Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies

Iskander Sanchez-Rola
Deustotech, University of Deusto

Igor Santos
Deustotech, University of Deusto

Davide Balzarotti
Eurecom

Abstract

All major web browsers support browser extensions to add new features and extend their functionalities. Nevertheless, browser extensions have been the target of several attacks due to their tight relation with the browser environment. As a consequence, extensions have been abused in the past for malicious tasks such as private information gathering, browsing history retrieval, or passwords theft — leading to a number of severe targeted attacks.

Even though no protection techniques existed in the past to secure extensions, all browsers now implement defensive countermeasures that, in theory, protect extensions and their resources from third party access. In this paper, we present two attacks that bypass these control techniques in *every* major browser family, enabling enumeration attacks against the list of installed extensions. In particular, we present a timing side-channel attack against the *access control settings* and an attack that takes advantage of poor programming practice, affecting a large number of Safari extensions. Due to the harmful nature of our findings, we also discuss possible countermeasures against our own attacks and reported our findings and countermeasures to the different actors involved. We believe that our study can help secure current implementations and help developers to avoid similar attacks in the future.

1 Introduction

Browser extensions are the most popular technique currently available to extend the functionalities of modern web browsers. Extensions exist for most of the browser families, including major web browsers such as Firefox, Chrome, Safari, and Opera. They can be easily downloaded and installed by users from a central repository (such as the Chrome Web Store [15] or the Firefox Add Ons [26]).

Unfortunately, extensions are also prone to misuse. In fact, due to their close relationship to the browser environment, they can be abused by an adversary in order to gather a wide range of private information — such as cookies, browsing history, system-level data, or even user passwords [7]. Due to this raising concern, the amount of research studying the security implications and vulnerabilities of browser extensions has rapidly increased in the last years [3, 4, 8, 10, 18, 21, 25].

When browser extensions were first introduced, websites were able to access all their local resources. As a consequence, malicious actors started to use that freely-accessible data to enumerate the extensions a user has installed in her system, or even to exploit vulnerabilities within installed extensions [23]. To mitigate this increasing threat, Firefox introduced the `contentaccessible` flag and Chrome a new manifest version [16] to implement some form of access control over the extension resources. In the rest of the paper we will refer to these security measures as *access control settings*. Developers of Safari decided to adopt a different mechanism, which consists in randomizing at runtime part of the extension URI [2]. We will refer to this second class of protection technique as *URI randomization*.

Information of the web browser has been used for a number of malicious or "questionable" purposes. For example, *Panopticlick* [12] creates a unique browser fingerprint using the installed fonts, among other features. *PluginDetect* [14] retrieves instead the list of plugins installed in the browser. Even worse, this technique has recently been used in two reported *fingerprinting-driven malware* campaigns [33, 37].

Thanks to the existing browser security countermeasures described above, so far extensions were protected against these fingerprinting techniques. Two very simple enumeration attacks were recently proposed to retrieve a small number of installed extensions in the browsers that adopted access control settings [6, 20]. These techniques

took advantage of accessible resources of the extensions present in Chrome and Firefox to identify a small number of popular extensions. In addition, XHOUND [34] was also recently proposed to enumerate extensions and perform fingerprinting, by measuring the changes in the DOM of the website.

In this paper we present the first in-depth security study of all the extensions resource control policies used by modern browsers. Our analysis show that **all browsers families** that currently support extensions are vulnerable to some form of enumeration attack. In particular, while the two design choices (i.e., access control settings or URI randomization) are both secure from a theoretical point of view, their practical implementation suffers from many different problems.

We discuss two offensive techniques to subvert these control policies, one based on a timing side-channel attack and one based on an involuntary leakage of the random URI token that affects many extensions. At the time of writing, these attacks undermine the extension security of all browsers. We also discuss a set of attacks based on these techniques, which allow third-parties to perform precise user fingerprinting, or to perform various types of targeted attacks, performing proof-of-concept tests of some of them.

We already reported the discovered problems to the involved browsers and extensions developers and we are currently discussing with them about possible fixes.

In summary, this paper makes the following contributions:

- We propose the first time-based extension enumeration attack that can retrieve the complete list of extensions installed in browsers that use access control settings. This method largely outperforms any previous extension fingerprinting methodology presented to date.
- We design a static analysis tool for Safari extensions, and use it to flag hundreds of potentially vulnerable cases in which the developers leaked the random extension URI. Through an exhaustive manual code analysis on a subset of the extensions, we confirm that this is indeed a very widespread problem affecting a large fraction of all Safari extensions.
- We show that browsers extension resources control policies are very difficult to properly design and implement, and they are prone to subtle errors that undermine their security. Our research led to numerous discussions with the developers of all major browsers and extensions, including the ones vulnerable to our attacks and the ones that are still in the design or testing phase. As a result, our study is helping to secure all browsers against these common errors.

The remainder of this paper is organized as follows. §2 provides the background on extension control methods. §3 describes the problems and two different attacks to subvert them. §4 describes the impact of the problems in a broad set of scenarios. We then discuss possible countermeasures and summarize the outcome of our research in §5. Finally, §6 discusses related work and §7 concludes the paper.

2 Background

All browsers that support extensions implement some form of protection to prevent arbitrary websites from enumerating the installed extensions and freely accessing their resources. After an extensive survey of several traditional and mobile browser families, we identified two main classes of protection mechanisms currently in use: *access control settings* (§2.1), and *URI randomization* (§2.2).

2.1 Access Control Settings

The most popular approach to protect extension resources from unauthorized accesses consists in letting the extensions themselves specify which resources they need to be kept private and which can be made publicly available. All browsers that adopt this solution rely on a set of configuration options included in a manifest file that is shipped with each extension. For security reasons, by default all the resources are considered private. However, developers can specify in the manifest a list of accessible resources.

This solution is currently used by all browsers based on Chromium, all the ones based on Firefox and Microsoft Edge.

Chromium family

The Chromium family includes all versions of Chromium (such as Google Chrome), and all browsers based on the Chromium engine (e.g., Opera, Comodo Dragon, and the Yandex browser).

Extensions in this family are written using a combination of HTML, CSS, and JavaScript [17]. They are not required to use any form of native code, as it is instead the case for plugins or other forms of browser extensions. Each Chromium extension includes a JSON file called `manifest.json` that defines a set of properties such as the extension name, description, and version number (see Figure 1 for an example of manifest). The

```
"name": "description",
"example": "Example extension",
"version": "1.0",

"browser_action": {
  "default_icon": "icon.png",
  "default_popup": "popup.html"},

"permissions": [
  "activeTab",
  "https://ajax.googleapis.com/"],

"web_accessible_resources": [
  "images/*.png",
  "style/double-rainbow.css",
  "script/double-rainbow.js",
  "script/main.js",
  "templates/*"], ...
```

Figure 1: Snippet of a Chrome Extension `manifest.json` file.

manifest is used by the browser to know the functionality offered by the extension and the permissions required to perform those actions [16].

In the first version of the manifest, there was no restriction over the resources of the extensions accessible from third-party websites. Because of that, different tools were released to take advantage of this weakness to enumerate user extensions and exploit their vulnerabilities [23]. To mitigate this threat, Google decided to introduce dedicated access control settings in the second version of the manifest file. This extension uses a parameter (`web_accessible_resources`) to specify the paths of packaged resources that can be used in the context of a website. Resources are available through the URL `chrome-extension://[extID]/[path]`. However, any navigation access to an extension or its resources is blocked by the browser, unless the extension resource has been previously listed as accessible in its `manifest.json`. This solution was explicitly designed to minimize the attack surface while protecting users' privacy.

Firefox family

Firefox family extensions (or Add-ons, as they are called in the Mozilla jargon) can add new functions to the web browser, change its behavior, extend the GUI, or interact with the content of websites. Add-ons have access to a powerful API called XPCOM [30], that enables the use of several built-in services and applications through the XPConnect interface. In the Firefox family (which includes for example Firefox Mobile, Iceweasel and Pale Moon), extensions are written in a combination of JavaScript and XML User Interface Language (XUL). Extensions are also allowed to use functionality from third-party binaries or create their own binary components. Recently, Mozilla changed its extension development framework, introducing the Add-on SDK of the JetPack project [28]. This development kit provides a high-level API, easing the development process and addressing some of the security issues of previous Firefox extensions.

The registration and allocation of the different extensions is performed through the *Chrome Registry* [27] which is also in charge of customizing user interface elements of the application window that are not in the windows content area (such as toolbars, menu bars, progress bars, or windows title bars). Each extension contains a `chrome.manifest` file that specifies options related to three main categories — content, locale, and skin — as exemplified in the following snippet:

```
content  ext                 src/content/
skin     ext  classic        src/skin/
locale   ext  en-US          src/locale/en-US/
content  pck  chrome/ext/pck contentaccessible=yes
```

As it was the case for Chromium extensions, originally there was no control performed to prevent external websites from accessing the different resources of an extension. And also in this case, developers decided to solve the problem by including a new option in the `chrome.manifest` (called `contentaccessible` and depicted in the last line of the previous example) that specifies which resources can be publicly shared. However, resources have a restricted access by default, unless `contentaccessible=yes` is specified in the manifest.

Firefox is now developing a new way of handling Add-ons called WebExtensions [29]. This technology is designed mainly for cross-browser compatibility, supporting the extension API of Chromium. Porting extensions between the two platforms will require few changes in the code of the Add-on. The new extensions will also use a `manifest.json`, including some extra data specific for Firefox (see Figure 2). In order to access the different resources of the extension, Firefox will use the `moz-extension://` schema.

As WebExtensions are currently in an early stage we are not including them in our tests, but we notified their developers and we will discuss more about them in §5.

Microsoft Edge

Edge will be the first Microsoft browser to fully support extensions. It will follow a Chrome-compatible extension model based on HTML, JavaScript and CSS. This means that the migration process to Microsoft Edge for

```
"applications": {
  "gecko": {
    "id": "{the-addon-id}",
    "strict_min_version": "40.0.0",
    "strict_max_version": "50.*"
    "update_url": "https://foo/bar"
  }
} ...
```

Figure 2: Snippet of a Firefox WebExtension manifest's new data.

Chrome extension developers will require minimal effort.

Beside the general web APIs, a special extension API will provide a deeper integration with the browser, making possible to access features such as tab and window manipulation. The manifest will be named `manifest.json` and will use the same JSON-formatted structure and general properties of the Chromium implementation. The URL to access the extension resources follows the `ms-browser-extension://[extID]/[path]` schema.

As the design is in its preliminary stages and it is not yet fully working, we are not including it in our analysis.

2.2 URI Randomization

As Safari was one of the last major browsers to adopt extensions, its developers implemented a resource control from the beginning to avoid enumeration or vulnerability exploitations of installed extensions. Instead of relying on settings included in a manifest file like all the other major browsers, Apple developers adopted a URI randomization approach. In this solution there is no distinction between private or public resources, but instead the base URI of the extension is randomly re-generated in each session.

Safari extensions are coded using a combination of HTML, CSS, and JavaScript. To interact with the web browser and the page content, a JavaScript API is provided and each extension runs within its own "sandbox" [1]. To develop an extension, a developer has to provide: (i) the global HTML page code, (ii) the content (HTML, CSS, JavaScript media), (iii) the menu items (label and images), (iv) the code of the injected scripts, (v) the stylesheets, and (vi) the required icons.

These components are grouped into two categories: the first including the global page and the menu items, and the second including the content, and the injected scripts and stylesheets. This second group cannot access any resource within the extension folder using relative URLs as the first group does. Instead, these extension components are required to use JavaScript

```
1 <script type = "text/javascript">
2 var myImage = safari.extension.baseURI +
3 "Images/paper.jpg";
4 document.body.style.cssText =
5 "background-image: url(" +myImage+ ")";
6 </script>
```

Figure 3: Example of background image load in CSS using absolute URLs in Safari extension.

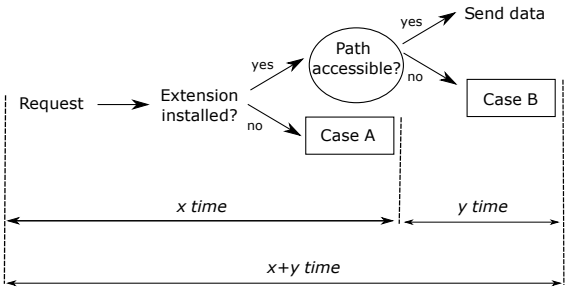

Figure 4: Resource accessibility control schema.

to access the randomized URI that changes each time Safari is launched. Absolute URIs are stored in the `safari.extension.baseURI` field, as shown in Figure 3.

3 Security Analysis

In the previous section we presented the two complementary approaches adopted by all major browser families to protect the access to extension resources. The first solution relies on a public resource URI, whose access is protected by a centralized code in the browser according to settings specified by the extension developers in a manifest file. The second solution replaces the centralized check by randomizing the base URI at every execution. In this case, the extension needs to access its own resources by using a dedicated Javascript API.

While their design is completely different, both solutions provide the same security guarantees, preventing an attacker from enumerating the installed extensions and accessing their resources. We now examine those two approaches in more detail and discuss two severe limitations that often undermine their security. It is important to note that these attacks can also be used in any type of device with a browser with extension capability, such as smartphones or smartTVs.

3.1 Timing Side-Channel on Access Control Settings Validation

As already mentioned, the vast majority of browsers adopt a centralized method to prevent third parties from

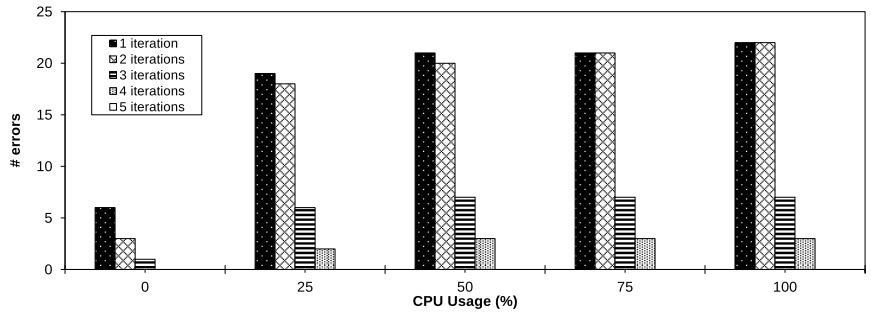

Figure 5: Comparison between number of iterations and errors with different CPU usages (%), .

accessing any resource of the extensions that have not been explicitly marked as public. Therefore, when a website tries to load a resource not present in the list of accessible resources, the browser will block the request. Despite the fact that, from a design point of view, this solution may seem secure, we discovered that all their implementations suffer from a serious problem that derives from the fact that these browsers are required to perform two different checks: (i) to verify if a certain extension is installed and (ii) to access their control settings to identify if the requested resource is publicly available (see Figure 4 for a simple logic workflow for the enforcement process). When this two-step validation is not properly implemented, it is prone to a timing side-channel attack that an adversary can use to identify the actual reasons behind a request denial: the extension is not present or its resources are kept private. To this end, we used the *User Timing API*[1], implemented in every major browser, in order to measure the performance of web applications.

As an example, an attacker can code few lines of Javascript to measure the response time when invoking a fake extension (refer to case A in Figure 4). For instance, in Chromium the requested URI could look like this:

```
chrome-extension://[fakeExtID]/[fakePath]
```

Then, the attacker can generate a second request to measure the response time when requesting an extension that actually exists, but using a non-existent resource path (case B in Figure 4):

```
chrome-extension://[realExtID]/[fakePath]
```

By comparing the two timestamps, the attacker can easily determine whether an extension is installed or not in the browser. Similar response times mean that the central validation code followed the same execution path on the two requests and, therefore, the extension is not installed in the browser. Otherwise, significantly different execution times mean that only the second test failed and, therefore, that the requested extension is present in the browser.

We performed an experiment in order to empirically tune the time difference threshold and the number of correct requests required to ensure the correctness of our attack. In particular, the following configuration was used:

- We configured 5 different CPU usages: 0%, 25%, 50%, 75%, and 100%. The experiment was executed on a 2.4GHz Intel Core 2 Duo with 4 GB RAM commodity computer.
- The attack was configured to be repeated from 1 to 10 iterations. Note that each iteration performs two calls to the browser: one that asks for the fake extension and one that asks for the actual extension with a fake path.
- We repeated each attack testing 500 times to avoid any bias. In this way, we performed: 2 calls $\times$ 10 iteration configurations $\times$ 500 times $\times$ 5 CPU usages, resulting in a total number of 275,000 calls.

We observed that, when the execution paths were different, the response times differed by more than 5%. It is important to remark that our method exploits the proportional timing difference between two different calls rather than using a pre-computed time for a specific device. Figure 5 shows the precision across different CPU loads and different numbers of iterations. Five iterations were sufficient enough to achieve a 100% success rate even under a 100% CPU usage.

Affected Browsers

We tested our timing side-channel attack on the two browser families (Chromium-based and Firefox-based) that use extensions access control settings.

[1]https://www.w3.org/TR/user-timing/

Table 1: Percentage extension detected by previous methods.

	Chrome	Firefox	Total
# Extensions Tested	10,620	10,620	21,240
% Previous Approaches	12.73%	8.17%	10.45%
% Our Approach	*100.00%*	*100.00%*	*100.00%*

Our experiments confirm that all versions of Chromium are affected by this vulnerability. Browsers such as Chrome, Opera, the browser of Yandex (largest search engine in Russia) and the browser of Comodo (largest issuer of SSL certificates) are included in this group. As aforementioned, we are not including Edge and Firefox WebExtensions because they are still in early stages of development. However, as they follow the same extension control mechanism as Chromium, they are also likely to be vulnerable to our timing side-channel attack.

Surprisingly, non-WebExtensions in Firefox suffer from a different bug that makes even easier to detect the installed extensions. The browser raises an exception if a webpage requests a resource for non-installed extension (case A in Figure 4), but not in the case when the resource path does not exist (case B in Figure 4). While the exception does not cause any visible effect in the page, an attacker can simply encapsulate the invocation in a `try-catch` block to distinguish between the two execution paths and reliably test for the presence of a given extension.

Extensions Enumeration

By telling apart the two centralized checks that are part of the extension settings validation (either because of the side-channel or because of the different exception behaviors), it is possible to completely enumerate all the installed extensions. It is sufficient for an attacker to simply probe in a loop all existing extensions to precisely enumerate the ones installed in the system.

In comparison, previous bypassing techniques [6, 20] were only able to detect a small subset of the existing extensions. In order to precisely assess the accuracy improvement over these previous techniques, we conducted an experiment on a set of 21,240 extensions. For this test, we decided to focus on the two browsers with the highest number of available extensions: Chrome and Firefox (Opera also has its own extension store, but the number of popular extensions is very low compared with the other browsers). In the case of Chrome, extensions are divided in three different groups: extensions, apps, and games. Although one of the groups is explicitly called extensions, all of them are installed as a `chrome-extension` and follow the same access control settings model.

At the time of writing, the number of recommended extensions in the games category (the smallest of the three) was 3,540. To keep a balanced dataset, we therefore selected also the top 3,540 of the remaining two categories, resulting in a balanced dataset of the 10,620 most recommended extensions.

For Firefox, the selection process was easier because its store makes no distinction among different categories. Therefore, we selected the 10,620 most popular Firefox extensions to keep our complete dataset equally balanced between the two browsers.

To measure the coverage of previous bypassing methods and compare it with the full coverage of our bypass technique, we combined the methods described in [6,20]. These methods are, to the best of our knowledge, the only ones that exist capable of enumerating extensions by subverting access control settings. These methods are based on checking the existence of externally accessible resources in extensions. To test them, we analyzed the manifest files of all extensions we downloaded, looking for any accessible resources.

Table 1 shows the obtained coverage using previous methods. Chrome extensions were easier to enumerate than the ones in the Firefox store. However, the coverage of these old methods is very low compared to the full coverage achieved by our method.

3.2 URI Leakage

Even if URI randomization control is completely centralized, it strongly depends on developers to keep resources away from any third-party access. In fact, extensions are often used to inject additional content, controls, or simply alert panels into a website. This newly generated content can unintentionally leak the random extension URI, thus bypassing the security control measures and opening access to all the extension resources to any other code running in the same page. In addition, the leaked random URI may be used by third-parties to unequivocally identify the user while browsing during the same session.

A simple example taken from the Web of Trust[2] extension is shown in Figure 6. The code snippet creates a new `iframe` (line #11), sets its `src` attribute to the `baseURI` random address of the extension (line #14), and adds the frame to the document body (line #19). As a result, any other JavaScript code running in the same page (and therefore potentially under control of an attacker) can retrieve the address of the injected `iframe` and use it to access any resource of the extension. In fact, once the random token is known, the browser offers no other

[2] `https://www.mywot.com/`

```
1  wot.rating = {
2  toggleframe: function(id, file, style){
3   try {
4     var frame = document.getElementById(
          id);
5     if (frame) {
6      frame.parentNode.removeChild(frame);
7      return true;
8     } else {
9      var body = document.
           getElementsByTagName("body");
10     if (body && body.length) {
11      frame = document.createElement("
            iframe");
12      if (frame) {
13       frame.src = safari.extension.
             baseURI+file;
14       frame.setAttribute("id", id);
15       frame.setAttribute("style", style)
             ;
16       if (body[0].appendChild(frame))
17        {return true;}
18     }
19    }
20   }
21  } catch (e) {
22    console.log("failed with"+e+"\n");}
23  return false;
24 }
```

Figure 6: Web Of Trust Safari extension function that creates an `iframe` in the website with the `baseURI` random variable as source.

security mechanism to protect the access to an extension resources.

While this may seem like a simple bug in the extension development, our experiments show that it is instead a very widespread phenomenon. The entire security of the extension access control in Safari relies on the secrecy of the randomly generated token. However, the token is part of the extension URI which is often used by the extensions to reference public resources injected in the page. As a result, we believe that this design choice makes it very easy for developers to unintentionally leak the secret token.

Estimating the Scale of the Problem

The Web-of-Trust example discussed above consists of a single function of 30 lines of code, but not all the cases are so obvious to identify without a complex static analysis of the extension.

To estimate how prevalent the problem is, we implemented a prototype analyzer that reports candidate cases of URI leakage in all Safari extensions. Our tool is based on Esprima[3] to perform a static analysis based on the Abstract Syntax Trees (ASTs) of all the different JavaScript components of the extension under analysis. Source and sinks are located by just looking for the specific code in the nodes of the tree, while the information flow is computed by following the different pieces of code that actually have access to the data along the different execution paths. In particular, the analysis is performed in three steps:

[3] `https://github.com/jquery/esprima`

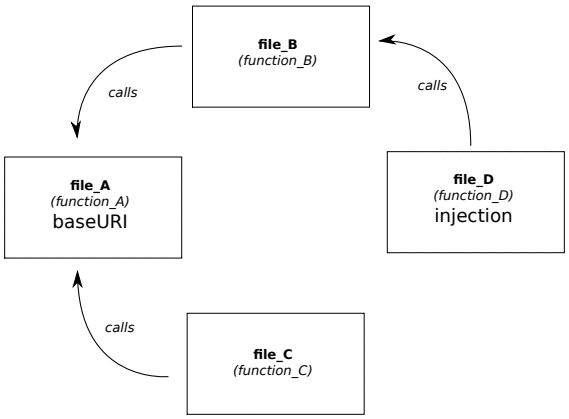

Figure 7: Simplified example schema of an extension that leaks the `baseURI`.

1. In the first step, the tool identifies the *source* locations where the code accesses the random extension URI (looking for calls to the `baseURI` method).

2. The tool then separately analyzes all the components that can use the retrieved value. Following the information flow (i.e., functions that are are called or are calling), this process is performed recursively until no more connections are found.

3. For every identified components, the tool locates the *sinks*, i.e., the location where new content is injected in the webpage (e.g., through the `createElement` and `appendChild` methods). If there is a connection between the `baseURI` access and the injection of an element in the website, the extension is flagged as suspicious and reported for further analysis.

The schema in Figure 7 shows a simplified example of an extension that leaks the `baseURI` using `function_A` of `file_A` to obtain the value, `function_B` of `file_B` as an intermediate phase, and `function_D` of `file_D` to finally make the injection on the website.

This technique is designed to act as a screening filter and NOT as a precise detection method. Indeed, the fact that an extension retrieves the `baseURI` and then uses it to create some content is not sufficient to identify if the full information is actually leaked. For instance, we

Table 2: Percentage of potential baseURI leakage in safari extensions.

Category	# Total Ext.	# P. Leak
Shopping	95	57.89%
Email	13	53.85%
Security	84	52.38%
News	20	45.00%
Photos	25	44.00%
Bookmarking	61	42.62%
Productivity	147	40.82%
RSStools	5	40.00%
Entertainment	37	37.84%
Translation	8	37.50%
Social	80	30.00%
Developer	57	29.82%
Other	42	26.19%
Search	42	21.43%
urlshorteners	5	0.00%
Total	*721*	*40.50%*

found an extension that used the `baseURI` to retrieve its version number and then injected an `iframe` with the version number included directly as part of its URL, but without leaking the complete `baseURI`.

To evaluate our tool, we downloaded and analyzed all the available extensions within the Safari Extension Gallery[4]. The 718 extensions belonged to 15 different categories (e.g., security, shopping, news, social networking, and search tools).

Table 2 shows the obtained results. In general, more than 40% of the Safari Extension Gallery were potentially vulnerable to our enumeration technique. We decided to manually analyze some of the results to determine whether the reported extensions actually performed the leak or not. Since the security category is among the ones with the highest percentage of extensions with a potential leak and it is also particularly sensitive due to the type of information these extensions usually deal with (such as user passwords), we decided to manually verify all the results for the extensions in this category.

With a considerable effort, we performed an exhaustive manual code review of all the security extensions, selecting those that were completely functional, excluding the ones that required payment for their services. Among the 68 extensions in this group, 29 were flagged as suspicious of making the leakage and 39 were not leaking it. From the suspicious ones, 20 out of 29 actually leaked the secret `baseURI`. In addition, we only identified one false negative that leaked the information but was not identified by our static analysis tool. In particular, this extension obtained the complete `URL`, including `baseURI`, but stored it locally. Within the extensions that are vulnerable to our attack, we found popular protection extensions such as Adblock[5], Ghostery[6], Web Of Trust[7], and Adguard[8]. The list also includes password managers, such as LastPass[9], Dashline[10], Keeper[11], and TeedyID[12] and combinations of the two functionalities (e.g., Blur from Abine[13]).

In summary, a relevant number of Safari extensions are vulnerable to our technique, including several important and very popular security-related extensions. As explained in §5, we are now in the process of validating all the results and contacting the developers of the affected extensions to fix their code.

4 Impact

In the previous section we discussed the security of access control settings and URI randomization, and we showed how every mechanism adopted by current browsers can be easily bypassed in practice. There are several possible consequences of abusing the information provided by our two techniques.

4.1 Fingerprinting & Analytics

The most accurate and controversial form of fingerprinting aims at building a unique identifier for each user device, such as *Panopticlick* [12]. It is considered a *stateless* technique, because in order to build and share the unique identifier, these techniques do not require to store anything on the user machine (in contrast with *stateful* techniques such as *Cookies*). To build a unique identifier, several features are retrieved from the user's machine and combined in a unique fingerprint. This process can be repeated across multiple websites and the identifier will always be the same for the same machine, allowing trackers to determine users' browsing history, among other tasks. Using the set of installed extensions can increase the uniqueness of the resulting fingerprint. To measure the exact fingerprinting ability of extension enumerations, a study should be performed to measure the discriminatory power of the most popular extensions available for each browser. To this end, we have conducted a preliminary study of this type of analysis in §4.3.

The techniques proposed in this paper can also be used to perform a completely accurate browser finger-

[4] `https://extensions.apple.com/`

[5] `https://getadblock.com/`
[6] `https://www.ghostery.com/`
[7] `https://www.mywot.com/`
[8] `https://adguard.com/`
[9] `https://lastpass.com/`
[10] `https://www.dashlane.com`
[11] `https://keepersecurity.com/`
[12] `https://www.teddyid.com/`
[13] `https://dnt.abine.com`

printing without checking the `User-Agent`. To this end, our method can be used to check for **built-in extensions**. These extensions are pre-installed and present in nearly every major web browser and there is no possibility for the user to uninstall them. Therefore, if we configure our techniques to check one of these built-in extensions that does not exist in other browsers, a website can precisely identify the browser family with 100% accuracy.

The installed extensions enumeration combined with the aforementioned browser identification can be used to determine users' demographics. The extensions that a particular user utilizes can be easily discovered by websites or third-party services. Installed extensions provide information about a particular user's interests, concerns, and browsing habits. For example, users with security and privacy extensions installed in their browsers such as *Ghostery* or *PrivacyBadger* are potentially more aware about their privacy than other users. The same happens with personalizing extensions, games, or any possible combinations of other extensions categories. In order to measure the feasibility of performing analytics through extensions, we have conducted a proof-concept test described in §4.3.

4.2 Malicious Applications

The information retrieved from the installed extensions can also be used for malicious purposes, as the information gathering phase about potential victims is usually the first step to perform a targeted attack. For instance, attackers can inject the extension enumeration code in a compromised website and search for users with shopping management extensions and password managers to narrow down their attack surface to only those users whose credit card information has a higher likelihood to be stolen. Another possibility would be to identify the presence of a major antivirus vendor extension to personalize an exploit kit or to decide whether the malicious payload should be delivered or not to a certain user.

In addition to the attacks already presented, in a recent work, Buyukkayhan et al. [7] presented *CrossFire*, a technique that allows attacker to perform malicious actions using legitimate extensions. The part that was left unanswered by the paper is how the attacker can identify a set of installed extensions to use for her purpose. By using our enumeration technique, an attacker can create completely functional malicious extensions by knowing all installed victim's extensions in beforehand.

Due to the variability of possible extensions, the information of a particular user can be exploited in different social-driven attacks (automated or not). For example, a malicious website can exploit the information about particular extensions being installed to impersonate and fake legitimate messages about that extensions, with the intention of deceiving the user and leading her to install malicious software. As an example, if a malicious website discovers that the user is using a concrete password management extension, it can create a fake window to ask the user to re-type her password. This attack is particularly severe in the case of Safari, since the attacker can actually access all the resources of an extension that leaks its `baseURI`. Hence, even a careful user who decides to analyze the website source cannot easily understand if a certain window or frame is created by an installed extension or by the site reusing the extension resources.

Table 3: Top 10 most Popular Extension Categories in the Chrome Store.

Category	% Usage
productivity	29.90
fun	10.45
communication	9.76
web development	7.74
accessibility	4.65
search tools	4.44
shopping	3.46
photos	3.12
news	2.40
sports	1.80

While the URI randomization control bypass does not provide a complete enumeration capability, when an extension leaks its random token it opens all its internal resources to the attacker. This is potentially very harmful as it increases the attack surface, allowing the attacker to access and exploit any vulnerability in one of the internal extension components. For example, Kotowicz and Osborn [23] presented a Chrome extension exploitation framework[14] that could be used when it was still possible to access all the different extension resources.

4.3 Viability Study

We have studied the viability of the estimated impact for several of the cases discussed before. In particular, we have analyzed their potential for performing analytics as well as the fingerprinting capability of extensions. We have omitted the malicious case studies due to their inherent ethical concerns. In addition, we believe that their implementations are more straightforward than in the proof-of-concept cases we tested and evaluated.

Analytics

In the case of the analytics capability of extensions, we have computed the popularity of the different categories

[14] `https://github.com/koto/xsschef`

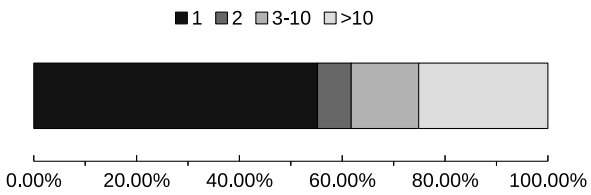

Figure 8: Distribution of anonymity set sizes regarding extensions.

established in the Chrome Web Store for each of the extensions that we previously analyzed in §3.1. In particular, we analyzed the 63 categories present in the 10,620 most popular Chrome extensions (Table 3 shows the 10 most popular categories).

The most popular category was "productivity" with 29.90% usage. Nevertheless, the definition of this category is not clear because it includes a wide-range of types of extensions such as ad blockers, schedulers, or office-related tools. Anyhow, a possible sub-categorization may be possible by means of the available description of each extension. The rest of the 10 most popular are more precise and may be helpful in order to perform analytics related tasks such as targeted advertisement or website personalization. For instance, the number of visitors with "shopping", "web development", or "sports" extensions, may help the website owner to personalize her content or ads accordingly, thus improving her number of visitors or ad revenues.

However, not only the most popular extensions may help the website owner to get a better understanding of her visitors and act accordingly. Indeed, less popular extensions, because their higher power of discrimination among users, can also be used for this task. For example, the usage of extensions from the "creative tools" category indicates that the visitor is prone to create content, the presence of extensions within "academic resources" category would likely indicate that the visitor is near the academic environment, "teacher tools" may imply that the visitor deliver at least some lectures, and "blogging" implies that the visitor is a blogger.

In summary, we believe that extensions are a powerful tool to perform fine-grained user analytics because of their diversity. Moreover, the information derived from the installed extensions of a web visitor, combined with the classical analytics information may lead to a better user analytics for website owners.

Device Fingerprinting

In order to understand and measure the capability of extensions for device fingerprinting, we implemented a page that checks the users' installed extensions among the top 1,000 most popular from the Chrome Web Store and the Add-ons Firefox websites, using the timing side-channel extension enumeration attack described in §3.1. Since our study involved the enumeration of several users' installed extensions, we informed the users about the procedure including the information gathered. Only after the user agrees to perform the experiment and share the collected information, the enumeration of her extensions is conducted. We also set a cookie on the user browser to prevent multiple resubmissions from the same user. In addition, to protect the user privacy, we only collected anonymous data.

Table 4: Comparison between Extensions with other Fingerprinting Attributes.

Method	**Entropy**
Extensions	0.869
List of Plugins	0.718
List of Fonts	0.548
User Agent	0.550
Canvas	0.475
Content Language	0.344
Screen Resolution	0.263

We disseminate the URL of the page through social networks and friends, asking them to participate in the study and further re-disseminate the link among their contacts. This way we collected the list of installed extensions from 204 participants from 16 different countries. Even though this number is smaller than in previous studies, we would like to remark that fingerprinting is not the actual goal of the paper but just a possible application of our attacks. In fact, this analysis is simply designed to determine the viability of our technique for device fingerprinting, either as a method by itself or by complementing other existing fingerprinting techniques.

Following the standard adopted in previous works [12, 24], we analyzed the extension anonymity sets of the fingerprinted users, which is defined as the number of users with the same fingerprint i.e., same extension set (the distribution of anonymity sets is shown in Figure 8). Overall, from the 204 users that participated in our study, 116 users presented a unique set of installed extensions, which means that 56.86% of the participants are uniquely identifiable just by using their set of extensions.

In addition, we also compare the discriminatory level of this proof-of-concept fingerprinting technique by computing its normalized Shannon Entropy [24] and comparing it with other fingerprinting attributes proposed in previous studies. In particular, Table 4 compares the different entropy values of the top six fingerprinting methods or attributes measured in the work by Laperdrix et al. [24] with our extensions-based fingerprinting method. We can notice that extensions pre-

Table 5: Current Browsers affected by our attacks. The last two lines refer to Extensions still under development.

Browser	Extensions Enumeration	Resource Access
Chromium Family	✓	
– Chrome	✓	
– Opera	✓	
– Yandex	✓	
...	✓	
Firefox Family	✓	
– Firefox Mobile	✓	
– Iceweasel	✓	
– Pale Moon	✓	
...	✓	
Safari	$\leq 40\%$	$\leq 40\%$
Microsoft Edge	in discussion	
Firefox WebExtensions	in discussion	

sented the highest entropy of the analyzed fingerprinting attributes — making them more precise than using the list of fonts or canvas-based techniques.

5 Vulnerability Disclosure and Countermeasures

5.1 Attack Coverage & Effects

In this paper we presented two different classes of attacks against the resource control policies adopted by all families of browsers on the market. Table 5 summarizes the overall impact of our methods.

As already mentioned, the coverage of our enumeration attack is complete in the case of the timing side-channel attack to access-control-based browser families (i.e., Chromium and Firefox Families) while approximately around 40% in URL randomization browsers (Safari).

Effects of Private Mode

"Incognito" or private mode is present in most of the modern browsers and it protects and restricts several accesses to the browser resources such as cookies or browsing history. Therefore, we decided to analyze if our attacks can enumerate extensions even when this mode is activated.

We discovered that all of our attacks accurately identified the list of installed extensions also within the private mode. This fact is due to several reasons. In the case of Chromium family browser, the browser checks for extensions in incognito mode, even though extensions are not allowed to access the websites [9]. Firefox and Safari did not include any check for extensions and, therefore, both websites and extensions are able to access each other.

```
1  GetFlagsFromPackage(const nsCString&
       aPackage,uint32_t* aFlags){
2    PackageEntry* entry;
3    if (!mPackagesHash.Get(aPackage, &
         entry))
4      return NS_ERROR_FILE_NOT_FOUND;
5    *aFlags = entry->flags;
6    return NS_OK;
7  }
8
9  GetSubstitutionInternal(const nsACString
       & root, nsIURI **result){
10     nsAutoCString uri;
11     if (!ResolveSpecialCases(root,
           NS_LITERAL_CSTRING("/"), uri)) {
12       return NS_ERROR_NOT_AVAILABLE;}
13     return NS_NewURI(result, uri);
14 }
```

Figure 9: Firefox functions that cause the difference between existing and not existing extensions.

```
3  const Extension* extension=
       RendererExtensionRegistry::Get()->
       GetExtensionOrAppByURL(resource_url)
       ;
4  if (!extension) {
5    return true;
6  }
```

Figure 10: Snip of Resource Request Policy function of Chromium that causes the difference between existing and not existing extensions (see Appendix for full code).

5.2 Timing Side-Channel Attack

The first class of attacks is the consequence of a poor implementation of the browser access control settings: Firefox-family browsers usage of extensions can be exploited to recognize the reason behind a failed resolution, and Chromium family timing-side channel allows an attacker to precisely tell apart the two individual checks performed by the browser engine.

The consequence, in both cases, is a perfect technique to enumerate all the extensions installed by the user. Given the open-source nature of these two browsers, we manually identified the functions responsible of the problem and indicated how to fix each of them.

Chromium family

We contacted the Chromium team to report the timing problem. The developers were quite surprised about the attack, because they believed that the time differences in

the checking phase were not significant enough to allow this type of timing side-channel attack. By inspecting the function responsible of checking the accessibility of a concrete extension path (see Figure 10), the two different steps described in section 3 can be clearly identified. First, the browser tests the existence of the extension (line #4) and finishes if the extension does not exist. If the extension does exist, it performs different checks to make sure that the path is accessible, returning a error message if it is not. These checks are the ones that permit the timing difference exploited in the attack.

We suggested a possible way to fix the code to avoid the time measurement by modifying the extension control mechanism to combine the internal extension verification and the resource check together in a single atomic operation (i.e., by modifying the extension existence check of line #4). This requires to replace the extension list with a hashtable containing the extensions and the full path of their resources.

While it may seem simple to fix the problem by making the check atomic, the problem remains if the attack is performed with real extension paths (easily obtainable) instead of fake paths. The timing difference would be the same as the one presented in Figure 4, with the only difference that the first check would validate the full path and not just the extension. At the time of writing, as it is a design-related problem, it is still not fixed.

In addition, as the new Firefox WebExtensions and Microsoft Edge (both currently in their early stages) use the same extension control mechanisms proposed by Chromium, we also notified their developers to make them aware of the issue described in this paper. We hope that our effort will help these two new versions to integrate by-design the necessary countermeasures to avoid these security problems since the beginning.

Firefox family

We also responsibly reported the Firefox non-WebExtensions problem that makes our enumeration attack possible to its developers, who acknowledged the issue and are currently discussing how to proceed. Specifically, Figure 9 show the function that causes the response difference regarding the extension existence.

The error returned when the resource path does not exist (line #4 and line #12 in Figure 9) does not raise any exception. Therefore, the solution is straightforward: return a `NS_ERROR_DOM_BAD_URI` error (i.e., the same one that is thrown when extension is not installed). This fix will not cause any issue to websites using extension paths, maintaining the functionality intact.

Regarding WebExtensions, the Firefox developers recently changed the way extensions are accessed in order to solve the timing side-channel and other related attacks. In particular, they changed the initial scheme (`moz-extension://[extID]/[path]`) to `moz-extension://[random-UUID]/[path]`. Unfortunately, while this change makes indeed more difficult to enumerate user extensions, it introduces a far more dangerous problem. In fact, the `random-UUID` token can now be used to precisely fingerprint users if it is leaked by an extensions. A website can retrieve this UUID and use it to uniquely identify the user, as once it is generated the random ID never changes. We reported this design-related bug to Firefox developers as well.

5.3 URI Leakage

The second class of attacks presented in the paper is quite different. In fact, the method that Safari's extension control employs to assure the proper accessibility of resources is, in principle, correct. However, Safari delegates to the extension developers the responsibility to keep the random URI secret. We believe that this is a very risky decision because most of the developers lack a proper understanding of the problem. As a consequence, our experiments confirm that a relevant number (40% in our preliminary experiments) of the extensions are likely to leak the `baseURI`, undermining the entire security solution. In particular, we discovered that important security extensions such as multiple password managers or advertisement blockers suffer from this `baseURI` leakage vulnerability and, hence, they are vulnerable to this attack. In the case of security extensions, this is particularly worrying due to the type of information they manage is usually very sensitive.

In this case the problem is even harder to solve, because it is not a consequence of an error in the extension control but of hundreds of errors spread over different extensions. Reaching out and training all the extension developers is a difficult task but Apple should provide more information on the proper way to handle the `baseURI` and about the security implications of this process.

In addition, we believe that Safari could benefit from adopting a lightweight static analysis solution (similar to the one we discuss in §3) to analyze the extensions in their market and flag those that leak the random token. This would allow to immediately identify potentially leaking extensions that may need a more accurate manual verification. In the meantime, we started reporting the problem to some security extensions we already manually confirmed, to help them solve their URI leakage problem.

5.4 Extension Security Proposal

In order to improve the security and privacy of browser extensions, we propose a solution that solves all the dif-

ferent problems presented in this paper.

1. All browsers should follow an extension schema that includes a random generated value in the URL: `X-extension://[randomValue]/[path]`. This random value should be modified across and during the same session and should be independent for each extension installed. For example, the browser should change it in every extension in every access. In this way, the random value cannot be used to fingerprint users.

2. Browsers should also implement an access control (such as `web_accessible_resource`) to avoid any undesirable access to all extensions resources even when the random value is unintentionally leaked.

3. Extensions should be analyzed for possible leakages before making them public to the users. Moreover, developer manuals should specifically discuss the problems that can cause the leakage of any random value generated.

6 Related Work

Security of Browser Extensions

The research community has made a large number of contributions analyzing the security properties of browsers extensions. A number of recent studies have focused on monitoring the runtime execution of browser extensions. Louw et al. [35, 36] proposed an integrity checker and a policy enforcement for Firefox legacy extensions. A more recent framework, Sentinel [31, 32], provided a fine-grained control to the users over legacy extensions, allowing them to define custom security policies while blocking common attacks to these extensions.

Other approaches have focused on providing security analysis of browsers extensions in order to discover security flaws. On the static analysis side, IBEX [18] is a framework to analyze security properties by means of a static methodology and it also allows developers to create a fine-grained access control and data-flow policies. VEX [3] is instead a static analyzer for Firefox JavaScript extensions that applies information flow analysis to identify browser extension vulnerabilities.

Dynamic extensions analysis includes the work of Djeric et al. [11], in which the authors proposed the use of dynamic analysis to track data inside the browser and detect malicious extensions. Dhawan et al. [10] proposed a similar approach to detect extensions that compromised the browser environment. In a similar vein, Wang et al. [39] used an instrumented browser to analyze Firefox Extensions. Hulk [21] is a dynamic analysis framework that controlled the activity of the browsing extensions, employing fuzzing techniques and HoneyPages adapted to the extensions. Hulk was used to analyze more than 48,000 Chrome extensions, discovering several malicious ones.

Despite the fact that these approaches are useful to detect malicious or compromised extensions, they are unfortunately useless against external attacks or information leakages. Our analysis has lead to the most complete set of attacks against resource accessibility control and `baseURI` randomization, allowing in both cases extension enumeration attacks that can be used as part of larger threats.

Similar to our own work, XHOUND [34] recently showed that the changes extensions perform on the DOM are enough to enumerate extensions. Using this technique, the authors also developed a new device fingerprinting technique and measured its impact. However, this approach has a much more limited applicability. In comparison, our techniques achieve a larger coverage, successfully enumerating 100% of the extensions for *access control* browsers and around 40% for those using *URI randomization*.

Web Timing Attacks

Web Timing attacks have been used for many different purposes, both in the client side and server side. Felten and Schneider [13] introduced this type of attacks as a tool to compromise users' private data and, specifically, their web-browsing history. In this way, a malicious attacker might obtain this information by leveraging the different forms of web browser cache techniques. By measuring the time needed to access certain data from an unrelated website, the researchers could determine if that specific data was cached or not, indicating a previous access.

Later, Bortz et al. [5] organized timing attacks in two different types of attacks: (i) direct timing, consisting in measuring the time difference in HTTP requests to websites and (ii) cross-site timing, which allows to obtain data from the client-side. The first type could expose website data that may be used to prove the validity of a username in certain secure website. The second type of attacks follow the same line of work of previous work by Felten and Schneider. They also performed some experiments that suggested that these timing vulnerabilities were more common than expected. In addition, Kotcher et al. [22] discovered that besides from the attacks previous discussed, the usage of CSS filters made possible the revelation of sensitive information such as text tokens exploiting time differences to render various DOM trees.

Two recent studies show that these attacks are far from being solved. Jia et al. [19] analyzed the possibility of determining the geo-locations of users thanks

to the customization of services performed by websites. Location-sensitive content is cached the same way as any other content. Therefore, a malicious actor can determine the victim's location by checking this concrete data and without relying in any other technique. Besides, Van Goethem et al. [38] proposed new timing techniques based on estimating the size of cross-origin resources. Since the measurement starts after the resources are downloaded, it does not suffer from unfavorable network conditions. The study also shows that these attacks could be used in various platforms, increasing the attack surface and the number of potential victims.

However, none of these timing techniques have been previously used to identify components of the web browser itself. Our new timing side-channel attacks are the first attacks capable of determining with 100% accuracy which extensions are installed in the browser, independently of the CPU usage.

7 Conclusions

Many different threats against the users security and privacy can benefit from a precise fingerprint of the extensions installed in the browser.

In this paper, we show that the current countermeasures adopted by all browser families are insufficient or erroneously implemented. In particular, we present a novel time side-channel attack against the access control settings used by the Chromium browser family. This technique is capable of correctly identifying any installed extension. Firefox WebExtensions and Microsoft Edge (early states) follow the same API and design, indicating that they may be prone to be vulnerable to the attack.

We also discuss a URI leakage technique that subverts the URI randomization mechanism implemented in Safari, that emerges from inappropriate extension implementations that leak the value of a random token. We implemented a new method to identify extensions with this potential leakage and we found out that up to 40% of Safari extensions could be vulnerable to this problem. After a manual inspection of security-related extensions, we discovered that many popular extensions are vulnerable to this attack. In addition, in the case of this attack, not only the extension is identified but also its resources can be accessed, posing as a more dangerous threat.

We also presented applications for our extension enumeration attacks. First, we propose different fingerprinting and user analytics techniques, demonstrating their feasibility in a real-world scenario. Second, we also proposed technique to use our enumeration techniques for malicious applications such as targeted malware, social engineering, or vulnerable extension exploitation.

We responsibly disclosed all our findings and we are now discussing with the developers of several browsers and extensions to propose the correct countermeasures to mitigate these attacks in both current and future versions.

Acknowledgments

This work is partially supported by the Basque Government under a pre-doctoral grant given to Iskander Sanchez-Rola.

References

[1] APPLE. Accessing Resources Within Your Extension Folder. `https://developer.apple.com/library/safari/documentation/Tools/Conceptual/SafariExtensionGuide/AccessingResourcesWithinYourExtensionFolder/AccessingResourcesWithinYourExtensionFolder.html`.

[2] APPLE. Safari Extensions Development Guide. `https://developer.apple.com/library/safari/documentation/Tools/Conceptual/SafariExtensionGuide`.

[3] BANDHAKAVI, S., TIKU, N., PITTMAN, W., KING, S. T., MADHUSUDAN, P., AND WINSLETT, M. Vetting browser extensions for security vulnerabilities with vex. *Communications of the ACM 54*, 9 (2011), 91–99.

[4] BARTH, A., FELT, A. P., SAXENA, P., AND BOODMAN, A. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* (2010).

[5] BORTZ, A., AND BONEH, D. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web (WWW)* (2007), ACM, pp. 621–628.

[6] BRYANT, M. Dirty browser enumeration tricks – using chrome:// and about: to detect firefox & addons. `https://thehackerblog.com/dirty-browser-enumeration-tricks-using-chrome-and-about-to-detect-firefox-plugins/index.html`.

[7] BUYUKKAYHAN, A. S., ONARLIOGLU, K., ROBERTSON, W., AND KIRDA, E. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In *Proceedings of the Network and Distributed System Security (NDSS)* (2016).

[8] CARLINI, N., FELT, A. P., AND WAGNER, D. An evaluation of the google chrome extension security architecture. In *Proceedings of the USENIX Security Symposium (SEC)* (2012).

[9] CHROMIUM. Extension in incognito. `https://blog.chromium.org/2010/06/extensions-in-incognito.html`.

[10] DHAWAN, M., AND GANAPATHY, V. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2009).

[11] DJERIC, V., AND GOEL, A. Securing script-based extensibility in web browsers. In *Proceedings of the USENIX Security Symposium (SEC)* (2010).

[12] ECKERSLEY, P. How unique is your web browser? In *Proceedings of the Privacy Enhancing Technologies (PETS)* (2010).

[13] FELTEN, E. W., AND SCHNEIDER, M. A. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security* (2000), ACM, pp. 25–32.

[14] GERDS, E. Plugindetect. http://www.pinlady.net/PluginDetect/.

[15] GOOGLE. Chrome Web Store. https://www.google.es/chrome/webstore/.

[16] GOOGLE. Manifest - web accessible resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[17] GOOGLE. What are extensions? https://developer.chrome.com/extensions.

[18] GUHA, A., FREDRIKSON, M., LIVSHITS, B., AND SWAMY, N. Verified security for browser extensions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2011).

[19] JIA, Y., DONG, X., LIANG, Z., AND SAXENA, P. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing 19*, 1 (2015), 44–53.

[20] K. KOTOWICZ. Intro to chrome add-ons hacking. http://blog.otowicz.net/2012/02/intro-to-chrome-addons-hacking.html.

[21] KAPRAVELOS, A., GRIER, C., CHACHRA, N., KRUEGEL, C., VIGNA, G., AND PAXSON, V. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the USENIX Security Symposium (SEC)* (2014).

[22] KOTCHER, R., PEI, Y., JUMDE, P., AND JACKSON, C. Cross-origin pixel stealing: timing attacks using css filters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1055–1062.

[23] KOTOWICZ, K., AND OSBORNAND, K. Advanced chrome extension exploitation. leveraging api powers for better evil. *Black Hat USA* (2012).

[24] LAPERDRIX, P., RUDAMETKIN, W., AND BAUDRY, B. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2016).

[25] LIU, L., ZHANG, X., YAN, G., AND CHEN, S. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* (2012).

[26] MOZILLA. Add-ons for Firefox. https://addons.mozilla.org/es/firefox/.

[27] MOZILLA. Chrome registration. https://developer.mozilla.org/en-US/docs/Chrome_Registration.

[28] MOZILLA. JetPack Project. https://wiki.mozilla.org/Jetpack.

[29] MOZILLA. WebExtension Add-ons. https://developer.mozilla.org/en-US/Add-ons/WebExtensions.

[30] MOZILLA. XPCOM Reference. https://developer.mozilla.org/en/docs/Mozilla/Tech/XPCOM/Reference.

[31] ONARLIOGLU, K., BATTAL, M., ROBERTSON, W., AND KIRDA, E. Securing legacy firefox extensions with SENTINEL. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (2013).

[32] ONARLIOGLU, K., BUYUKKAYHAN, A. S., ROBERTSON, W., AND KIRDA, E. Sentinel: Securing legacy firefox extensions. *Computers & Security 49* (2015), 147–161.

[33] SECURITY RESPONSE, SYMANTEC. The Waterbug attack group. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/waterbug-attack-group.pdf, 2015.

[34] STAROV, O., AND NIKIFORAKIS, N. Xhound: Quantifying the fingerprintability of browser extensions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2017).

[35] TER LOUW, M., LIM, J. S., AND VENKATAKRISHNAN, V. Extensible web browser security. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (2007).

[36] TER LOUW, M., LIM, J. S., AND VENKATAKRISHNAN, V. Enhancing web browser security against malware extensions. *Journal in Computer Virology 4*, 3 (2008), 179–195.

[37] THREAT INTELLIGENCE, FIREEYE. Pinpointing Targets: Exploiting Web Analytics to Ensnare Victims. https://www2.fireeye.com/rs/848-DID-242/images/rpt-witchcoven.pdf, 2015.

[38] VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1382–1393.

[39] WANG, L., XIANG, J., JING, J., AND ZHANG, L. Towards fine-grained access control on browser extensions. In *Proceedings of the International Conference on Information Security Practice and Experience* (2012).

Appendix

```
1  bool ResourceRequestPolicy::
       CanRequestResource( const GURL&
       resource_url, blink::WebFrame* frame
       , ui::PageTransition transition_type
       ) {
2    CHECK(resource_url.SchemeIs(
         kExtensionScheme));
3    const Extension* extension =
         RendererExtensionRegistry::Get()->
         GetExtensionOrAppByURL(
         resource_url);
4    if (!extension) {
5      return true;
6    }
7    std::string
         resource_root_relative_path =
8        resource_url.path().empty() ? std
             ::string()
9        : resource_url.path().substr(1);
10   if (extension->is_hosted_app() && !
         IconsInfo::GetIcons(extension).
         ContainsPath(
         resource_root_relative_path)) {
11     LOG(ERROR) << "Denying load of " <<
           resource_url.spec() << " from "
           << "hosted app.";
12     return false;
13   }
14   if (!WebAccessibleResourcesInfo::
         IsResourceWebAccessible(extension,
          resource_url.path()) && !
         WebviewInfo::
         IsResourceWebviewAccessible(
         extension, dispatcher_->
         webview_partition_id(),
         resource_url.path())) {
15     GURL frame_url = frame->document().
           url();
16     GURL page_origin = ablink::
           WebStringToGURL(frame->top()->
           getSecurityOrigin().toString());
```

Figure 11: Resource Request Policy function of Chromium that causes the difference between existing and not existing extensions (part 1)

```
17     bool is_empty_origin = frame_url.
           is_empty();
18     bool is_own_resource = frame_url.
           GetOrigin() == extension->url()
           || page_origin == extension->url
           ();
19     bool is_dev_tools = page_origin.
           SchemeIs(content::
           kChromeDevToolsScheme) && !
           chrome_manifest_urls::
           GetDevToolsPage(extension).
           is_empty();
20     bool transition_allowed = !ui::
           PageTransitionIsWebTriggerable(
           transition_type);
21     bool is_error_page = frame_url ==
           GURL(content::
           kUnreachableWebDataURL);
22
23     if (!is_empty_origin && !
           is_own_resource && !is_dev_tools
            && !transition_allowed && !
           is_error_page) {
24       std::string message = base::
             StringPrintf("Denying load of
             %s. Resources must be listed
             in the
             web_accessible_resources
             manifest key in order to be
             loaded by pages outside the
             extension.", resource_url.spec
             ().c_str());
25       frame->addMessageToConsole(
26       blink::WebConsoleMessage(blink::
             WebConsoleMessage::LevelError,
              blink::WebString::fromUTF8(
             message)));
27       return false;
28     }
29   }
30   return true;
31 }
```

Figure 12: Resource Request Policy function of Chromium that causes the difference between existing and not existing extensions (part 2)