the browser. We note that having a fully instrumented environment is ideal when running E2E tests but it is a dangerous feature for an end-user to have enabled, thus the need for a limited CDP agent for extensions. Given these limitations (mainly the absence of the `Target` domain), an alternate API to list targets and attach to them is needed. Therefore, the Debugger API provides JavaScript bindings to perform these operations. In short, an extension has to follow three steps to instrument a target:

1) List all active targets with `chrome.debugger.getTargets()` to get the

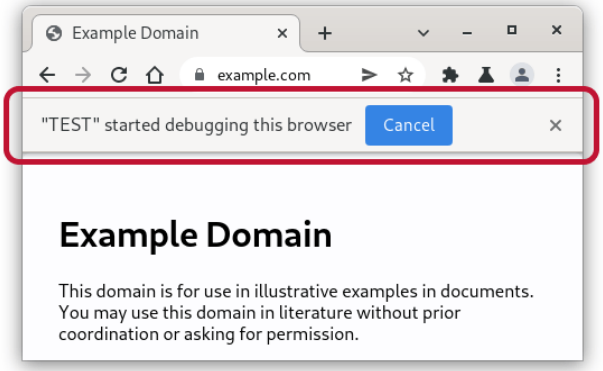

Figure 2. A sample debugger infobar in Chromium highlighted in red.

target, tab or extension ID of the one that will be instrumented.

2) Call `chrome.debugger.attach()` to attach to the desired target. Immediately after attaching, the browser will start showing a notification to inform the user of this event.
3) Use `chrome.debugger.sendCommand()` to send CDP commands to the debugged target and, optionally, add a listener to `chrome.debugger.onEvent` to be notified of CDP events.

The Debugger API imposes limitations on what targets an extension can attach to. While an extension can debug itself, it cannot instrument service workers, background pages or tabs from other extensions, nor it can attach to targets with a URL scheme other than "http://" or "https://". We explore ways of bypassing these security mechanisms in §4.

User awareness. When a debugging extension successfully attaches to a target, Chromium notifies users by rendering an *infobar* across all open browser tabs [74] as shown in Figure 2.[8] This notification will not go away until there are no attached debuggers left. Optionally, a user can click on the cancel button to force all debuggers from an extension to be detached from their respective targets. However, nothing prevents a malicious extension from reattaching to the previous targets immediately after the user cancels the infobar, and this can happen without any user awareness (§4.2).

3. Threat model

Our fist threat model, named Threat Model A (TMA), assumes an attacker who can successfully trick a user into installing a malicious extension that declares the `debugger` permission, thus getting access to the capabilities described in §2.2. This extension may be distributed through official platforms such as the Chrome Web Store in the form of a signed CRX package [61]. Attacks §4.1 and §4.2 assume this threat model.

8. We note that debugger infobars can be completely disabled if Chromium is launched with the command line flag `--silent-debugger-extension-api`.

A second more restrictive threat model, named Threat Model B (TMB), requires users to install extensions through sideloading. Sideloading consists in loading an unpacked extension [31], or drag-and-dropping a CRX package or ZIP file over the extensions management page (*i.e.*, `chrome://extensions`). Because unpacked extensions and ZIP archives lack the cryptographic signature that CRX files have, their contents are not integrity-protected nor their authenticity can be verified. Therefore, they can spoof their ID and impersonate other legitimate extensions. We introduce that feature in §4.4). However, as they are not CRX packages, they cannot be distributed through the official Chrome Web Store. Attacks §4.4, §4.5 and §4.6 use Threat Model B.

We note that sideloading is a feature intended for developers so it requires users to enable "Developer Mode" on their browsers for it to work. However, malicious actors can easily trick users into doing so or even silently run a malicious PowerShell script that automates this process, a technique that has already been observed in the wild [18], [65]. A threat actor that gains execution capabilities outside the browser on a victim's device might prefer sideloading an extension over dropping an executable for ease of development and feature set completeness. Browser extensions that work across different operating systems are easy to develop, and the Debugger API lets attackers evaluate arbitrary JS code and intercept network traffic in cleartext without the hassle of setting up a proxy with a trusted self-signed certificate.

Although TMB is a more restrictive threat model, it is widely accepted as realistic by the community. For example, most numbered vulnerabilities (CVEs) involving a malicious extension assume exactly this model (*e.g.*, [55]–[59]), and it is also found in previous academic works studying malicious browser extensions [20], [45]. Furthermore, sideloading is a common practice to install extensions in regions where the Chrome Web Store is not available and alternative marketplaces have emerged [1]–[3]. One notable example is China. There, sideloading is the only way to install an extension because the Chrome Web Store is not available despite Google Chrome having the largest browser market share [73].

4. Attacks

We next introduce six attacks that exploit design vulnerabilities in the Debugger API and in the logic for granting restricted capabilities to highly privileged extensions. Attacks are grouped by threat model (first TMA, then TMB) and sorted by severity in ascending order.

Methodology. We performed a systematic manual code review process to find issues in the design and implementation of the Debugger API. To assess its security, we defined a set of security requirements that an hypothetically secure extension API must comply with based on the official CRX API Security Checklist [13]. This is a public document used by Chromium developers as a baseline to follow best security practices [12].

To do so, we systematically review the JavaScript bindings exposed by the `chrome.debugger` object [14]. For each binding, we flag its source code and that of the browser components it depends on. Then,

TABLE 2. SUMMARY OF ATTACKS AND THEIR CAPABILITIES. ALL THE ATTACKS WORK IN CHROMIUM BROWSER, GOOGLE CHROME, MICROSOFT EDGE, BRAVE BROWSER, OPERA, VIVALDI AND UNGOOGLED CHROMIUM.

Attack	Theat Model	Opened tabs			Running extensions			Browser instance			
		List	Evaluate JS code*	Intercept traffic	List	Evaluate JS code*	Intercept traffic	Steal browsing cookies	Steal credit cards and passwords	Change settings and flags	Record profiling traces
Listing active targets (§4.1)	TMA	●	○	○	●	○	○	○	○	○	○
Running on regular tabs (§4.2)	TMA	●	◐	◐	●	○	○	●	○	○	○
Running on interstitials (§4.3)	TMB	●	◐	◐	●	○	○	●	○	○	○
Running on WebUI tabs (§4.4)	TMB	○	●†	○	○	○	○	○	●	●	○
Running on extensions (§4.5)	TMB	●	◐	◐	●	●	●	●	○	○	○
Attaching to browser (§4.6)	TMB	●	●	●	●	●	●	●	●	●	●

○ Not capable at all.
◐ Only on regular tabs and interstitials.
● Fully capable of.
* Useful for stealing data from a target and manipulating its behavior.
† Except tabs from incognito windows.

we thoroughly inspect the tainted source code to find violations of the following requirements:

- *SR01 – User Awareness.* Users must be clearly informed of the implications of an extension making use of the Debugger API (*e.g.*, during installation) and notified when an extension is actively using it (*e.g.*, using infobars).
- *SR02 – Isolation.* The Debugger API must respect browser profiles and honor users' privacy choices (*e.g.*, incognito mode) and restrict the scope of extensions using it accordingly.
- *SR03 – Access Control.* The Debugger API must enforce rules to prevent extensions from accessing sensitive resources (*e.g.*, browsing history) or targets (*e.g.*, tabs) outside their supposed reach.
- *SR04 – Spoofing Avoidance.* Extensions should not be able to modify critical parts of the browser UI (*e.g.*, settings page) to stop them from misleading users into performing a given action.

To identify such violations, we looked for security checks or preconditions that the Debugger API should comply with. For instance, to satisfy SR01, a call to display an infobar with a localized message is likely expected when an extension attaches to a target. Once a requirement violation is found, we manually verified its potential exploitability by implementing a prototype extension and testing its effectiveness against the latest Chromium Stable release at the time.

Table 2 summarizes the attacks that we found using this methodology and their impact, and Table 3 shows violations of the above Security Requirements for each attack. The order in which we introduce the attacks goes from the least to the most impactful, starting from merely listing the opened tabs and running extensions (§4.1), then gaining access to evaluate arbitrary code on each of those targets (§4.2, §4.3, §4.4, §4.5), and finally being able to take control of the browser (§4.6). Note that some of our attacks achieve similar results. We include them nonetheless as they are independent from one another (*i.e.*, they are based on different flaws, thus mitigating one attack will not affect the other).

Prevalence. We are able to reproduce our attacks in all major Chromium-based browsers (Google Chrome, Microsoft Edge, Opera and Vivaldi), including those with a special focus on privacy (Brave browser and Ungoogled Chromium [22]). We note that we have not tested these attacks on mobile versions of Chromium (*e.g.*, Google Chrome for Android) as there is no straightforward way of installing extensions in those builds as of this writing. We provide Proofs-of-Concept (PoCs) for all of the attacks in the artifacts associated with this paper for independent validation. These PoCs are fully-commented extensions based on our threat model. They showcase various real-world scenarios that make use of our findings.

TABLE 3. SECURITY REQUIREMENT VIOLATIONS PER ATTACK.

Attack	SR01		SR02	SR03	SR04
	Install-time	Run-time			
§4.1	✗	✗	✗	✗	
§4.2	✗	✗‡	✗	✗	
§4.3	✗	✗‡	✗	✗	✗
§4.4	✗	✗‡		✗	✗
§4.5	✗	✗‡	✗	✗	
§4.6	✗	✗‡	✗	✗	✗

✗ Denotes a violation of a security requirement.
✗‡ Only a violation if infobars have been disabled.

4.1. Listing active targets

The Debugger API is implemented in such a way that extensions have to call the `chrome.debugger.attach()` function to start instrumenting a particular target using the CDP (see §2.3). To indicate the target, its target ID must be passed as a parameter to this function. Yet, extensions can extract a list of running target IDs by calling `chrome.debugger.getTargets()`.

Attack vector. The `chrome.debugger.getTargets()` function lists not only the targets an extension is allowed to attach to (usually tabs with a URL beginning with "http://" or "https://"), but also other targets the extension is not supposed to be capable of debugging. Any extension declaring the `debugger` permission in its manifest can call this function from both content scripts and background pages without requiring any user interaction. This could result in a privacy abuse as it can expose sensitive data such as users' browsing history, thus violating SR03. Being able to access this type of sensitive information raises privacy concerns as it could be linked to user identities (*e.g.*, by also harvesting cookies or other Personal Identifiable Information).

```
const visited = new Set();
setInterval(async () => {
  const targets = await chrome.debugger.getTargets();
  for (const target of targets) {
    if (visited.has(target.url)) {
      // Ignore previously visited URLs
      continue;
    }
    visited.add(target.url);
    console.log({time: Date.now(), url: target.url});
  }
}, 1000);
```

Listing 1. JavaScript code for polling running targets every second.

Impact. Alongside the target ID, the Debugger API also grants access to the URL of the target, the page title and even the favicon URL. A malicious extension can call `chrome.debugger.getTargets()` to easily and accurately monitor the set of opened tabs and the list of websites visited by the user, including those from incognito windows. Note that the malicious extension does not need to be granted permission to run in the former context, thus breaking SR02. Additionally, because service workers and background pages are also listed, an attacker can also get the list of running extensions.[9] Access to installed extensions can be used for improving device fingerprinting techniques, as in some cases is enough to unequivocally identify a user [47], [68], [71]. Since there is no rate-limit to getting the list of targets, an attacker could keep polling this information every few seconds or less to detect changes in enabled extensions or page navigation events, as proposed in Listing 1. An important remark is that SR01 is also violated as no debugger infobar (see §2.3) is shown when calling the `chrome.debugger.getTargets()` method, thus making this attack fully silent. We provide a PoC within the artifacts to detect running extensions that solely uses the Debugger API.

4.2. Running on regular tabs

Regular tabs are the least privileged targets a CDP client can attach to. They use either the "http://" or "https://" scheme. We note that the "ftp://" scheme is unsupported since October 2021 [30], and "file://" is restricted by default and requires the user to manually opt in [26].

Attack vector. Instrumenting a regular tab is trivial. The extension only needs to declare the `debugger` permission in its manifest and then attach to the desired tab using its tab ID or target ID. Both IDs can be known with the methods described in §4.1. Once attached, CDP messages can be exchanged (see §2.3) as long as the extension does not get detached from the target, which can happen programmatically by purposely calling `chrome.debugger.detach()`, or when the tab is closed or changes the URL to a restricted page (*i.e.*, outside the scope of regular tabs). Because there is no finer control over this API, an extension with the `debugger` permission will have unrestricted access to the Debugger API in its entirety. This breaks SR03 as it can be abused to access sensitive resources that otherwise would require additional permissions (*e.g.*, the browser cookie store).

After attaching to a target, an infobar is shown to the user giving the option to force-detach the debugging extension (§2.3). However, during our research we find that extensions can reattach immediately afterwards and that no rate-limit protection exists against this behavior. In practice, to the user this would look like the cancel button of an infobar does nothing, rendering it ineffective due to the immediacy at which extensions can re-attach. In addition, due to an incomplete security check, an extension can also attach to regular tabs from incognito windows, even when it has not been granted permission to do so, which is the default behavior. While trying to attach to an incognito tab by providing the tab ID will result in a *"No tab with given id"* error, no additional input verification is performed when we use its target ID, thus letting the extension debug tabs from incognito windows and violating SR02.

Impact. As mentioned in §2.3, the Debugger API is a general substitute for virtually any other extension API. In an nutshell, with this API, a malicious extension can:

- *Steal sensitive user information.* By being able to manipulate the DOM or evaluate arbitrary JavaScript expressions, the extension can read email addresses, passwords, credit card numbers and other Personal Identifiable Information (PII) already accessible to the regular tab. It is also possible to monitor all network traffic sent by the regular tab to which the extension is attached to for the same purpose. Additionally, the `Network.getAllCookies` command lets extensions exfiltrate any cookie stored in the browser regardless of the URL of the debugged regular tab, thus providing a historical perspective of users' browsing habits and even leaking sensitive data encoded in the cookies as Listing 2 shows.
- *Manipulate runtime behavior.* Apart from stealing data, evaluating arbitrary code on regular tabs is also useful for modifying the UI of a web application or changing its intended behavior. For instance, an attacker may inject a JavaScript file on a banking app to alter the recipient of a wire transfer while hiding this information from the user.
- *Modify site permissions.* The `Browser.grantPermissions` command can grant any site access to privacy-sensitive resources (*e.g.*, microphone) without further user consent, bypassing the need for additional permission in an extension's manifest.

We verify the previous capabilities by creating several extensions that use the Debugger API to achieve different goals. As a PoC, we provide an extension that attaches to tabs from incognito windows, and another one that list the metadata for all cookies stored in the browser.

4.3. Running on security interstitial tabs

Chromium comes with a built-in security feature known as *Safe Browsing* for blocking known phishing

9. Similarly, the Management API [38] provides a method for listing installed extensions, though it requires declaring an additional permission.

```
// Attach to ourselves (any regular tab will do)
const targets = await chrome.debugger.getTargets();
const targetId = targets.find(target => {
  return target.url === document.location.href;
}).id;
await chrome.debugger.attach({targetId}, "1.3");

// Get all cookies
const res = await chrome.debugger.sendCommand(
  {targetId},
  "Network.getAllCookies"
);
console.log(res.cookies);
```

Listing 2. JavaScript code for reading all browser cookies.

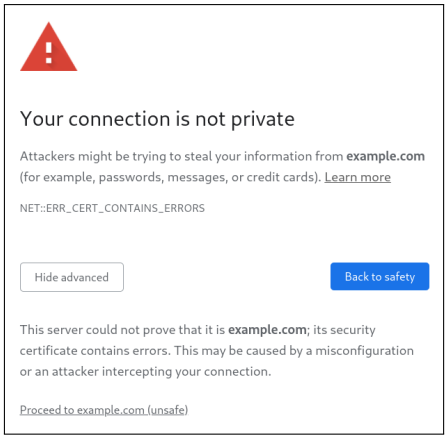

Figure 3. A privacy error interstitial in Chromium.

and malware sites [63]. When users enable this feature, the browser will block navigation to URLs included in a blocklist and instead display an *interstitial* dialog asking the user to confirm a dangerous action (*e.g.*, continue anyway and visit the site) or abort and go back to the previous page. Interstitials are also used for showing SSL/TLS warnings like an expired or invalid certificate as shown in Figure 3, or when connecting to a public network behind a Captive Portal, among other use cases.

Attack vector. Given the sensitive nature of interstitials, these "loud" dialogs are always served by Chromium at `chrome-error://chromewebdata/`, which is a special unreachable URL (*i.e.*, it cannot be visited by typing it on the browser's address bar). Serving such interstitials like so presents a security advantage: because the target is no longer a regular tab (*i.e.*, it has a different scheme), extensions cannot theoretically run on it, thus preventing a malicious actor from, for instance, using the `scripting` extension API to evaluate arbitrary JavaScript code on the page. However, this restriction does not apply to the Debugger API, which can attach to targets with the former unreachable URL. This lack of access control to such a sensitive class of resources enables several attacks that can impersonate or modify a security interstitial.

Impact. Given that extensions can use the Debugger API to attach to a security interstitial tab, an attacker can use this capability to:

- *Impersonate interstitials.* Extensions can evaluate arbitrary code to modify the appearance of these notifications to show a different message or content, therefore modifying the browser UI and violating SR04. Performing this attack is as trivial as querying the DOM nodes we want to modify (*e.g.*, using `document.querySelector`) and then changing their HTML or text contents. A clear malicious use case for this is a phishing attack, given that it can produce an interstitial with the same look and feel of the ones triggered by Chromium to trick the user into performing some dangerous action, like downloading an executable as Figure 4 shows.
- *Skip interstitials.* An attacker can also force the browser to automatically skip interstitials on some websites by making the extension programmatically click on the "Proceed" button or calling `certificateErrorPageController.proceed()`. This could be used by a malicious extension to skip TLS warnings or error messages (*e.g.*, related to certificate validation), which can facilitate TLS man-in-the-middle (MITM) attacks using forged certificates. The same strategy can be applied to sideload malware by skipping Safe Browsing.

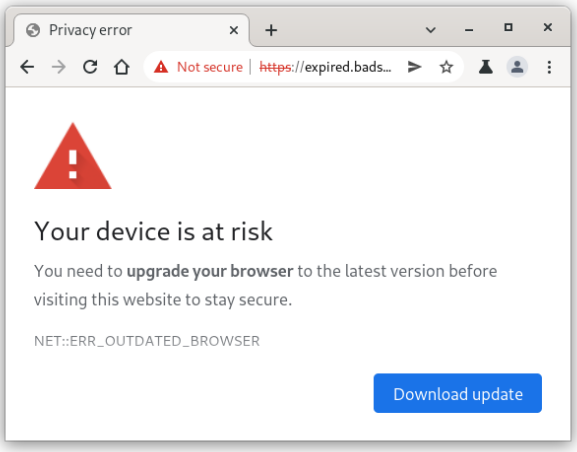

Figure 4. An impersonated security interstitial used to trick the user into downloading a malicious executable under the pretext that a browser update is needed.

To demonstrate this attack, we created a PoC extension that automatically bypasses security interstitials whenever it encounters one.

4.4. Running on WebUI tabs

A vast portion of the User Interface (UI) in Chromium is implemented using web pages. A well-known case are, for example, the browser settings, which can be accessed by navigating to the `chrome://settings` URL. There are well over 70 different URLs using the internal "chrome://" scheme, ranging from the bookmarks menu to the browsing history. Google calls these *Chrome UI* or *WebUI* pages [77]. An almost complete list of WebUI URLs can be found on `chrome://about`.

These pages run in a higher-privileged context with more capabilities than regular tabs, which are provided through (i) internal extension APIs that are granted

```
bool ExtensionDevToolsClientHost::MayAttachToURL(
  const GURL& url,
  bool is_webui
) {
  if (is_webui) {
    return false;
  }
  // [...]
  return true;
}
```

Listing 3. Code to prevent extensions from attaching to WebUI pages (adapted from [14]).

based on the URL; and (ii) message passing communication using Mojo JavaScript bindings [76]. For example, the `chrome://extensions` page, responsible for letting the user manage the installed browser extensions, has access to an internal extension API (*i.e.*, `developerPrivate`) for that very purpose.

For obvious reasons, extensions cannot run on WebUI pages as that would have devastating consequences for the browser's security model. However, on March 2013 the experimental flag `extensions-on-chrome-urls` was added to Chromium to circumvent this limitation and allow extensions that explicitly declared the `chrome://*` permission to access WebUI pages [9]. Later on, in 2018, a security report flagging this issue triggered a response from the Chromium Security Team [43] that limited the capabilities of the flag by blocking extensions using the Debugger API from attaching to WebUI targets regardless of the value of the previous flag as Listing 3 shows.

Attack vector. Not being able to use the Debugger API does not imply that extensions cannot run on WebUI tabs. An extension can still use the `tabs` or `scripting` APIs to evaluate JavaScript expressions when the previous flag is enabled. In fact, the official "Screen Reader" extension [34] has access to the "chrome://" scheme without needing it to be enabled at all. This extension is an accessibility tool targeted towards users with visual impairments that can read aloud the contents, buttons, menus and other elements of a page. We find other instances of security concessions made in favor of usability in platforms such as Android, where it has proven to be abusable by attackers [21], [44]. Because a significant part of the browser UI is implemented using web pages, the extension has to be able to interact with WebUI tabs too to read its contents. To accomplish it, the Chromium developers made an exception for this particular extension and granted it privileges to run on the former higher-privileged targets without requiring any further flags. This backdoor access is implemented by defining an allowlist solely containing the Screen Reader extension ID, which is queried every time an extension wants to run on a given URL to determine if it is restricted or not (see Listing 4).

Using a hardcoded allowlist for this purpose is not a robust nor secure design choice as the ID of an extension can be impersonated. A *clone* extension with the same ID as the Screen Reader extension can take advantage of its privileged position within the browser and acquire its additional capabilities. Unfortunately, this is rather trivial to perform. Figure 5 shows how Chromium calculates extension IDs by taking a DER-encoded RSA public key and producing a SHA-256 digest of its bytes, with the

```
bool PermissionsData::CanExecuteScriptEverywhere(
  const ExtensionId& extension_id,
  ManifestLocation location
) {
  if (location == ManifestLocation::kComponent) {
    return true;
  }
  const auto& allowlist = GetScriptingAllowlist();
  return base::Contains(allowlist, extension_id);
}

bool PermissionsData::IsRestrictedUrl(
  const GURL& document_url,
  std::string* error
) const {
  // Grant access to Screen Reader extension
  if (CanExecuteScriptEverywhere(ext_id_, loc_)) {
    return false;
  }

  // Block access to WebUI without flag
  bool allow_on_chrome_urls = ForCurrentProcess()
    ->HasSwitch(kExtensionsOnChromeURLs);
  if (
    document_url.SchemeIs(kChromeUIScheme) &&
    !allow_on_chrome_urls
  ) {
    return true;
  }

  // Block access to other extensions without flag
  if (
    document_url.SchemeIs(kExtensionScheme) &&
    document_url.host() != ext_id_ &&
    !allow_on_chrome_urls
  ) {
    return true;
  }

  // [...]
  return true;
}
```

Listing 4. Code to restrict the URLs an extension can run on (adapted from [16]).

```
{
  "manifest_version": 3,
  "name": "Clone extension",
  "version": "0.0.1",
  "key": "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDEGB
    i/oD7Yl/Y16w3+gee/95/EUpRZ2U6c+8orV5ei+3CRsBsoXI
    /DPGBauZ3rWQ47aQnfoG00sXigFdJA2NhNK9OgmRA2evnsRR
    bjYm2BGltwpaLsgQPPus3PyczbDCvhFu8k24wzFyEtxLrfxA
    GBseBPb9QrCz7B4k2QgxD/CwIDAQAB"
}
```

Listing 5. Sample manifest impersonating the Screen Reader extension.

added distinction that the output hash in hexadecimal form uses an "a" to "p" alphabet instead of the standard "0-9" and "a-f". This digest is then cropped to return only the first 32 characters [79]. To get this public key for the ID generation, the browser looks at the header of the extension's signed CRX package it came in and additionally performs an integrity check against its signature [61]. However, if the extension is sideloaded from a local directory (unpacked extension) or a ZIP archive instead of using a CRX package (*e.g.*, downloaded from the Chrome Web Store), this public key information is not available. In those cases, the ID is derived from the absolute path the extension was loaded from unless there is a `key` property in the extension's `manifest.json` file (see Listing 5). If the property exists, whatever value it has will be used as the extension's Base64-encoded public key [28], thus making it possible to produce clone extensions.

Clone extensions have some limitations. Without the private key of the impersonated extension, we cannot produce a valid signed CRX package and upload it to

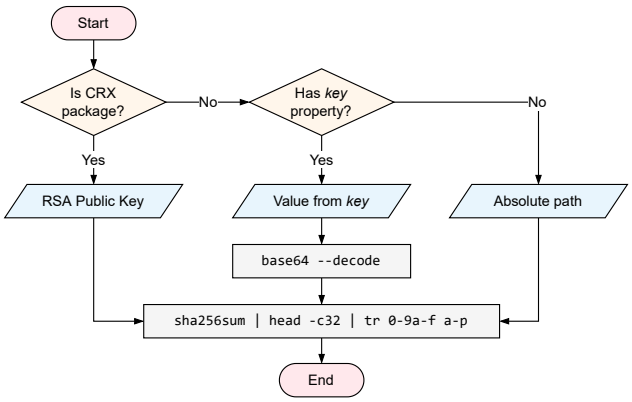

Figure 5. Flowchart for calculating extension IDs.

the Chrome Web Store. Therefore, the easiest alternate distribution method is to create a ZIP archive of the clone extension and convince the user to enable Developer Mode and then drag-and-drop the ZIP file over the `chrome://extensions` page to install it. This is one way of *sideloading* extensions into Chromium and, as discussed in §3, it is a usual threat model for malicious extensions.

Impact. Allowing extensions to run arbitrary code on WebUI pages violates both SR03 and SR04 because these pages have access to sensitive resources (*e.g.*, browsing history) and control parts of the browser UI. Some applications of this attack include:

- *Modify browser settings.* A clone extension can evaluate JavaScript on the `chrome://settings` page to read or tamper with the browser settings. This can be done without user interaction.
- *Steal passwords and credit cards.* Chromium comes with a built-in password and payment methods manager. Because plain text passwords and credit card details are accessible through the settings page, an attacker can obtain this information in plain text.[10]
- *Read GAIA ID.* Every Google account is identified by a unique *Google Accounts and ID Administration ID*, or GAIA ID for short [41]. This identifier is stored by the browser when a Google account has logged in and can be leaked from the `chrome://signin-internals/` page.
- *Modify browser flags.* Some browser settings (like the flag mentioned in this section) are not intended to be changed by regular users because of security reasons. However, since these are listed on `chrome://flags`, they can be arbitarily changed by a malicious extension.
- *List omnibox predictors.* To provide relevant autocomplete suggestions, Chromium keeps track locally of all text typed into the search bar (*i.e.*, *omnibox* [75]). The page `chrome://predictors/` exposes this data as a WebUI tab, making it accessible to an attacker.

10. In some Operating Systems like Windows, Chromium stores passwords encrypted with the logged user credentials. For this reason, leaking passwords might require users to authenticate themselves to proceed.

Our PoC demonstrates how this attack allows obtaining the details of payment methods found in `chrome://settings/payments`, one of the browser's settings pages.

4.5. Running on other extensions

Similar to WebUI pages, which are served from the privileged "chrome://" scheme, extensions also run in a restricted environment, each having their own dedicated base URL in the form of `chrome-extension://<extension-id>`. In practice, this implies that an extension can access any assets and run on targets belonging to its base URL (like its background page or service worker), but it is not allowed to do the same with other extensions. Despite running in isolation from each other, extensions can communicate with other extensions, tabs and background scripts through message passing [32]. While message communication can be used to evaluate code on a vulnerable target [23], extensions can never directly run on a third-party extension using browser extension APIs like they can with regular tabs.

Attack vector. The same experimental flag used in §4.4 to run on WebUI tabs grants access to the "chrome-extension://" scheme (Listing 4). However, there is no way to declare a host permission with this scheme in the manifest file of an extension as that will throw a validation warning and the browser will ignore the permission altogether. A crucial aspect here is that, while most browser extension APIs, like `tabs` and `scripting`, will verify host permissions before running on a target, the Debugger API implicitly grants access to any URL an extension is authorized to access. For this reason, it only checks whether a URL is restricted or not by calling the `PermissionsData::IsRestrictedUrl()` method. Because this method effectively considers a URL unrestricted when the `extensions-on-chrome-urls` flag is enabled, the Debugger API can be used to attach to any target from any extension. This can be used as an alternative exploitation method without requiring sideloading nor impersonating the Screen Reader extension using the technique described in §4.4.

Impact. A malicious extension exploiting this technique can attach to an existing legitimate extension to:

- *Steal sensitive information.* By attaching to another extension, an attacker can evaluate JavaScript code to steal passwords, PGP private keys or other sensitive information, hence breaking SR03. A clear target for this are password manager extensions.
- *Manipulate the extension behavior.* Similar to §4.2, debugging an extension can be used to change its appearance and behavior. This allows, for example, stealing funds by modifying the receiving address of a cryptocurrency transaction initiated with MetaMask [19].

This attack presents an interesting property that eases its distribution and amplifies its impact. In the past, there have been instances of legitimate extensions going rogue. One common cause is developers selling their published

items to other companies that then push a malicious update to steal private user information. These extensions operate for a limited time on a trusted marketplace (*e.g.*, the Chrome Web Store) before they get spotted and delisted from the market [64], [70]. Another popular approach to distribute malware is to publish a fake extension impersonating or cloning a legitimate one to trick a user into trusting the former [10]. This is accomplished either by having a very similar UI or just by building a modified version of the original extension with some patches applied to add the malicious functionality. The former approach has been seen in a recent operation attributed to North Korea that made the news in January 2022 [18]. The attack described in this section would not require repackaging fake extensions nor buying existing ones. Instead, an attacker just needs to distribute a malicious extension that will attach to as many legitimate targets (*i.e.*, extensions) as needed to monitor and manipulate their behavior.

To demonstrate the stealing capabilities, we created a PoC extension that attaches to LastPass [48] and Mailvelope [52] extensions to steal passwords and PGP private keys, respectively.

4.6. Attaching to the browser target

We mentioned in §2.2 that extensions using the Debugger API are not supposed nor allowed to attach to the browser target. This is presumably to prevent a rogue extension from taking control of an end-user's browser. Since they cannot use the `Target` domain, extensions attach to targets using the `chrome.debugger.attach()` JavaScript binding provided by the Debugger API, which allows an extension to send CDP commands to, and only to, a given target. That is, if an extension wants to instrument two different tabs, it will need to attach to both of them separately. If it were to exist, a hypothetical extension with access to the browser target would be capable of instrumenting any tab or extension running in the browser as the `Target` CDP domain does not enforce further access control rules.

Attack vector. For reasons similar to those discussed for the Screen Reader extension (See §4.4), there is another special extension that is granted a privileged status with more capabilities, including the ability to attach to the browser target using the Debugger API: the Perfetto UI extension [40]. This is an official development tool from Google that interacts with a web app [33] to profile, record, and view Chromium execution traces. To obtain these traces, the Perfetto UI extension uses commands from the `Tracing` CDP domain, which are not accessible outside the browser target. Unlike the running on WebUI tabs and on other extensions attacks, there is no flag to replicate this backdoor access. As so, it can only be exploited by sideloading a clone extension that impersonates the Perfetto UI extension following the steps we introduced in §4.4. This is hardcoded in the Chromium source code and allows the Perfetto UI extension to attach to the browser target by specifying the undocumented "browser" target ID, as shown in Listing 6 and Listing 7.

The browser target that the Perfetto UI extension can attach to is not as powerful as the actual browser target

```
constexpr char kBrowserTargetId[] = "browser";
constexpr char kPerfettoUIExtensionId[] =
  "lfmkphfpdbjijhpomgecfikhfohaoine";

bool ExtensionMayAttachToBrowser(Extension& ext) {
  return ext.id() == kPerfettoUIExtensionId;
}

bool DebuggerFunction::InitAgentHost() {
  // [...]

  if (
    *debuggee_.target_id == kBrowserTargetId &&
    ExtensionMayAttachToBrowser(*extension())
  ) {
    agent_host_ = CreateForBrowser();
  }

  // [...]
  return true;
}
```

Listing 6. Code to allow attaching to browser target (adapted from [14]).

```
const CDP_VERSION = "1.3";
const targetId = "browser";
chrome.debugger.attach({targetId}, CDP_VERSION);
```

Listing 7. JavaScript code to attach to the browser target.

because some domains and commands are inaccessible (*e.g.*, the `Network` domain is missing from this CDP host). Nevertheless, this tool still has access to more domains than it seems to need to deliver its functionality, which goes against the principle of least privilege. One of these domains is the `Target` domain, which from here is not capable of communicating with other targets because the `Target.sendMessageToTarget` command does not deliver CDP messages to to their destination. Another functionality that does not work is the flattened access [50] —an improved mode of operation intended to deprecate the former command— as the `sessionId` property that needs to be added to CDP messages for this mode to work cannot be sent using the Debugger API. However, these limitations can be bypassed to send commands to any arbitrary target as follows:

1) Attach to the browser target using the Debugger API.
2) Create a *proxy* target (a tab with an `about:blank` URL will do) and attach to it using the Debugger API like in the previous step.
3) Send the `Target.exposeDevToolsProtocol` command to the browser target to expose a pair of JavaScript bindings in the proxy target. Then, they will be used to communicate with the desired arbitrary target through a different communication channel that will allow sending CDP messages with the `sessionId` property.
4) Use the `window.cdp` object from the proxy target to attach to the desired final target (*e.g.*, a WebUI tab or an extension background page) and start sending commands to it through the proxy.

Impact. By impersonating the Perfetto UI extension, an attacker can abuse the CDP browser target to gain all capabilities from the previous attacks combined. In practice, this means an extension can hijack any regular

```
// Register CDP event listener
function onEvent(source, method, params) {
  if (params.responseStatusCode) {
    console.log("Response", params.responseHeaders);
  } else {
    console.log("Request", params.request.headers);
  }
}
chrome.debugger.onEvent.addListener(onEvent);

// Attach to browser target and enable domain
const targetId = "browser";
await chrome.debugger.attach({targetId}, "1.3");
await chrome.debugger.sendCommand({targetId},
  "Fetch.enable");
```

Listing 8. Simplified JavaScript code for logging all browser network traffic.

tab, WebUI page or extension belonging to any browser context, including incognito windows.

We include two different artifacts to exemplify this attack and its impact. The first one is another credit card stealer similar to the one we made for §4.4. However, this sample uses the Debugger API and the browser target to attach to `chrome://settings/payments`. The second sample is a browser-wide traffic monitor that logs all requests and responses from any opened tab or running extension, including incognito windows and WebUI tabs (see Listing 8 for a simplified code snippet).

5. Discussion

In this section, we discuss the impact of our attacks, our concerns with the current Chromium extensions architecture, and the effectiveness of the solutions already implemented or proposed by the Chromium Development Team.

5.1. Impact

The attacks presented in this paper allow a malicious extension to abuse flaws in the Debugger API to steal sensitive user information and manipulate runtime behavior (Table 2), all while violating basic security requirements such as isolation or access control (Table 3). In addition, since most of the Chromium UI is implemented using WebUI pages (§4.4), an attacker is also able to change browser settings or even experimental flags. We have also shown how the Debugger API can be misused to modify site permissions (§4.2) or bypass security interstitials without any user awareness or intervention (§4.3).

Our attacks can have a high impact due to the privileged capabilities an attacker can acquire, but also because they are rooted in core components of the Chromium Project, which is critical in today's web browser market. According to Statcounters [72], Google Chrome has a 60% of market share and it is present in popular forks such as Microsoft Edge, Brave browser, and Opera among others. Furthermore, three of our attacks (§4.4, §4.5, §4.6) are in part the result of a questionable approach to integrate two officially-branded Google extensions with Chromium. This approach consists on hardcoding the IDs of such extensions into the browser's source code, thus widening the impact of the attacks by propagating these changes in Chromium forks as well. Other Chromium-based products may inherit this artifact and its associated risks, even if said extensions are not intended to run in them. This might be a even larger problem for products with a more permissive extension origin policy since they might not be aware of this threat vector.

5.2. Root causes

The attacks proposed in our work are possible due to a combination of factors that have shaped core components of the Chromium project and its development. We now discuss these enablers.

Design flaws. The attacks described in this paper do not exploit flaws in the implementation of a specification (*e.g.*, memory corruption issues). Instead, they originate from design decisions where the risk-benefit trade-off may not be well balanced. One clear example is the attack described in §4.3. Attaching to security interstitials using the Debugger API was not possible until Chromium 70 when this capability was intentionally granted [42] to prevent the Lighthouse extension—an official Google development tool—from detaching when running tests on a web application without offline support [62]. While the implementation of this change did not introduce any bugs *per se*, the design decision behind it is, at a minimum, questionable because security interstitials are triggered after a critical event that requires user consent.

The backdoor accesses used in §4.4, §4.5 and §4.6 suggest design choices that were not sufficiently evaluated from a security standpoint. These privileges were implemented in such a way that they blindly trust the ID of an extension without verifying its signature. In the case of the Screen Reader extension, even the Chromium developer who introduced the change in October 2011 explicitly acknowledged that this solution is "temporary" and that a long-term alternative is needed [53]. However, as of the writing of this paper, this temporary code written 11 years ago has not yet been superseded nor there is any indication that it will be.

As for the Perfetto UI extension, the change was introduced in January 2020 [46] to let https://ui.perfetto.dev get traces from all tabs or renderers through this extension. Unfortunately, more CDP domains apart from `Tracing` were exposed when allowing attaching to a limited instance of the browser target. It is evident that this decision was not deeply thought-out and that extensions are not expected to attach to the browser target given the convoluted workaround we had to come up with for §4.6.

Lack of specifications. While the Chromium Project has public Design Documents for most of its components [15], we could not find the specification for some extension APIs, including the Debugger API. This makes it challenging to independently determine whether a particular browser behavior is a feature or a bug. Thus, we were unsure whether listing and attaching to targets from incognito windows (§4.1, §4.2) was intended or not until April 2022 when we got confirmation from the Chrome Development Team that it was certainly not intended. Another example is the `extensions-on-chrome-urls` flag used in both §4.4 and §4.5. While it allows extensions to run on "chrome://" URLs, it also grants the Debugger API access to the "chrome-extension://" scheme. We could not find this behavior documented anywhere. Once again, we

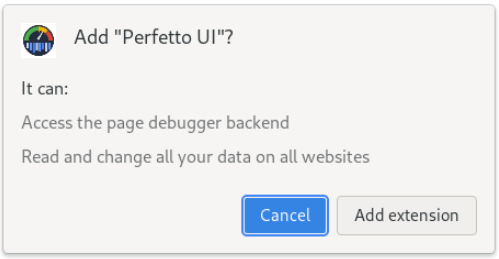

Figure 6. Install confirmation dialog for an extension using the Debugger API.

are unsure whether this is intentional or not as the name of the flag does not suggest it will also apply to browser extensions. To make matters worse, to this date we still do not know why the flag was introduced in the first place as the associated Chromium issue[11] has not yet been made public.

Overpowered APIs. Extensions can control almost every single aspect of the browser through the use of extension APIs. As we demonstrate in this paper, the Debugger API is one of the more powerful ones, if not the most. Some CDP commands can even break basic browser isolation mechanisms (§4.2). This allows accessing or modifying any cookie stored in the browser without the user being clearly aware of the capabilities of the extension. Additionally, we have shown how targets from incognito windows are not isolated either as the Debugger API can list and attach to them even when the extension has not been granted explicit permission.

Despite the Debugger API being a highly privileged and potentially dangerous capability, the associated risk is not reflected accordingly. We found all of our attacks violate SR01: when installing an extension that declares the `debugger` permission, the install prompt merely shows a vague warning informing the user that the extension will have access to the *"page debugger backend"* (Figure 6). We find this risk communication strategy very inadequate. Other platforms have dealt with apps that abuse powerful permissions in the past. In the case of Android, this was partially addressed by mandatorily locking certain resources behind runtime permissions [4]. Although Chromium extensions have a similar concept called "optional permissions," it is up to the developers to use them or keep requesting a permission at install-time [25].

Another issue with the Debugger API is that users have little control over which targets an extension is allowed to debug. As mentioned in §4.2, users can force a debugging extension to detach from all targets by closing an infobar (§2.3). Nevertheless, extensions can reattach immediately after. Even if the user becomes aware of this deceptive behavior, there is little they can do to prevent it besides uninstalling the malicious extension. This situation turns the notification infobar into a merely informative mechanism that can be easily defeated by an effectively overpowered extension.

11. See https://crbug.com/174183.

5.3. Solutions

We reported all of our findings to Google. The issue related to the attack from §4.1 was flagged as a duplicate of a similar report[12] by another researcher in August 2021, which stayed stale for months until we submitted our own report. We did not know this at the time of reporting. Both this bug and the one we present in §4.3 have already been addressed and fixed by adding appropriate security controls. The different vulnerabilities that enable the attacks from §4.4, §4.5 and part of §4.6 were merged into a single issue[13] after Google flagged the rest of them as duplicates on December 2021. More than a year later, this issue is still open and awaiting a fix from the Chromium Development Team.

The other bug[14] that we reported in relation to §4.6 was initially considered a security vulnerability but then flagged as "not-a-security-bug," consequently making it public. While the issue remains open and at some point there was some discussion around preventing side-loaded extensions from attaching to the browser target, no progress has been made since.

Regardless of whether our attacks exploit actual bugs or intended features, it is evident that the Debugger API provides extensions with a wide range of powerful capabilities, and that such privileged capabilities require extensive and effective security controls. At some point during the vulnerability disclosure process, members of the Chromium Team proposed restricting the Debugger API in its entirety to users with developer mode enabled. This was later deemed not ideal as it would impact legitimate developers. Instead, one strategy we propose for risk reduction would be to follow the principle of least privilege and redesign a finer grained permission system for the Debugger API. Examples of this idea abound in other platforms, including the Linux capabilities introduced in the kernel starting with version 2.2 [49], or the splitting of the `location` permission in Android into more than one for different purposes. In the case of the Debugger API, having a separate permission for each CDP domain will greatly improve security and reduce the impact of abusive extensions. In fact, this is explicitly advised in the CRX API Security Checklist from the Chromium Security Team [13].

The security principle of Separation of Privilege (*i.e.*, granting access based on meeting multiple independent conditions) can also contribute to risk reduction. Two easy-to-implement additional security checks that can be tested when an extension intends to access the Debugger API are:

- *"Trusted" v. "non-trusted" extensions.* Chromium already can detect the origin of an extension (*e.g.*, Chrome Web Store, sideloaded). The browser could use this information to restrict certain manifest permissions to "trusted" extensions that are signed or coming from a verified distribution platform.
- *Asking for user consent at runtime.* Instead of just showing a merely informative infobar (§2.3),

12. See https://crbug.com/1236325.
13. Reported in https://crbug.com/1276497.
14. Reported in https://crbug.com/1301966.

we suggest replacing it with a consent dialog asking the user to grant or deny extension attachment requests. A similar approach to put sensitive APIs behind a user gesture permission or a security warning was proposed by a member of the Chromium Development Team.

Regarding the Screen Reader and Perfetto UI extensions, hardcoding the identifier of a privileged subject is not a robust security mechanism and opens backdoors. Perhaps the safest option would be to convert them to internal Chromium components. In doing so, the extra capabilities needed exclusively by these two extensions could be packaged into private extension APIs accessible to them instead of granting access based on a hardcoded credential.

To discern whether potential issues are actual intended features and to ease their assessment, we suggest including the corresponding technical specification alongside code changes. This would facilitate to publicly audit code changes to make sure they comply with the specification. Additionally, having a link between the design and the implementation makes Design Documents easier to find.

6. Related work

Detecting harmful extensions. Previous research on the web extensions ecosystem has focused on detecting malicious browser extensions (MBE). DeKoven *et al.* describe a methodology for identifying users visiting Facebook that had MBEs installed in their browser [20], which builds upon the work of Kapravelos *et al.* in Hulk [45]. Based on it, they were able to label close to 2k malicious Chromium and Firefox extensions and notify their users. Pantelaios *et al.* used a sample of almost 1M different extension releases to propose a system for detecting malicious updates [60]. Saini *et al.* introduce attacks where colluding extension share and access sensitive information without being noticed [67]. Similarly, other works focus instead on extensions intentionally leaking personal data. Chen and Kapravelos developed an automated taint-analysis framework and found that almost 4k of extensions downloaded from the Chrome Web Store leaked privacy-sensitive information [11]. Weissbacher *et al.* propose Ex-Ray, a technique for dynamically detecting browsing history leaks through traffic analysis [78].

Detecting vulnerable extensions. Prior studies looked for browser extensions with bugs that can be exploited by web pages or another extension. Bandhakavi *et al.* used static analysis to automatically flag potentially vulnerable code in legacy Firefox extensions [6]. Fass *et al.* improved on this with DOUBLEX, a classifier that managed to accurately detect known flaws on a labeled vulnerable extension dataset [23].

Listing running extensions. Sanchez-Rola *et al.* used timing side-channel attacks to abuse flaws in the implementation of access control settings in multiple browsers that support extensions [68]. Laperdrix *et al.* looked at CSS modifications injected into web pages by extensions to fingerprint the set of extensions running in a browser [47]. Starov and Nikiforakis proposed a similar technique in XHOUND, an automated framework to measures changes in the DOM for the same purpose [71].

Analysis of the browser extensions architecture. All the previously mentioned works showcase how powerful extensions are. However, they mainly focus on identifying abusive or abusable extensions, not on discussing the browser extension architecture that enables their behavior. Reeder *et al.* performed a user study on warning messages displayed by the browser [66]. They found that, while warning design has come a long way, there is still room for improvement as the context of a warning largely influences the resulting outcome.

7. Conclusions

In this work we presented several attacks that exploit vulnerabilities in the Chromium Debugger API. Our attacks allow a malicious extension to take full control of the browser, access sensitive information, impersonate other extensions, and exploit restricted features not intended to be used by any extension. We demonstrated some of these actions through several PoCs that we made publicly available. Our attacks affect every major Chromium-based browser. We reported our findings to Google, which already fixed some of the vulnerabilities and is addressing the rest at the time of this writing.

Even though inadequate security checks are enablers for some of our attacks, we believe that the Debugger API grants extensions excessive capabilities through just one permission, and that the granting mechanism does not fully convey the risk to the user. We have provided constructive discussion on the root cause for most of these vulnerabilities and potential strategies to improve the security of the Debugger API.

Acknowledgements

We are deeply grateful to our anonymous reviewers and to Chromium's Security Team for their valuable insights and recognition. This research was supported by the Spanish Government grant ODIO (PID2019-111429RB-C21 and PID2019-111429RB-C22); the Region of Madrid, co-financed by European Structural Funds ESF and FEDER Funds, grant CYNAMON-CM (P2018/TCS-4566); and by the EU H2020 grant TRUST aWARE (101021377). José Miguel Moreno was supported by the Spanish Ministry of Science and Innovation with a FPI Predoctoral Grant (PRE2020-094224). Narseo Vallina-Rodriguez was supported by a Ramon y Cajal Fellowship (RYC2020-030316-I).

The opinions, findings, and conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of any of the funding bodies.

References

[1] Crx4Chrome. https://www.crx4chrome.com/, 2022.

[2] ExtManager. https://www.extfun.com/, 2022.

[3] GugeApps. https://www.gugeapps.net/, 2022.

[4] Android Developers. Permissions on Android. https://developer.android.com/guide/topics/permissions/overview, 2022.

[5] Pieter Arntz. Chrome extensions that lie about their permissions. https://blog.malwarebytes.com/puppum/2020/08/chrome-extensions-that-lie-about-their-permissions/, 2020.

[6] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *Proceedings of the 19th USENIX Security Symposium*. USENIX Association, August 2010.

[7] Kayce Basques. Analyze runtime performance. https://developer.chrome.com/docs/devtools/evaluate-performance/, 2017.

[8] Kayce Basques. Open Chrome DevTools. https://developer.chrome.com/docs/devtools/open/, 2018.

[9] Alice Boxhall. Allow extensions on chrome:// URLs, when flag is set and permission is explicitly requested. https://codereview.chromium.org/12792005/, 2013.

[10] Martin Brinkmann. Don't download this Microsoft Authenticator extension for Chrome: it is fake. https://www.ghacks.net/2021/05/18/dont-download-this-microsoft-authenticator-extension-for-chrome-it-is-fake/, 2021.

[11] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1687–1700. ACM, 2018.

[12] Chromium Developers. Chromium Security Education. https://www.chromium.org/Home/chromium-security/education/, 2022.

[13] Chromium Developers. CRX API Security Checklist. https://docs.google.com/document/d/1RamP4-HJ7GAJY3yv2ju2cK50K9GhOsydJN6KIO81das/pub, 2022.

[14] Chromium Developers. debugger_api.cc. https://chromium.googlesource.com/chromium/src/+/100.0.4896.190/chrome/browser/extensions/api/debugger/debugger_api.cc, 2022.

[15] Chromium Developers. Design Documents. https://www.chromium.org/developers/design-documents/, 2022.

[16] Chromium Developers. permissions_data.cc. https://chromium.googlesource.com/chromium/src/+/100.0.4896.190/extensions/common/permissions/permissions_data.cc, 2022.

[17] Catalin Cimpanu. A third of all Chrome extensions request access to user data on any site. https://www.zdnet.com/article/a-third-of-all-chrome-extensions-request-access-to-user-data-on-any-site/, 2019.

[18] Catalin Cimpanu. North Korean hackers stole nearly $400M in cryptocurrency in 2021. https://therecord.media/north-korean-hackers-stole-nearly-400m-in-cryptocurrency-in-2021/, 2022.

[19] ConsenSys Software Inc. MetaMask. https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn, 2022.

[20] Louis F. DeKoven, Stefan Savage, Geoffrey M. Voelker, and Nektarios Leontiadis. Malicious Browser Extensions at Scale: Bridging the Observability Gap between Web Site and Browser. In *Proceedings of the 10th USENIX Workshop on Cyber Security Experimentation and Test*. USENIX Association, August 2017.

[21] Wenrui Diao, Yue Zhang, Li Zhang, Zhou Li, Fenghao Xu, Xiaorui Pan, Xiangyu Liu, Jian Weng, Kehuan Zhang, and XiaoFeng Wang. Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 261–275. USENIX Association, September 2019.

[22] Eloston and open-source contributors. Ungoogled Chromium. https://ungoogled-software.github.io/, 2022.

[23] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben 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*, page 1789–1804, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Pavel Feldman and open-source contributors. Playwright. https://playwright.dev/, 2022.

[25] Google Chrome Developers. Declare permissions. https://developer.chrome.com/docs/extensions/mv3/declare_permissions/, 2014.

[26] Google Chrome Developers. Match patterns. https://developer.chrome.com/docs/extensions/mv3/match_patterns/, 2017.

[27] Google Chrome Developers. Declare permissions and warn users. https://developer.chrome.com/docs/extensions/mv3/permission_warnings/, 2018.

[28] Google Chrome Developers. Manifest - Key. https://developer.chrome.com/docs/extensions/mv3/manifest/key/, 2018.

[29] Google Chrome Developers. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/, 2021.

[30] Google Chrome Developers. Deprecations and removals in Chrome 95. https://developer.chrome.com/blog/deps-rems-95/, 2021.

[31] Google Chrome Developers. Getting started. https://developer.chrome.com/docs/extensions/mv3/getstarted/, 2021.

[32] Google Chrome Developers. Message passing. https://developer.chrome.com/docs/extensions/mv3/messaging/, 2021.

[33] Google Chrome Developers. Perfetto UI. https://ui.perfetto.dev/, 2021.

[34] Google Chrome Developers. Screen Reader extension. https://chrome.google.com/webstore/detail/screen-reader/kgejglhpjiefppelpmljglcjbhoiplfn, 2021.

[35] Google Chrome Developers. What are extensions? https://developer.chrome.com/docs/extensions/mv3/overview/, 2021.

[36] Google Chrome Developers. API Reference. https://developer.chrome.com/docs/extensions/reference/, 2022.

[37] Google Chrome Developers. chrome.debugger API. https://developer.chrome.com/docs/extensions/reference/debugger/, 2022.

[38] Google Chrome Developers. chrome.management API. https://developer.chrome.com/docs/extensions/reference/management/, 2022.

[39] Google Chrome Developers. Manifest V2 support timeline. https://developer.chrome.com/docs/extensions/mv3/mv2-sunset/, 2022.

[40] Google Chrome Developers. Perfetto UI extension. https://chrome.google.com/webstore/detail/perfetto-ui/lfmkphfpdbjijhpomgecfikhfohaoine, 2022.

[41] Google Developers. Access Control. https://developers.google.com/issue-tracker/concepts/access-control, 2021.

[42] Dmitry Gozman. Allow debugger extensions to access error page. https://chromium-review.googlesource.com/1228789/, 2018.

[43] Dmitry Gozman. Do not allow chrome.debugger to attach to WebUI pages. https://chromium-review.googlesource.com/935961/, 2018.

[44] Anatoli Kalysch, Davide Bove, and Tilo Müller. How Android's UI Security is Undermined by Accessibility. In *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*, pages 1–10. Association for Computing Machinery, 2018.

[45] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the 23rd USENIX Security Symposium*, pages 641–654. USENIX Association, August 2014.

[46] Andrey Kosyakov. Allow Perfetto UI to attach to browser target via chrome.debugger. https://chromium-review.googlesource.com/1986333/, 2020.

[47] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *Proceedings of the 30th USENIX Security Symposium*, pages 2507–2524. USENIX Association, August 2021.

[48] LastPass US L.P. LastPass: Free Password Manager. https://chrome.google.com/webstore/detail/lastpass-free-password-ma/hdokiejnpimakedhajhdlcegeplioahd, 2022.

[49] Linux Programmer's Manual. Overview of Linux capabilities. https://man7.org/linux/man-pages/man7/capabilities.7.html, 2021.

[50] Andrey Lushnikov. Using Chrome DevTools Protocol. https://github.com/aslushnikov/getting-started-with-cdp, 2021.

[51] Andrey Lushnikov and open-source contributors. Puppeteer. https://pptr.dev/, 2022.

[52] Mailvelope GmbH. Mailvelope. https://chrome.google.com/webstore/detail/mailvelope/kajibbejlbohfaggdiogboambcijhkke, 2022.

[53] Dominic Mazzoni. Whitelist ChromeVox so it can script WebUI pages. https://codereview.chromium.org/8100009/, 2011.

[54] MDN contributors. Browser Extensions. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions, 2021.

[55] National Vulnerability Database. CVE-2018-6178. https://nvd.nist.gov/vuln/detail/CVE-2018-6178, 2020.

[56] National Vulnerability Database. CVE-2019-5768. https://nvd.nist.gov/vuln/detail/CVE-2019-5768, 2020.

[57] National Vulnerability Database. CVE-2020-15973. https://nvd.nist.gov/vuln/detail/CVE-2020-15973, 2021.

[58] National Vulnerability Database. CVE-2021-21111. https://nvd.nist.gov/vuln/detail/CVE-2021-21111, 2021.

[59] National Vulnerability Database. CVE-2022-0107. https://nvd.nist.gov/vuln/detail/CVE-2022-0107, 2022.

[60] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 477–491, 2020.

[61] Joshua Pawlicki. CRX3 Design Doc. https://docs.google.com/document/d/1pAVB4y5EBhqufLshWMcvbQ5velk0yMGl5ynqiorTCG4, 2017.

[62] Ward Peeters. Extension error from running Lighthouse on a GitHub page. https://github.com/GoogleChrome/lighthouse/issues/5798#issuecomment-413668655, 2018.

[63] Niels Provos and Ian Fette. All About Safe Browsing. https://blog.chromium.org/2012/01/all-about-safe-browsing.html, 2012.

[64] Mishaal Rahman. Google just booted The Great Suspender off the Chrome Web Store for being malware. https://www.xda-developers.com/google-chrome-the-great-suspender-malware/, 2021.

[65] Paul Rascagneres, Thomas Lancaster, and Volexity Threat Research. SharpTongue Deploys Clever Mail-Stealing Browser Extension "SHARPEXT". https://www.volexity.com/blog/2022/07/28/sharptongue-deploys-clever-mail-stealing-browser-extension-sharpext/, 2022.

[66] Robert W Reeder, Adrienne Porter Felt, Sunny Consolvo, Nathan Malkin, Christopher Thompson, and Serge Egelman. An experience sampling study of user reactions to browser warnings in the field. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–13, 2018.

[67] Anil Saini, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. Colluding browser extension attack on user privacy and its implication for web browsers. *Computers & Security*, 63:14–28, 2016.

[68] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *Proceedings of the 26th USENIX Security Symposium*, pages 679–694. USENIX Association, August 2017.

[69] Software Freedom Conservancy. Selenium. https://www.selenium.dev/, 2022.

[70] Tom Spring. Copyfish Browser Extension Hijacked to Spew Spam. https://threatpost.com/copyfish-browser-extension-hijacked-to-spew-spam/127125/, 2017.

[71] Oleksii Starov and Nick Nikiforakis. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, pages 941–956, 2017.

[72] Statcounter Global Stats. Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share, 2022.

[73] Statcounter Global Stats. Desktop Browser Market Share China. https://gs.statcounter.com/browser-market-share/desktop/china, 2022.

[74] The Chromium Projects. Infobars. https://web.archive.org/web/20210518212452/https://www.chromium.org/user-experience/infobars/, 2008.

[75] The Chromium Projects. Omnibox. https://www.chromium.org/user-experience/omnibox/, 2008.

[76] The Chromium Projects. Mojo docs (go/mojo-docs). https://chromium.googlesource.com/chromium/src/+/main/mojo/README.md, 2022.

[77] The Chromium Projects. WebUI Explainer. https://chromium.googlesource.com/chromium/src/+/main/docs/webui_explainer.md, 2022.

[78] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. Ex-Ray: Detection of History-Leaking Browser Extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, page 590–602, New York, NY, USA, 2017. Association for Computing Machinery.

[79] Rob Wu. Obtaining Chrome Extension ID for development. https://stackoverflow.com/a/23877974/1956278, 2019.