Web applications security and privacy

Dolière Francis Some

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Sécurité et vie privée dans les applications web

Dolière Francis Somé

Université Côte d'Azur / Inria

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur

Dirigée par : Tamara Rezk
Co-encadrée par : Nataliia Bielova

Soutenue le : 29 octobre 2018

Devant le jury, composé de :

Davide Balzarotti	Professeur	Eurecom
Nataliia Bielova	Chercheur	Inria
Stefano Calzavara	Chercheur	Università Ca' Foscari Venezia
Walid Dabbous	Directeur de recherche	Inria
Christoph Kerschbaumer	Chercheur	Mozilla
Nick Nikiforakis	Professeur Assistant	Stony Brook University
Tamara Rezk	Chercheur	Inria
Andrei Sabelfeld	Professeur	Chalmers University
Mike West	Ingénieur	Google

Sécurité et vie privée dans les applications web

Composition du jury

Rapporteurs :		
Davide Balzarotti	Professeur	Eurecom
Andrei Sabelfeld	Professeur	Chalmers University
Examinateurs :		
Stefano Calzavara	Chercheur	Università Ca' Foscari Venezia
Walid Dabbous	Directeur de recherche	Inria
Christoph Kerschbaumer	Chercheur	Mozilla
Mike West	Ingénieur	Google
Invité :		
Nick Nikiforakis	Professeur Assistant	Stony Brook University
Co-directeur de thèse:		
Nataliia Bielova	Chercheur	Inria
Directeur de thèse:		
Tamara Rezk	Chercheur	Inria

Résumé

Dans cette thèse, nous nous sommes intéressés aux problématiques de sécurité et de confidentialité liées à l'utilisation d'applications web et à l'installation d'extensions de navigateurs. Parmi les attaques dont sont victimes les applications web, il y a celles très connues de type XSS (ou Cross-Site Scripting). Les extensions sont des logiciels tiers que les utilisateurs peuvent installer afin de booster les fonctionnalités des navigateurs et améliorer leur expérience utilisateur.
Content Security Policy (CSP) est une politique de sécurité qui a été proposée pour contrer les attaques de type XSS. La Same Origin Policy (SOP) est une politique de sécurité fondamentale des navigateurs, régissant les interactions entre applications web. Par exemple, elle ne permet pas qu'une application accède aux données d'une autre application. Cependant, le mécanisme de Cross-Origin Resource Sharing (CORS) peut être implémenté par des applications désirant échanger des données entre elles.
Tout d'abord, nous avons étudié l'intégration de CSP avec la Same Origin Policy (SOP) et démontré que SOP peut rendre CSP inefficace, surtout quand une application web ne protège pas toutes ses pages avec CSP, et qu'une page avec CSP imbrique ou est imbriquée dans une autre page sans ou avec un CSP différent et inefficace. Nous avons aussi élucidé la sémantique de CSP, en particulier les différences entre ses 3 versions, et leurs implémentations dans les navigateurs. Nous avons ainsi introduit le concept de CSP sans dépendances qui assure à une application la même protection contre les attaques, quelque soit le navigateur dans lequel elle s'exécute. Finalement, nous avons proposé et démontré comment étendre CSP dans son état actuel, afin de pallier à nombre de ses limitations qui ont été révélées dans d'autres études.
Les contenus tiers dans les applications web permettent aux propriétaires de ces contenus de pister les utilisateurs quand ils naviguent sur le web. Pour éviter cela, nous avons introduit une nouvelle architecture web qui une fois déployée, supprime le pistage des utilisateurs.
Dans un dernier temps, nous nous sommes intéressés aux extensions de navigateurs. Nous avons d'abord démontré que les extensions qu'un utilisateur installe et/ou les applications web auxquelles il se connecte, peuvent le distinguer d'autres utilisateurs. Nous avons aussi étudié les interactions entre extensions et applications web. Ainsi avons-nous trouvé plusieurs extensions dont les privilèges peuvent être exploités par des sites web afin d'accéder à des données sensibles de l'utilisateur. Par exemple, certaines extensions permettent à des applications web d'accéder aux contenus d'autres applications, bien que cela soit normalement interdit par la Same Origin Policy. Finalement, nous avons aussi trouvé qu'un grand nombre d'extensions a la possibilité de désactiver la Same Origin Policy dans le navigateur, en manipulant les entêtes CORS. Cela permet à un attaquant d'accéder aux données de l'utilisateur dans n'importe qu'elle autre application, comme par example ses mails, son profile sur les réseaux sociaux, et bien plus. Pour lutter contre ces problèmes, nous préconisons aux navigateurs un système de permissions plus fin et une analyse d'extensions plus poussée, afin d'alerter les utilisateurs des dangers réels liés aux extensions.

Mots-clés : web, navigateurs, applications web, securité, same-rigin policy, content security policy, cross-origin resource sharing, extensions de navigateurs, communication inter-iframes, confidentialité, vie privée, pistage, empreinte de navigateurs

Abstract

In this thesis, we studied security and privacy threats in web applications and browser extensions. There are many attacks targeting the web of which XSS (Cross-Site Scripting) is one of the most notorious. Third party tracking is the ability of an attacker to benefit from its presence in many web applications in order to track the user has she browses the web, and build her browsing profile. Extensions are third party software that users install to extend their browser functionality and improve their browsing experience. Malicious or poorly programmed extensions can be exploited by attackers in web applications, in order to benefit from extensions privileged capabilities and access sensitive user information.
Content Security Policy (CSP) is a security mechanism for mitigating the impact of content injection attacks in general and in particular XSS. The Same Origin Policy (SOP) is a security mechanism implemented by browsers to isolate web applications of different origins from one another.
In a first work on CSP, we analyzed the interplay of CSP with SOP and demonstrated that the latter allows the former to be bypassed. Then we scrutinized the three CSP versions and found that a CSP is differently interpreted depending on the browser, the version of CSP it implements, and how compliant the implementation is with respect to the specification. To help developers deploy effective policies that encompass all these differences in CSP versions and browsers implementations, we proposed the deployment of dependency-free policies that effectively protect against attacks in all browsers. Finally, previous studies have identified many limitations of CSP. We reviewed the different solutions proposed in the wild, and showed that they do not fully mitigate the identified shortcomings of CSP. Therefore, we proposed to extend the CSP specification, and showed the feasibility of our proposals with an example of implementation.
Regarding third party tracking, we introduced and implemented a tracking preserving architecture, that can be deployed by web developers willing to include third party content in their applications while preventing tracking. Intuitively, third party requests are automatically routed to a trusted middle party server which removes tracking information from the requests.
Finally considering browser extensions, we first showed that the extensions that users install and the websites they are logged into, can serve to uniquely identify and track them. We then studied the communications between browser extensions and web applications and demonstrate that malicious or poorly programmed extensions can be exploited by web applications to benefit from extensions privileged capabilities. Also, we demonstrated that extensions can disable the Same Origin Policy by tampering with CORS headers. All this enables web applications to read sensitive user information. To mitigate these threats, we proposed countermeasures and a more fine-grained permissions system and review process for browser extensions. We believe that this can help browser vendors identify malicious extensions and warn users about the threats posed by extensions they install.

Keywords: web, browser, web application, security, same origin policy, content security policy, cross-origin resource sharing, browser extensions, cross-iframe communication, privacy, third party web tracking, browser fingerprinting

Acknowledgements

MERCI
THANK YOU
BARKA

Contents

Chapter 1

Introduction

Browsers are everywhere. Billions of devices such as personal computers, smartphone, tablets, and even TVs connect to the Internet via powerful browsers, that are able to display rich web applications. Differently from native applications, web applications remove the burden for developers to provide a specific version of their applications for different devices. This allows developers to reach almost any device and many more users. This is possible because the web is based on standards supported by browsers and used by developers to write their applications. Web standards such as HTML5, CUSS, JavaScript and HTTP are powerful enough to enable the creation of web applications which have nothing to envy of traditional web applications, in terms of features, functionality and performance. In recent years, many applications such as Microsoft Office and Skype, that have dominated the desktop applications landscape, are experiencing serious competition from alternatives based on the web, and are themselves now moving to the web.
Long ago, websites were made of static content solely provided by the owner of the site. Today web applications are interactive, and are made by reusing content and assembling building blocks provided by third parties. Using third party content make it easy to quickly build fully fledged web applications: a restaurant site can use Google Maps to show its location to clients, link to a social network page for users to leave comments, collect feedback from users directly in the site, or even monetize the site by displaying third party advertisements to users.
Hence, part of the content that form web applications are also generated by users. By creating accounts on web applications, users can leave comments which are displayed to other users. They can interact and share content with one another.

1 Security threats

It is fundamental that the data a user entrusts to a web application does not get leaked to a third party. The Same Origin Policy (SOP) [125] is a baseline security mechanism, implemented by browsers, to ensure that websites they run cannot directly access each other's data: for instance, the user bank information is accessible only to the bank website, and not to her email website and vice versa. To ensure that, each application executes in a different browsing context. However, when an application embeds third party content in its pages, say a script, then the third party content executes with the same privileges (in the same context) as any other content provided by the owner of the application. In other words, it has access to any data of the hosting application. It is then the responsibility of the developer to ensure that it is safe to include such third party content in his website —

that the third party is not leaking user data.
If third party content present in a website is usually included by the developer because she trusts it, there are many attacks that can be leveraged by an attacker to inject malicious content in web applications. One of the most prevalent of those attacks is Cross-Site Scripting (XSS). An XSS attack occurs when an attacker is able to inject malicious content in a vulnerable application via a comment text field for instance. When such content is displayed to other users (going to the application to view the list of posted comments), the attacker content (code) is also displayed. Since it is a code and not an expected text content, the code is executed, with the same privileges as the code provided by the application developer. The attacker can then take advantage of this position to access and exfiltrate user data, leak user authentication cookies and mount session hijacking attacks in order to take actions on the user's behalf. To help mitigate XSS attacks in particular and a broad range of content injection attacks in general, the World Wide Web Consortium (W3C) introduced Content Security Policy (CSP) [275]. CSP makes it possible for a developer to declare the origins of (trusted) content allowed to load in her application. At runtime, content not whitelisted in the CSP of the application are blocked by the browser. Thanks to CSP, the browser can block content injected by an attacker who exploited a vulnerability in the website.

2 Privacy threats

The Hypertext Transfer Protocol (HTTP) used to exchange data between web browsers and application servers is coined as being `stateless`. In other words, when a browser connects to a server to retrieve some data, the server just handles the request, responds with the appropriate content and forgets about the request it has just served and the browser which made it. Cookies have been added to the protocol to make it `stateful`. Cookies are sent by web servers, stored in the user browser and attached to future requests to the same server, so that the server can link subsequent requests from the same browser. The ability to store cookies in the user browser is primarily meant to (but not limited to) first parties (web applications the user interacts with). Indeed, every third party content that web applications embed can also use cookies in order for the third party content provider to recognize the browser from which a request is being made to load the third party content. Additionally, the browser automatically attaches to third party requests, the first party application in which they are embedded. Combining cookies and the name of the first party website allows a third party to build a browsing history for the specific user (all the websites that include content from the third party content, and that the user has visited). Many third parties serve content to numerous web applications, putting them in a position where they can track many users as they browse the web. Tracking information gathered by third parties can serve various purposes such as advertisement, or put the user privacy at risk by revealing their health situation, political or religious views, interests, etc. Another tracking scenario that has been extensively demonstrated in the literature is browser fingerprinting. In this scenario, trackers collect the properties of the user browser (for instance, the name, version of the browser and operating system, list of fonts and plugins installed), build a fingerprint of her browser and store it on a server. When the user visits another website, the tracker collects again the properties of her browser, and compares it with the previously stored fingerprints in order to recognize and track the user.

3 Extensible browsers

Even though modern web browsers are powerful platforms able of executing all sorts of simple to very complex web applications, most of them also provide mechanisms for users to further extend and customize their browsers. One of the most currently widespread mechanism for doing so is based on browser extensions or addons. The WebExtensions API for instance, is a cross-browser extension API sported by major browsers including Chrome, Firefox, Opera and Microsoft Edge. Nowadays, extensions are very widespread among browsers. The Google Chrome Web Store has more than 60k extensions, and there are hundreds to thousands of extensions available for other platforms. Many extensions are already used by dozens of millions of users. Among those are adblockers and other privacy-preserving extensions, more and more passwords managers and various helper extensions for improving and easing users' browsing experience. In contrast to plugins such as Adobe Flash, Java Applets that are external native applications used by browsers to display non-standard web content (Flash movies, Java applets), browser extensions however are third party programs, tightly integrated to browsers, where they execute with elevated privileges. For instance, unlike traditional web applications, they are not subject to the SOP, and therefore can access user data on any web application, including applications where users are logged into. Extensions can also intercept and tamper with HTTP communications between web applications and web servers.
Due to their privileged nature, browsers extensions are the targets of many attacks that put the security and privacy of users at risk. Compromising an extension gives an attacker unlimited access to user data on any application, including sensitive information on any application they are logged to.
For security reasons, browser extensions and web applications execute in separate contexts. Web applications cannot access extensions privileged execution contexts. Extensions however have access to web applications execution contexts and in particular to their Document Object Model (DOM), which they can manipulate. They can add, modify or remove elements from the DOM of web pages. Content they inject directly in the DOM of web pages will be considered as part of the web application, thus executing with the same privileges and accessing data in the application context. Apart from the DOM, extensions and web applications can interact in various ways in order to exchange data via the localStorage, or by setting up communication channels using `postMessage`-like APIs. Extensions can also intercept and modify the headers of HTTP exchanges between web applications and web servers. This include sensitive security critical headers such as the Cross-Origin Resource Sharing (CORS) ones, used in HTTP requests and responses to authorize or not cross-domain requests.

4 Scope of the thesis

4.1 Improving the effectiveness of CSP

Content Security Policy is a page-specific policy. This means that when a web application does not protect all its pages with a CSP, then instead of targeting the CSP-protected pages, an attacker can target the non-protected ones. Once the attack succeeds, the SOP allows the attacker to propagate this attack to all other pages of the application, including those which are CSP-protected.
Since its introduction, CSP has experienced three major versions: the second version is a W3C specification and the current third version is already in an advanced development

state. New versions of CSP come with their set of features which were not present in previous versions and changes that alter the semantics of previous CSP versions. Furthermore, different browsers support different versions of CSP. Even when they support the same version, their implementations are not always compliant with the specification or even with one another. In these settings, developers have to ensure that policies they deploy take into consideration the peculiarities of each CSP version, and browsers implementations, so that the CSPs provide the same security protection while preserving the functionality of the application, in all browsers. To help developers cope with these intricacies, we formalize the differences between CSP versions and browsers implementations, and propose and prove a set of rewriting rules to effectively build policies which are independent of CSP versions and browsers implementations.
Previous studies have shown the limitations of CSP at mitigating XSS attacks, because of subtle bypasses such as JSONP and open redirects. We reviewed the previous solutions that have been proposed and showed that they do not fully mitigate the identified shortcomings of CSP. Therefore, we proposed to extend the CSP specification. We motivate the need for a blacklisting mechanism in CSP, a URLs parameter filtering mechanism, new directives for explicitly preventing redirections, and an efficient reporting mechanism for collecting feedback of the runtime enforcement of CSP. We demonstrate the feasibility of our proposals with an example of implementation.

4.2 Server-side third party tracking protection

Third party web tracking has attracted significant attention from the research community the past years. Many mechanisms have been proposed to help mitigate them. We observe that most of these solutions are focused on the client-side, proposed by browser vendors in the form of settings for users to control third party cookies, browse in private and other incognito modes, or in the form of privacy-preserving browser extensions provided by developers. If some browsers enable some basic tracking protection features by default, enabling more advanced ones is not always easily accessible to the average Internet user. Furthermore, installing browser extensions is not easy to the majority of users. In other words, these solutions are only effective for advanced users, leaving the vast majority of them unprotected.
We propose a server-side third party web tracking protection mechanism. The design of such a solution is challenging for many reasons: first of all, third party content are important and many web applications in the wild cannot afford not using them. Nonetheless, we argue that a developer embedding third party content is more interested in the content itself, rather than in the underlying tracking that can take place. As such, this is a practice that developers may want to remove from third party content embedded in their applications. Moreover, many web applications are already deployed, and a solution requiring significant effort from developers to change their applications is likely going to not be considered by the majority of them.
Considering these requirements, the solution we propose can be plugged to an already existing web application, and does not prevent the developer from using third party content. The system automatically rewrites the original pages of the application, in order to redirect third party content requests to a middle party. There, any tracking information (cookies, identification of the first party) is removed, and the request is forwarded to the third party. In other words, the middle party anonymizes the requests and forwards them to the third party. When the third party replies, the responses are also anonymized before being returned to the browser. This makes it impossible for the third party to recognize

the user's browser initiating the request, and thus prevents third party web tracking.

4.3 Web applications meet browser extensions

Browsers supporting the WebExtensions API usually assign to each extension, a unique identifier that distinguishes it from other extensions. Chrome and Opera assigns each extension a permanent identifier, which is the same in all user browsers. Firefox however assigns a random identifier to the extension on a per-browser basis. Content that extensions inject in web pages DOM can either be hosted on a remote server, or be located in the extension package on the user browser. Content of extensions package that can be injected in web pages are referred to as `web accessible resources`. Extensions identifiers, web accessible resources and content that extensions inject in web pages can be used to successfully discover extensions and fingerprint the user's browser. Previous studies that have quantified the fingerprintability of browser extensions have done so with rather limited user bases (less than a thousand). We performed an analysis of the fingerprintability of users based on their set of extensions by analyzing the list of installed extensions from more than 16k users. We also show that the websites to which users are logged into further add to the uniqueness of users.
We analyzed the communications between browser extensions and web applications, and discovered many extensions which privileged capabilities can be exploited by web applications in order for instance to bypass the SOP and access any other web application data, evade user privacy preferences of not being tracked by storing data in extensions permanent storage, access user browsing cookies, history, bookmarks, topsites, list of installed extensions, or even download files and save them on the the user's device.
Finally, we discussed the security implications of the ability of extensions to tamper with HTTP headers, in particular security-critical headers such as those used in Cross-Origin Resource Sharing (CORS) requests. In fact, by default, the SOP does not allow cross-origin requests. CORS is a refinement of the Same Origin in which web servers can authorize cross-origin requests by sending dedicated HTTP headers. An extension can basically disable the SOP in browsers by appropriately tampering with CORS headers in cross-origin requests. Consequently, by acting as man-in-the-middle, they can authorize any unauthorized cross-origin requests, by allowing an attacker to transparently gather all user data on any web application.

5 Outline

In the background Chapter 2, we describe concepts and technologies necessary to ease the reader's understanding of other chapters. The rest of the thesis is organized in 3 parts. Part I describes works related to Content Security Policy. It is further divided in 3 chapters describing CSP violations due to the Same Origin Policy (Chapter 3), dependencies in CSP directives (Chapter 4), and extensions we propose to complement and extend CSP (Chapter 5). Part II is related to third party web tracking and browser fingerprinting. Chapter 6 presents our work on server-side tracking prevention and Chapter 7 discusses browser fingerprinting based on browser extensions and web logins. Finally, Part III is dedicated to works on the security and privacy implications of browser extensions. Chapter 8 presents our work analyzing the threats posed by the communications between browser extensions and web applications, and Chapter 9 discusses the implications of the Cross-Origin Resource Sharing (CORS) headers manipulations by browser extensions. Finally, Chapter 10 concludes.

6 List of publications and submissions

1. Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. On the content security policy violations due to the same-origin policy. In Barrett et al. [171], pages 877–886
2. Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. Control what you include! - server-side protection against third party web tracking. In Bodden et al. [174], pages 115–132
3. Gábor György Gulyás, Dolière Francis Somé, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. In *To appear in the Proceedings of the 2018 ACM on Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, Canada, October 15 - 19, 2018*, 2018
4. Dolière Francis Somé and Tamara Rezk. DF-CSP: Dependency-Free Content Security Policy. Submitted for review
5. Dolière Francis Somé and Tamara Rezk. Extending Content Security Policy: Blacklisting, URL arguments filtering and Monitoring. Submitted for review
6. Dolière Francis Somé. EmPoWeb: Empowering web applications with browser extensions. Submitted for review
7. Dolière Francis Somé. Breaking the Same Origin Policy for free - On CORS headers manipulations by browser extensions. Submitted for review

Chapter 2

Background

This chapter introduces different concepts and technologies that are used throughout this thesis. It presents the state-of-the-art of the these key concepts so as to ease the reader's understanding of the following chapters and works presented in this thesis.

1 Web applications ecosystem

Browsers are application platforms able of running web applications, that provide different services to users. The web ecosystem is built on common standards, implemented both by browser vendors and web applications developers. This makes it possible for any browser to correctly render any website.
The web architecture is commonly referred to as a client-server architecture, where the browser is the client and the server is the machine hosting the application. When a user needs to interact with an application, the browser makes a request to the server hosting it. The server responds with the application, which is then displayed to the user. The web architecture can also be seen as a multi-tier system [192,247], where the browser is the first or presentation tier and the web server represents the logic tier, associated to data tiers which are usually databases where the server stores and manages the web application data. To communicate with one another, the different tiers are connected to a common network, usually the Internet, and support common communications protocols such as HTTP [1] used for transmitting data among them.

1.1 Web servers

A web server is often wrongly referred to as the computer device (machine) hosting a web application. Actually, a web server is itself an application running on a computer device, connected to a network (usually the Internet), in order for the clients (browsers) to communicate with the web server. Well-known web servers are Apache HTTP Server [9], Microsoft Internet Information Services (IIS) [95] and Nginx [104]. Web servers support a set of technologies for powering web applications. They usually support a programming language used for generating webpages and handling requests from clients. These are called server-side programming languages and include for instance PHP [113], ASP.NET [11], Python [119], Node.js [105]. The server-side language can be used to serve static resources, or dynamically generate resources and webpages on the fly, based on the characteristics of a request. Web servers are also usually associated with databases used to manage web applications and user data. Well known database management systems are MySQL [103], PostgreSQL [115], Oracle [111], MongoDB [99].

1.2 HTTP protocol

HTTP (HyperText Transfer Protocol) [1] is one of the most used protocols for the communications between web browsers (clients) and web servers. It usually involves a request, sent from the client (browser) to the server which processes the request and responds. Among other things, HTTP requests carry information such as (i) the URL (Uniform Resource Locator) [144] [1] or web address of the resource to be accessed on the server; (ii) a method (`GET`, `POST`, `OPTIONS`, `HEAD`, etc.) which indicates the action to be applied to the resource; (iii) a list of key/value pair HTTP headers providing additional information about the requests, and eventually (iv) data (request body) to be transmitted with the requests. Similarly, HTTP responses from the server contain a status code (200, 404, etc.) that indicates whether the requested resource is available or not, a list of response headers and the response body (data).

Request headers
Host: www.google.com
User-Agent: Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Cookie: NID=1234

Response headers
Content-Type: text/html; charset=UTF-8
X-Frame-Options: SAMEORIGIN
Set-Cookie: JAR=abcd; expires=Thu, 21-Feb-2019 14:39:31 GMT; path=/; domain=.google.com; HttpOnly; Secure
Content-Security-Policy: default-src 'self'; frame-ancestors 'self'

Table 2.1 – HTTP headers (excerpt) exchanged between the browser (client) and the server for an access to `https://www.google.com`

Table 2.1 shows an excerpt of HTTP headers exchanged between the browser and the server for an access to `https://www.google.com` [2]. Among the request headers are the `User-Agent` header and its value `Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0` which means that the user has a Mozilla Firefox version 61.0 browser running on a Linux (Fedora) computer. The browser is also transmitting cookies (using the `Cookie` header) to the server. The `content-type` response header tells that the accessed resource should be handled by the browser as an HTML content (that is to say, a webpage).
HTTP communications are transmitted on the network unencrypted. This open up to man-in-the-middle attacks [91], where an attacker can intercept and modify the communications between a client and a server. HTTP protocol is then referred to as an insecure protocol. HTTPS [75] is the secure counterpart of HTTP, where communications are encrypted between the source and the destination. In this work, we often use HTTP to mean the protocol in general including its insecure and secure counterparts.

HTTP cookies The HTTP protocol is coined as being `stateless` [73]. In other words, when a browser connects to a server to retrieve some data, the server just handles the

1. URLs are translated into IP addresses
2. For the purposes of this work, some headers have been added and some header values changed

request, responds with the appropriate content and forgets about the request it has just served and the browser which made it. Therefore the server will not make a link between future requests from the same browser and previous ones [73]. However, most modern applications rely on the ability to recognize the user's browser that connect to the web application servers. In this light, HTTP cookies have been introduced to make HTTP `stateful`. A cookie is a piece of information (an identifier) sent by a server in an HTTP header, i.e. `Set-Cookie` in Table 2.1, to be stored in a user browser and attached to subsequent requests (using the `Cookie` header) between the browser and the server. The first time a user visits a website, the server returns the requested data as well as the cookies (using the `Set-Cookie` header). Later on, future requests to the same server will automatically be attached the cookies previously stored by the server, allowing the latter to recognize the user whose browser is connecting to the server. HTTP cookies have made viable web applications such as e-commerce, and most of the web applications make extensive use of cookies in order to provide their services to users. In a typical e-commerce website for instance, a cookie is attached to a user browser, allowing the website to track the list of items the user adds to her basket. On a social network website for example, once a user is logged in (using her credentials, i.e. her username and password), the server will store a cookie in the user's browser and will no long ask for her credentials. Hence, each time the user accesses a new page or service of the app, the cookie is used to authenticate her, in order to authorize or not, the access. In Table 2.1 the browser is sending a cookie previously stored by the server using the `cookie` request header.
Cookies are organized on a per-domain basis. Listing 2.1 below shows an example of a cookie to be set for `google.com` and its subdomains.

```
set-cookie: JAR=abcd; expires=Thu, 21-Feb-2019 14:39:31 GMT;
   path=/; domain=.google.com; HttpOnly; Secure
```

Listing 2.1 – Example of a cookie to be set for `google.com` and its subdomains

A variety of information compose the cookie header. The cookie name is `JAR` and its value is `abcd`. It expires on the 21st of February 2019. After this period, the browser will no longer attach this cookie to requests. The part `domain=.google.com` means that the cookie will be attached to requests to `google.com` domain and also its subdomains (Section 2 discusses domains and subdomains). The flag `HttpOnly` instructs the browser that the cookie must only be attached to HTTP requests, and not exposed to scripts running in webpages from `google.com` and its subdomains. The `Secure` flag means that cookies must be attached to HTTPS requests only, and not to insecure HTTP requests.

1.3 Web documents

Web applications are composed of a set of web documents typically written according to the HyperText Markup Language (HTML) standard [70]. In its current status, the standard is known as HTML5. Web documents are also usually referred to as HTML documents, pages or webpages. HTML is a markup language and elements that compose the structure of a webpage are described with HTML tags. The specification defines among other things, the semantics of the different tags, the attributes that can be added to the tags (or elements), and the ability to nest tags in order to create more and more complex webpages. Tags are typically used in a pair of markups, in particular when the tag can nest other tags or contain text: the start (opening) tag which contains the attributes, and the end (closing) tag specifying where the tag declaration stops. Otherwise, when the tag does not accept any nested tag or text, the closing tag is usually omitted. Nested elements are called the children, and the nesting element is the parent.

```
<html>
  <head>
    <title>Title of the page</title>
    <script src="http://third.com/script.js"></script>
    <link rel="stylesheet" href="http://third.com/styles.css">
  </head>
  <body>
    <p>First paragraph</p>
    <img src="http://third.com/picture.png" alt="Picture">
    <iframe src="stylesheet" href="http://third.com/iframe.png">
       </iframe>
  </body>
</html>
```

Listing 2.2 – Example of an HTML document

Listing 2.2 shows an example of a simple HTML document. The `html` tag is the root element of the document in which are nested the `head` and `body` elements. The title of the page, declared with the `title` element is nested inside the `head` element. The latter also nests a `script` and a `link` elements with different attributes `src, rel, href` and their related values. In the `body` element is declared a `p` tag which indicates a paragraph in the document, an `img` element which declares an image, and an `iframe` element than embeds another web page in the first page.
To render a webpage (produce the user interface or UI), the browser parses it and produces a Document Object Model (DOM) [48]. Then each portion of the document is rendered according to its semantics as defined by the HTML standard. The rendering of some elements may require the browser to make an HTTP request to fetch their content. This is the case for instance, when the browser encounters a `script, img, iframe` element with a `src` attribute, or a `link` element with a `href` attribute. Many other elements (`audio, video, source, ...`) may also require the browser to make HTTP requests in order to render them.
In order to build rich web applications as we know today, HTML is usually further associated with other technologies.

1.4 Cascading Style Sheets (CSS)

The Cascading Style Sheets (CSS) [22] is widely used to describe the presentation (formatting and appearance) of HTML elements in a web document: for instance, the color, fonts, size to use in order to display elements.
CSS files are included in web documents with the `link` tag (See Listing 2.2), or with the `style` tag or even set with the `style` attribute of HTML elements.

```
@font-face {
  font-family: "CustomFont";
  src: url("/fonts/customfont.woff2") format("woff2"),
  url("/fonts/customfont.woff") format("woff");
}

div {
    color:black;
    font-family: "CustomFont";
    font-size: 14px;
    background-image:url(http://example.com/background.png);
```

```
}
```

Listing 2.3 – Example of a stylesheet.

Listing 2.3 shows an example of a stylesheet. It loads a font (`CustomFont`) and applies it to `<div>` elements. It also the defines the color, font size and background image of all `<div>` elements.

1.5 JavaScript

JavaScript (JS) [50] is a prototype-based, multi-paradigm, dynamic scripting language, losely-typed, interpreted or JIT-compiled programming language, supporting object-oriented and imperative programming styles, with first-class functions and closures [80]. The language defines different constructs such as literals, functions, objects, variables scopes, prototype inheritance, etc. Functions in JavaScript are first-class because they can be assigned as values to variables and objects, they can be passed as parameters to function calls and returned as values of functions executions. Closures are inner-functions (a function defined inside another function called the parent), which in particular can make use of variables defined in the scope of the parent.

JavaScript dynamic features

JavaScript has many dynamic features. Unlike object-oriented languages such as Java where objects are created out of classes using the `new` construct, objects in JavaScript can be defined in many different other ways: object expressions, definitions via the `Object` object. Functions can also be dynamically created with the `Function` object. Inheritance a-la-JavaScript is a prototype-based inheritance. Any object can be the prototype of any other object, and objects may have no prototype. When a property is not directly defined in an object, the prototype chain is traversed in order to look it up.
One of the most dynamic features of JavaScript is undoubtedly the `eval` function. It takes as an argument a string, turns it into a code and executes it. Likewise, functions such as `setTimeout, setInterval, Function` also dynamically turn strings into code. We refer to them as `eval`-like functions. They are among the most interesting but also the most challenging features of JavaScript, when it comes to analyzing JavaScript programs [210, 240]. These functions pose a lot of security issues in JavaScript applications, as their use can allow an attacker to execute arbitrary code in web applications.

The web programming language

JavaScript is the defacto programming language of the web, implemented by browsers and extensively used in websites [153, 229] to make HTML documents interactive. The `script` tag serves to include scripts (JavaScript programs) in a webpage. JavaScript is also used by non-browser environment such Node.js [105]. JavaScript programs manipulate webpages thanks to the DOM API of the webpage that browsers expose to them. They can register event listeners or callbacks (functions) that are invoked in reaction to events occurring in the webpage (a mouse move, a user click on an element, a network event). The DOM is accessible via the `document` object, which they can use to query for HTML elements in the page, add, remove or change elements and their attributes. In the listing below, the text of the first paragraph in the a webpage is changed to *New paragraph text*.

```
document.getElementsByTagName("p")[0].innerText = "New paragraph
    text"
```

The `document` object is in reality a property of the global object `window`, whose properties contain different information about a webpage and the browser in which it is rendered. Different aliases are sometimes used to refer to the `window` object: `this, self, global` [3].

1.6 Browsing contexts

A browsing context is an execution environment where browsers load and render a web document [19]. A browser tab or window corresponds to a browsing context. Tabs and windows represent the User Interface (UI) of a webpage. Associated to a browsing context are a set of resources and a common memory. In particular, scripts running in the context of a page have access to all objects and data related to the context.
A webpage can be embedded into another webpage, using an HTML `iframe` (as shown in Listing 2.2) or `frame` tag. The embedded document will be placed inside a different browsing context, called a nested browsing context. The embedding document is called the parent or top-level context, and the embedded document the child or nested context. A browsing context that has no parent is called a top-level browsing context, others are nested browsing contexts. An iframe can further embed another iframe and so forth, causing different levels of nested browsing contexts [19]. Nested browsing contexts are rendered in the UI of their parent context.

2 The Same Origin Policy

The Same Origin Policy (SOP) is a fundamental security mechanism implemented by web browsers [125]. Among other things, it defines (i) the ability to include third party content in webpages, (ii) the ability for a webpage to interact with third party servers in order to load data, and (iii) the interactions between browsing contexts.

2.1 Origin

Browsers associate web applications and their resources (pages, files, content, data) to a single origin. Resources that have the same origin are called same-origin resources, otherwise they are considered cross-origin resources. An origin is usually made of 3 components of a URL [144]: the scheme or protocol, the host or domain name and the port number.
Let us consider the URL `https://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash`. The host or domain name (`sub.host.com`) is the name of the machine (server) hosting the resource [4]. A domain name is further divided into different parts or levels, with the TLD (top-level domain) being the final component of the domain name. In our example, `com` is the the top-level domain (TLD), `host.com` the second level domain (TLD+1). Third and higher level domains are called sub domains (i.e. `sub.host.com`). Cookies for instance are organized by hosts in browsers (See Section 1.2). The scheme is the communication protocol used by browsers (clients) and servers to exchange data. In our example, `https` is the scheme of the URL. Different communication protocols are associated to different schemes (`http` for the HTTP protocol, `https` for the HTTPS protocol, `ftp` for the FTP protocol, ...). Finally, the port is a unique number that identifies a network-based application (an application that uses the network to communicate with different entities) running on a machine. In our case, the network-based application is the web server which communicates with web browsers. By default, different protocols are associated to specific port

3. This depends on the context: `this, self` are exactly the global `window` object if used outside of any function, that is, the global scope [80]
4. This is later converted to an IP address

numbers. The HTTP protocol is associated to the port 80 and the HTTPS protocol to the port 430. Hence, a URL whose protocol is `http` but does not include a port number is implicitly associated to the port number 80. URLs also contain additional information such as the path to the resource being accessed, query strings or URL parameters which are additional data passed to the URL, user credentials (user name and password), etc. A web application or website is typically a collection of web resources which has a common domain name.

Remote and inline content

Schemes can be grouped in two main categories: network and local schemes [54]. Network schemes include HTTP(S) schemes (`http, https`), `ftp, ws, wss` and local schemes include `about, blob, data, file, filesystem, javascript`, among others. A resource which is embedded in a webpage with a network-scheme URL, requires that the browser connects to the server hosting the resource (using the communication protocol corresponding to its scheme), in order to fetch the content of the resource. Local-scheme URLs or URIs (Uniform Resource Identifiers) [143] on the other hand, are URLs which do not trigger a network communication. The following listing shows the inclusion of a `data` URI image in a webpage [45]

```
<img src="
ANSUhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4
//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU
5ErkJggg==" alt="Red dot" />
```

The content of the image is wholly indicated in the URL. The browser just has to decode and render it, and not make a connection to a remote server. We refer to content with network-scheme URLs as remote or external content. In the case of JavaScript, we refer to them as external libraries [153, 229]. Otherwise, we refer to them as inline content, when the URL scheme is a local-scheme.
It is worth noting the case of inline scripts, stylesheets and iframes, which can also be declared without any URL. The `<style>` tag is used for declaring inline stylesheets. An inline script is a script whose `<script>` tag does not have any `src` attribute. Its content (code) is directly indicated between the start and end tag. Inline iframes can also be created using JavaScript `document.createElement, document.write` APIs [10].
Also important is the `srcdoc` attribute introduced in HTML5 [203] for creating inline iframes. The content of the iframe is directly indicated in the `srcdoc` attribute. Note that the `srcdoc` attribute takes precedence on the `src` attribute, when both are specified on an `<iframe>` element. Browsers supporting both attributes will ignore `src` in presence of `srcdoc` [203].

2.2 Cross-origin embeddings

The Same Origin Policy defines rules regarding the inclusion (embedding) of cross-origin resources in a webpage [125]. A cross-origin or third party resource is a resource whose origin is different from that of the page in which it is embedded (say, a resource included in a page using an HTML tag). If a page whose URL is `http://example.com` embeds a script with the URL `http://third.com/script.js`, then the script is considered a third party or cross-origin script. The page itself is usually referred to as the first party.
The SOP allows the inclusion of third party content in a webpage. This has many security implications.

— Once loaded, a third party script will execute with the same privileges as first party scripts. In other words, it can access and manipulate any data, object, the DOM and any API exposed to the page, as if it was loaded from the page's own origin.

— Cookies sent by the third party server in response to the request to fetch (load) the third party resource, will be associated to the third party domain, and not to the first party domain. That notwithstanding, third party scripts in particular can access the first party cookies via JavaScript APIs such as `document.cookie` [49].

When the embedded resource is a cross-origin iframe, then the iframe is loaded inside a cross-origin browsing context.

Interactions between browsing contexts

The Same Origin Policy also governs the interactions between different (nested) browsing contexts. It allows a script in a browsing context to directly access other same-origin contexts and their related data, DOMs, etc. For instance, a script in a page can access and manipulate the DOM of an iframe from the same origin and vice versa. Similarly, two same-origin iframes, embedded in a page, can directly access each other's context.
Cross-origin interactions however are disallowed by the SOP. Nonetheless, it is worth mentioning the case of sub domains. When 2 cross-origins contexts differ only in their full domain names, while having the same second-level (`TLD+2`) domain component, they can relax their origins in order to enable direct interactions with each other's browsing context. Assume that the origins of the 2 cross-origins contexts are `http://sub.host.com` and `http://www.host.com`. Since their common `TLD+2` component is `host.com` then both pages can execute

```
document.domain = "host.com"
```

in order to become same-origin contexts, and directly access each other's context. Now both contexts are associated to the same origin `http://host.com`, allowing them to interact. This is referred to as relaxing an origin. Note that even if the origin of one of the context was already `http://host.com`, the context must also explicitly relax its origin to be able to interact with sub domains which have relaxed their origins into `http://host.com`.

Cross-origin communications

If cross-origin browsing contexts do not have direct access to one another's browsing context, they can however communicate by exchanging messages. This also applies to same-origin contexts. Message exchanges or message passing are achieved with the cross-origin communication `postMessage`API [116]. Below is a listing showing how to use this API, where `message` is the message or data to send, and `origin` is the origin of the contexts the message is to be sent to.

```
postMessage(message, origin)
```

Messages are dispatched on a per-origin basis. That is, all contexts whose origin matches that of the `origin` parameter (the second parameter) will receive the message. To send the message to all contexts, one can specify `*` as value for the `origin` parameter.
To receive messages, browsing contexts have to register listeners for events triggered by incoming messages, as shown in the listing below.

```
addEventListener("message", function(event){
  message = event.data;
  origin  = event.origin
});
```

The messages sent cause the triggering of a `message` event in the destination contexts. The message sent and the origin of the context that sent it, are accessible from the `event` object, as shown in the listing above.

Sandboxing iframes

The HTML5 specification [70] introduced the `sandbox` attribute to be used with the `<iframe>` tag.

```
<iframe src="http://example.com/iframe.htm" sandbox></iframe>
```

Listing 2.4 – Sandboxing an iframe

The `sandbox` attribute, as shown in Listing 2.4, applies many restrictions on the nested browsing context of the iframe, and has among other things, the effect of altering the origin of the iframe context. Instead of `http://example.com` as an origin, as one may expect, the iframe now has a different origin called a `unique` origin. The main property of a unique origin is that it does not match any other origin, not even another unique origin. Hence, a unique origin is considered a cross-origin compared to any other origin. The SOP therefore disallows access from an origin to a unique origin context and vice versa. Cross-origin communications can nonetheless be set up with unique origins.
The `sandbox` attribute also prevents the iframe from loading plugins, scripts, submitting forms, displaying popups, navigating the top-level context, etc. Most of these default restrictions can be relaxed by adding specific values (flags) to the attribute.

— The value `allow-same-origin` of the `sandbox` attribute gives back to the sandboxed context, its expected origin. The iframe of Listing 2.4 would no longer be assigned a unique origin, but rather `http://example.com` as expected.

— The value `allow-scripts` renables scripts execution inside the sandboxed context.

— The value `allow-forms` renables forms submissions.

It is worth mentioning the fact that, any context further nested inside a sandboxed iframe, inherits the sandbox restrictions of its top-level contexts, in addition to the sandbox restrictions of the nested context itself. For instance, if scripts execution is not allowed in an iframe, scripts execution is also not allowed in any context nested within the sandboxed iframe.

2.3 Cross-origin reads

This refers to the ability of loading cross-origin data in the context of a web page. AJAX (Asynchronous JavaScript + XML) [12] involves the use of various technologies that enable scripts executing in a webpage, to connect to web servers in order to submit and fetch data and update the user interface, without having to reload the page. In comparison, the use of HTML forms [71] for sending and receiving information have the inconvenience of causing webpages to reload.
AJAX requests are made using JavaScript objects such as `XMLHttpRequest` [155] or `fetch` [54]. Listing 2.5 shows a simple AJAX request made using the `fetch` API.

```
fetch("http://host.com/data");
```

Listing 2.5 – Simple AJAX request using the `fetch` API

By default, the SOP does not allow cross-origin AJAX requests. Hence, a webpage from an origin cannot use AJAX to fetch data on a web server of a different origin (cross-origin).

JSONP

One of the most used techniques to circumvent cross-origin reads is JSONP (JavaScript Object Notation with Paddding) [5] This hack relies on the fact the Same Origin Policy does not prevent the inclusion of third party (cross-origin) scripts (See Section 2.2). To load third party data with JSONP, one injects in the page a script whose URL parameters include the name of a JavaScript `function` defined in the context of the page. This function specifies how the data returned by the web server will be handled. The listing below shows the inclusion of a script to load JSONP data. The URL is passed the parameter `callback` whose value `foo` is the JavaScript function to be called to handle the data returned by the server.

```
<script src="http://third.com/script.js?callback=foo"
```

The third party server then generates a response which contains the name of the function to which it passes the data as an argument, as shown in the following listing.

```
foo(cross-origin-data);
```

When the response is returned to the browser, the function `foo` will be invoked and the data treated according to the definition of the function. The main limitation of JSONP lies in the possibilities that it offers. Only the HTTP `GET` method is used for making requests. Hence, data can only be submitted as parameters to the URLs. Compared to AJAX requests, JSONP has way less possibilities. To enable cross-origin requests, the Cross-Origin Resource Sharing (SOP) mechanism has been introduced.

3 Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) [34] is a refinement of the Same Origin Policy. It involves the exchange of dedicated HTTP headers between web browsers and web servers, in order for the latter to authorize (or not) cross-origin AJAX requests. Before CORS, the SOP mandated that browsers block cross-origin requests. With CORS, the control is given to web servers. In particular, the browser indicates to the target server, the origin of the webpage making the cross-origin request. The server can then decide whether it accepts requests from that origin (cross-origin requests). It is important to note here that web servers have full control over accepting or rejecting CORS requests.
To work, CORS must be implemented both by the browsers and the web servers. However, CORS is fully backwards compatible. Hence, if either browsers or web servers do not implementing the mechanism, then browsers will fallback to the traditional SOP, by blocking cross-origin requests [6].

3.1 Types of CORS requests: simple and preflighted

CORS is defined in the Fetch specification [55] and many online resources discuss how to deploy and use it [33,127,148]. There are 2 types of CORS requests: simple and preflighted

5. The technique is named JSONP because it was mostly meant for loading data with the JSON format. However, it can be used to load other types of data that can be handled by a JavaScript program such as texts, numbers, XML, etc.

6. If the browser supports CORS, it will always attempt to make cross-origin requests. If web servers respond with some CORS headers, it will enforce them, otherwise, it falls back to SOP. This may seem inefficient in case the web server does not support CORS at all, because there is a round-trip request from the browser to the server. This is because there is no HTTP headers for a web server to express that it does not accept cross-origin requests for instance

requests. Preflighted requests require 2 requests in order to be fulfilled: first, the browser makes a `preflight` request, then in a second time, makes the effective CORS request.
Simple CORS requests are those that basically have the same characteristics as HTTP requests already possible with HTML forms [54]. Hence, in simple requests:

— only 3 HTTP methods are allowed: `GET`, `POST` and `HEAD`

— only a predefined set of HTTP headers can be used (`Accept`, `Content-Language`, ...). The type of data that can be transmitted in simple CORS requests, using the `Content-Type` request header, can only be `application/x-www-form-urlencoded`, `multipart/form-data`, and `text/plain` [7].

As an analogy, simple CORS requests can be understood as a way of submitting and fetching HTML forms data without reloading a webpage.
As mentioned above, requests that are more elaborated than simple CORS requests are preflighted CORS requests. This is the case when in the AJAX request, one uses an HTTP method other than `GET`, `POST` and `HEAD`, or uses custom HTTP headers other than those allowed in simple requests [8]
In order to fulfill preflighted requests, browsers actually make 2 sequential requests. A first `preflight` request, made using the `OPTIONS` HTTP method, is sent to the cross-origin server, to notify it about the characteristics of the preflighted requests that the webpage is willing to make (i.e. the webpage wants to communicate using an HTTP method other than `GET`, `POST`, or `HEAD` or it wants to send some custom HTTP headers). When the server authorizes the preflight request (by responding with dedicated HTTP headers), then the browser will send the effective CORS request, using the specific method and custom HTTP headers.
Finally, by default, CORS requests are made without including the credentials (cookies and authorization tokens) that may have been previously set by the server in the user browser. However, webpages can instruct that CORS requests be made with credentials. In this case, the browser will add the cross-origin server credentials to the CORS request. Nonetheless, the server has here again full control and can accept or deny the request with credentials. Note that requests with credentials are sensitive. If authorized by the server, they allow a webpage from an origin to access potential user data on cross-origin web applications.

3.2 CORS headers

Table 2.2 presents the different headers used for making CORS requests [40, 55]. Intuitively, there is a one-to-one correspondence between each CORS request header sent by the browser, and the dual used by the web server to respond in order to authorize or not the cross-origin request.

Origin of the request First of all, the `Origin` request header is always added to cross-origin requests. It tells the web server about the origin of the webpage from which the cross-origin request is being made.

```
fetch("http://third.com");
```

Listing 2.6 – Simple CORS request without credentials, made using the `fetch` object

7. These are the possible data types for HTML forms
8. Using ReadableStream object or registering an event listener on an XMLHttpRequestUpload object also triggers preflighted requests [33]

Request headers	Response headers
Origin	Access-Control-Allow-Origin
Access-Control-Request-Method	Access-Control-Allow-Methods
Access-Control-Request-Header	Access-Control-Allow-Headers
-	Access-Control-Expose-Headers
-	Access-Control-Mage-Age
Cookie, Authorization	Access-Control-Allow-Credentials

Table 2.2 – CORS headers exchanges between web browsers and servers. In many cases, there is a one-to-one correspondence between the requests and responses headers. The browser sends a header, and the server uses its dual to authorize or reject cross-origin requests

Consider the request in Listing 2.6. If it is made by a webpage whose URL is http://example.com:8080/home.htm, then the browser will add to the cross-origin request, the header `Origin` and set its value to http://example.com:8080, which is the origin of the webpage. The target web server (http://third.com) then knows the origin of the request. If it accepts requests from pages with this origin, it can express this to the browser by using the `Access-Control-Allow-Origin` response header. It has two possibilities as values for the `Access-Control-Allow-Origin` headers: either `*` or the same value as that of the `Origin` header (that is http://example.com:8080 in our example). In the second case, the values of the `Access-Control-Allow-Origin` and the `Origin` headers match because they have the same value. For the first case, `*` matches any origin. In either cases, the cross-origin request is authorized by the server. Any other value returned by the server for the `Access-Control-Allow-Origin` header, or the absence of this header would have been interpreted by the browser as an unauthorized cross-origin request.

Custom HTTP headers and methods Consider the CORS request in Listing 2.7, that makes use of the method `PUT`. This request is a preflighted request because of the method `PUT`, which is not allowed in simple CORS requests.

```
fetch("http://third.com", {
  method: "PUT"
});
```

Listing 2.7 – Preflighted request, because of the use of `PUT` method in the request

Therefore, a preflight request is first made. In addition to the `Origin` header, the browser also adds the `Access-Control-Request-Method` header to the request. The value of this header is the the `PUT` method, which tells the server about the method the client is willing to use to make the cross-origin request. To authorize the use of this method, in addition to the `Access-Control-Allow-Origin` header, the server adds the `Access-Control-Allow-Methods` header in its response. The value of this header must be a comma-separated list of allowed HTTP methods that include `PUT` as a value[9].
Similarly, the `Access-Control-Request-Header` is used in a preflight request to indicate to the web server that the client would like to make a request with custom HTTP headers. Listing 2.8 shows an example of a request that will be preflighted because the value `application/json` of the `Content-Type` header is not allowed in simple CORS requests.

```
fetch("http://third.com", {
```

9. The value of the header can be also be only the `PUT` method

```
    method: "POST",
    headers: {
      "Content-Type": "application/json"
    },
    body: "..."
  })
```

Listing 2.8 – Preflighted request, because `application/json` as a value for the `Content-Type` header is not allowed in simple CORS requests.

In this case also, the browser makes a first preflight request. In addition to the `Origin` header, it also adds the `Access-Control-Request-Header` header, with `Content-Type` as its value. In case of multiple custom headers, the browser adds all of them as values to to the `Access-Control-Request-Header` header, separating them with a comma. To authorize requests with the custom headers, the server sends the `Access-Control-Allow-Headers` header, whose value is a comma-separated list of headers, containing the headers sent in the `Access-Control-Request-Header`. A custom header is allowed if it is listed in the `Access-Control-Allow-Headers` header values.
Additional response headers can be added by the server in the responses to preflight requests. The value of the `Access-Control-Expose-Headers` header is a list of response headers that the server authorizes to be made visible by the browser to the webpage which issued the request. The value of the `Access-Control-Mage-Age` header indicates a number of seconds during which the response to the preflight request can be cached by the browser. Till this delay expires, the browser is allowed to omit the first preflight request, for similar preflighted requests to the same server.
After a successful preflight request, the browser makes the effective CORS request, with the authorized methods and headers.

Requests with credentials By default, CORS requests are made without adding the credentials of the cross-origin server to the requests. However, the webpage issuing the request can explicitly instruct the browser to include the server's credentials, as shown in Listing 2.9.

```
fetch("http://third.com", {
  credentials: "include"
});
```

Listing 2.9 – Making a CORS request with credentials, using the `fetch` API

In this case, the browser will add any credentials (cookies, authorization keys) previously set by the web server in the user browser. To accept the request with credentials, the web server returns the `Access-Control-Allow-Credentials` response header, setting its value to `true`. It must also explicitly set the origin of the webpage as the value of the `Access-Control-Allow-Origin` response header, and not use the `*` wildcard [10]. Otherwise, the request fails.

3.3 CORS and sandboxing

It is worth mentioning the case of CORS requests made from sandboxed contexts (See Section 2.2). In such contexts with unique origins, the value of the `Origin` header is set to `null` by the browser in cross-origin requests. This makes it impossible for a web server (by considering only the value of this `Origin` header) to know the real origin of the sandboxed

10. `*` matches any origin in the case of requests without credentials, but not requests with credentials

page from which a cross-origin request is being made. Hence, special care is to be taken by the server, when considering cross-origin requests from unique origins. As the request may be coming from any sandboxed context, one should not rely on the `Origin` header to allow access to user data. For instance, one may not accept CORS with credentials from unique origins without further information, such as the use of an additional token for example.

4 Security and privacy threats of third party content in web applications

Thanks to powerful scripting capabilities rendered possible by JavaScript, webpages can be made highly interactive and dynamic. Today's web applications have nothing to envy of traditional desktop applications in terms of features and performance, due to the constant improvements in browsers and web technologies. Web applications collect, store and manage data provided by users. For instance, many web applications offer users the possibility to provide their profile information (name, birth date, addresses, photos, credit card numbers...), which are stored by web servers, and used to later give access to different services provided by the application. Web applications are also collaborative. For instance, a news website can offer users the possibility to comment on articles and share their views with other users.
The time is now long gone, when webpages were made of content originating solely from the page's own origin. Today, webpages embed various content from third parties. Such pages are usually referred to as mashups, as they are built using content from different third parties.
Attacks are widespread in the web [135], and web applications may contain vulnerabilities that malicious users (attackers) can exploit to inject and execute malicious content (scripts) in web applications. Third party content providers may also be malicious, or they can get compromised by attackers, resulting in the injection and execution of malicious third party content in web applications.
In this section, we describe security and privacy threats, both related to the presence of third party content in web applications.

4.1 Cross-Site Scripting (XSS)

An XSS attack is a content (script) injection attack [41] in web applications that usually involves a vulnerable website, the attacker or malicious user, and the victim user. Intuitively, the attack consists in injecting some malicious code in a vulnerable web page of a website, so that it is executed in the browser of the victim user. Once injected, the malicious code is indistinguishable from legitimate (benign) code executing in the context of the page, as we have mentioned in Section 2.2. The malicious code gains the same power as any code in the context of the page, allowing the attacker to steal user information such as cookies, mount phishing attacks [112] (by making HTML forms that contain user credentials point to a server under the control of the attacker), record sensitive information provided by the user, etc. XSS is regularly ranked among the top 10 vulnerabilities in web applications [135].

Types of XSS attacks

Three types of XSS attacks are usually admitted depending on the way they are conducted [3]: persistent or stored, reflected and DOM-based XSS. DOM-based XSS is a

variant of both persistent and reflect XSS. In persistent and reflected XSS, the vulnerability is in the server's response, which fails at correctly sanitizing the inputs it receives from the attacker, but outputs them as part of the response. In a DOM-based XSS however, the vulnerability lies on the client-side code (JavaScript) manipulating the attacker-controlled data without proper sanitization, leading to the execution of the attacker content as a code [3].

4.2 Third party web tracking

Third party web tracking is the ability of a third party to recognize a user as she browses the web and record her browsing history [225]. The more the websites which embed content from a third party, the more precise is the browsing profile that the third party can build about users visiting these websites. The main implication of such practices is a violation of user privacy. In fact, the browsing profile can reveal sensitive information about a user: her habits, political opinions, religious beliefs, sexual orientation, etc. Studies have shown that third party tracking is often done with the purpose of web analytics, targeted advertisement or other forms of personalization, and even user surveilliance [189].
Different techniques of third party tracking (or more precisely user recognition techniques) have been demonstrated in the wild [188, 217, 225, 242]. Mayer and Mitchell [225] grouped them in two categories called `stateful` (cookie-based and super-cookies) and `stateless` (fingerprinting) mechanisms. We illustrate a cookie-based tracking mechanism, and browser fingerprinting below.

Cookie-based tracking

If a third party has its content included in many different webpages, then, each time a user visits one of these pages, the request made to load the content tells the third party about the webpage in which the content is embedded. Usually, browsers add the `Referer` header to requests, setting its value to the URL of the page in which the content is embedded. Therefore, to enable (cookie-based stateful) tracking, the third party can set a cookie in the user's browser, the first time a request is made to load content. Then, as browsers will attach the cookie to subsequent requests to the same third party, (See Section 1.2), the latter can combine the URL of the page and the cookie, to recognize the user and build her browsing profile (all the pages the user has ever visited).

Browser fingerprinting

Browser or device fingerprinting is another tracking mechanism that was first demonstrated by Eckersley [185], where the user identifier is her browser and its properties. In this scenario, when the user first visits a website that embeds some content from a tracker, the tracker does not store any stateful information (i.e., a cookie) in the user's browser. It rather collects different properties of the user's browser, including for instance the user agent, i.e. `Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0` (which contains information such as the name, type, version, ... of the browser and the OS on which it runs), the list of plugins, fonts, browser extensions installed in the user's browser, or the set of websites she is logged into. It then combines all these information to build a unique fingerprint of the user browser. This fingerprint is stored on the tracker's server. Later on, when the user visits another website that also embeds some content from the tracker, the tracker can recognize the user by once again collecting the properties of the user browser, and comparing the current fingerprint with the set of user fingerprints it has

already collected. In case of a match, the tracker successfully recognizes the user behind the browser. It is rather intuitive to convince ourselves about the effectiveness of a stateful tracking, since it is based on unique identifiers that are set in users' browsers. Many users however, can have browsers with similar properties, and thus similar fingerprints as well. Nonetheless, the efficacy of stateless mechanisms has been extensively demonstrated. Since the pioneer work of Eckersley [185], new fingerprinting methods have been revealed in the literature [163–165, 173, 180, 188, 230, 259, 263], able of uniquely identifying users with high accuracy. In our study on browser extensions and Web logins fingerprinting (See Chapter 7), we found that around 90% of Chrome browser users who have installed at least one extension and are logged into at least one website, are uniquely identifiable.

5 Content Security Policy

The Content Security Policy (CSP) [258] is a W3C (World Wide Web Consortium) mechanism that allows programmers to control which resources can be loaded in a webpage. In fact, the SOP allows the embedding of content from different origins, including from third parties (See Section 2.2). CSP can be understood as a refinement of the Same Origin Policy, for further restricting the origins from which content can be loaded within a webpage. Using CSP, a developer can explicitly whitelist the origins of content that are trusted and can thus be loaded in webpages. Hence, once a webpage is rendered, browsers would block any content not whitelisted in the CSP of the page. Such content may have been injected by an attacker who exploited a vulnerability in the application. CSP is useful as a defense-in-depth mechanism for mitigating the impact of content injection attacks in general and Cross-Site Scripting (XSS) in particular [41]. As a first line of defense, developers are invited to properly sanitize content that they receive from users, before including them in web pages either from the server side or from the client-side. Thereafter only, can one deploy a CSP to further guard against the impact of these attacks.
Following the success of CSP version 1 (CSP1 [261]), the W3C has standardized the second version of CSP (CSP2 [275]) and the next version (CSP3 [272]) is already in an advanced development state. Many browsers implement CSP, and more and more websites deploy it to protect their pages against attacks [162, 177, 255, 267, 269].

5.1 Directives

In order to whitelist content of different types, developers express CSP policies using directives to choose a type of content, and directive values to choose trusted origins [261, 272, 275]. Table 2.3 shows the meaning of CSP directives. The list of CSP directives presented here is not exhaustive, but comprises the majority of them. We have chosen to focus mostly on directives used for restricting content inclusion in webpages.
Each directive targets a specific type of content, and can thus be used to restrict the origins from which content of the particular type can load from. Directive `script-src` is the most used feature of CSP in today's web applications [267]. It specifies trusted origins where scripts can be loaded from. The `default-src` is a directive used as a fallback for `*-src` directives (directives which names end with `-src`). When `default-src` is present in a policy, and any of the directives which fallback to it is not specified, then, the missing directive implicitly inherit the restrictions (values) of `default-src`. The directive helps for instance to apply the same restrictions on many directives at a time, without explicitly specifying them. The `sandbox` directive is used to sandbox the webpage to which it applies, as it the page was a sandboxed iframe (See Section 2.2).

Directive	Description
script-src	scripts execution with <script> tag, XSLT stylesheets, JavaScript events listeners (onload, onclick) on HTML elements
object-src	embedded plugins (Flash, PDF, etc) with object, embed, applet tags
plugin-types	allowed types of plugins (application/pdf, application/x-shockwave-flash) specified by the type attribute of object, embed, applet tags
style-src	stylesheets (CSS) embedded with link, style tags
font-src	fonts loaded via @font-face property of stylesheets and FontFace JavaScript API
img-src	images inclusion with img tag, poster of video tag
connect-src	applies to AJAX requests (XMLHttpRequest, Fetch), WebSocket, EventSource
media-src	applies to audio, video tags
frame-src, child-src	applies to frames loaded with iframe, frame tags
frame-ancestors	origins of top-level contexts which can nest the page as an iframe
form-action	applies to the values of the action attribute of HTML form elements
sandbox	sandbox a page and its resources. This is similar to the use of the sandbox attribute on a iframe (See Section 2.2)
report-uri, report-to	endpoints where to submit CSP violations reports (content in the page not matching the policy)
default-src	used as a fallback directive for any type of content, which directive is not explicitly added to the policy. It applies to *-src like directives

Table 2.3 – Excerpt of CSP directives and their descriptions

5.2 Directive values

Directive values or source lists are basically the trusted origins from which content can be loaded from. The CSP specification [261, 272, 275] defines for each directive, the possible values that it can have. We give here an overview of possible values commonly used in directives.

Host expressions

Host expressions are used to whitelist content based on their origins. Hosts can either be full origins (i.e. https://example.com:8080), origins with paths (https://example.com:8080/path/), used to whitelist a set of content, or even a specific content (i.e. https://example.com/script.js). The scheme part of the origin can be omitted (in which case it will be assigned the scheme of the page in which the policy is enforced). The domain name of the origin can contain wildcards (i.e. *.example.com) to include the sub domains of a higher-level domain. Finally, the port can also be omitted (in which case it corresponds to the default port of the origin scheme specified, or otherwise the port of the webpage). The host and port can take the special value *, in which case they match URLs with any host and port number. The * can appear as a standalone value in a directive. In this case,

it allows content from any origin (excluding some local-scheme URIs content, depending on CSP version). Finally, `'none'` is a special directive value to express that no content are allowed to load.
The semantics of the host expressions values has evolved between CSP3 and previous versions [272] w.r.t insecure schemes and ports. In CSP1 and CSP2, insecure (HTTP) origins allow content only from the exact origin. In CSP3, this origin also allows content from its secure (HTTPS) counterpart. For instance, whitelisting `http://example.com` implicitly also whitelists `https://example.com` in CSP3.

Schemes

They are used to whitelist content with a specific scheme. For instance, the value `https:` in a policy matches any URL whose scheme is `https` (i.e `https://example.com/content`). Common schemes include `https:`, `http:`, `data:`, `blob:`, `wss:`, `ws:`, `filesystem:` and `mediastream:` (the colon is required when specifying schemes in CSPs).

Keywords

CSP defines many keywords with different semantics. The `'self'` keyword is used to whitelist content from a page's own origin. One can also directly include the full origin of the page in the policy to whitelist it.
Inline scripts and DOM event handlers represent the main vector for content injection attacks. By default, CSP bans the inclusion of inline scripts and DOM event handlers in webpages. However, the `'unsafe-inline'` keyword can be used to allow the inclusion of inline scripts and stylesheets. Enabling inline scripts in particular makes a CSP ineffective against attacks [177, 214, 267].
JavaScript has many functions that can be used for turning strings into code. This includes `eval, setInterval, setTimetout, Function` (See Section 1.5). Such functions are banned by default from policies. The specification however defines the `'unsafe-inline'` keyword (to be used with `script-src` and `style-src` directives), in order to renable the use of these functions.
One of the reasons that hindered a wider adoption of CSP was the lack of a mechanism for easily accommodating dynamically injected scripts. Weichselbaum et al. [267] then proposed the `'strict-dynamic'`, a keyword to be used to allow scripts whitelisted with nonces or hashes, to further dynamically inject additional scripts even if they are not explicitly whitelisted in a policy. In anterior CSP versions, for a dynamic script to be allowed to load, it must have matched the policy of the page. In CSP3, if the dynamic script is loaded from a script which is already loaded (because it matches the policy), then the dynamic script is allowed to load, even though it does not match the policy. When the `'strict-dynamic'` keyword is used in a policy, we refer to the overall policy as a `strict CSP`.
Directives which do not rely on hosts and schemes have dedicated keywords. For instance, the values of the `sandbox` directive are those defined by the HTML specification for the `sandbox` attribute of iframes and includes among others, `allow-scripts`, `allow-same-origin` and `allow-forms` [76] (See Section 2.2). The values set to the directive `plugin-types` are MIME types [96] of content and include for instance `application/pdf` (for PDF documents), `application/x-shockwave-flash` (for Adobe Flash plugins), `text/html` (for HTML documents), `text/css` (for CSS files), `image/png, image/jpeg, image/gif` (for images of type PNG, JPEG, and GIF respectively), etc.

Nonces and hashes

Nonces and hashes have been introduced starting from CSP2 mainly to allow the whitelisting of individual inline scripts and stylesheets. So far, there is no way to whitelist individual DOM event handlers using nonces, or hashes [11]. The use of nonces improves on the effectiveness of CSPs regarding inline scripts [272, 275] by allowing browsers to distinguish between trusted scripts and attacker code. Comparatively, the use of `'unsafe-inline'` in a policy automatically makes CSP ineffective against attacks. [177, 214, 267].
A nonce is a token, randomly generated using strong cryptographic routines in order to prevent an attacker from guessing the nonce in advance [275]. To whitelist a script with a nonce, the nonce is added to the `script-src` directive, and also as a value of the `nonce` attribute of the corresponding `<script>` tag in the page. A single nonce can be assigned to multiple scripts.
To whitelist a script or stylesheet with a hash, one computes the hash of the script of stylesheet content using Secure Hash Algorithms (SHA) [126]. Then, the hash is encoded in base64 and added to the `script-src` or `style-src` directive. Hence, when the browser encounters a script or stylesheet in the page, it will compute its hash, and compare it with the whitelisted ones in the policy. If there is a match, the script or stylesheet is allowed to execute, otherwise it is blocked.

Precedence of directives values

It is worth mentioning the case of policies that combine `'strict-dynamic'`, with nonces/hashes and the `'unsafe-inline'` keyword. Consider the policy shown in Listing 2.10.

```
script-src 'strict-dynamic' 'nonce-random123' trusted.com https:
    'unsafe-inline'
```

Listing 2.10 – CSP differently enforced depending on the version

— In CSP3, only `'strict-dynamic'`, the nonces and hashes are considered. We refer to polices with nonces and `'strict-dynamic'` as strict CSPs. In strict CSP, the hosts, schemes, `'self'` and `'unsafe-inline'` keywords are ignored. So a script is allowed to load if it has a valid nonce or hash, or if it is dynamically injected by a script which has already loaded (thanks to `'strict-dynamic'`).

— In CSP2, `'strict-dynamic'` is ignored. If nonces or hashes are declared, `'unsafe-inline'` is also ignored. Other values are enforced, including nonces, hashes, hosts, schemes, and other keywords.

— Finally, in CSP1, nonces, hashes and `'strict-dynamic'` are discarded. Hosts, schemes, others keywords such as `'unsafe-inline'` are enforced.

5.3 CSP modes and headers

A CSP to be enforced on a webpage is deployed either as an HTTP response header, or in the HTML response body of the webpage using a `<meta>` tag. The name of the CSP header depends on the mode of deployment. There are 2 modes. In the report-only mode, the browser does not prevent a content that do not match the policy from loading. These content are simply reported to the developer at the reporting endpoint specified in the policy with the `report-uri` or `report-to` directives (See Table 2.3). To deploy a CSP in report-only mode, one uses the `Content-Security-Policy-Report-Only` header.

11. There is draft proposal in the current CSP3 [272], to be able to use hashes for whitelisting individual DOM event handlers. This mechanism is however not yet implemented by browsers

In the dual enforcement mode, content that do not match the policy are effectively blocked, before being reported. The `Content-Security-Policy` can be used to deploy a policy in enforcement mode [12].
The `<meta>` has some restrictions. Report-only policies are not enforced when delivered in the `<meta>` tag. Moreover, even when delivered in enforcement mode, the `sandbox` and `frame-ancestors` directives are ignored by the browser when a policy is delivered in the `<meta>` tag.
CSP allows to deploy and enforce many policies on the same page. One can either deploy multiple policies in a single header by separating them with a comma, or send multiple headers (report-only or enforcement mode), each with a single or set of policies. In any case, multiple policies are all individually enforced. A content is allowed to load if it is allowed by all of the policies.

5.4 Example of CSPs

We present here different examples of policies.

Origin-based policies

Listing 2.11 presents a CSP where scripts are whitelisted based on their origins. We call them origin-based policies.

```
script-src trusted.com redirect.com partials.com/scripts/;
img-src https:
```

Listing 2.11 – Example of an origin-based CSP

Only scripts from the explicitly specified origins are allowed to load in the webpage on which this policy will be deployed. Assume that this policy is deployed on a page with the URL `https://example.com`, then the injection of a script with URL `https://trusted.com/script.js` in the webpage is allowed since the script comes from the whitelisted origin `trusted.com`. Images can be loaded from any secure domain. No restrictions are set on other types of content.

Nonce-based policies

Listing 2.12 shows a nonce-based policy.

```
script-src 'nonce-random12345' 'strict-dynamic';
```

Listing 2.12 – Example of CSP with nonces

Nonces are used to whitelist individual scripts. To allow a script to load, one injects `<script src="https://trusted.com/script.js" nonce="random12345"></script>` in the page (Note the use of the `nonce` attribute which value is a nonce whitelisted in the policy of the page).
With the presence of the `'strict-dynamic'` keyword, scripts that load can further dynamically inject additional non-parser-inserted scripts. Listing 2.13 shows an example of a parser inserted script, that will fail to load in presence of `'strict-dynamic'`.

```
document.write('<script src="https://example.com/script.js");
```

Listing 2.13 – Parser-inserted script

12. In CSP1, the CSP headers were prefixed with X-. The X-Webkit-CSP was also used for delivering policies

To load dynamic scripts, one can inject them as shown in the following Listing 2.14.

```
var script = document.createElement("script");
script.src = "https://example.com/script.js";
document.body.appendChild(script);
```

Listing 2.14 – Non-parser-inserted scripts

Contrary to the CSP in Listing 2.11 where one knows the exact origins from which content can load, in the case of strict CSP, scripts that effectively load are known only at runtime.

Delivering policies

Listing 2.15 shows the deployment of a CSP in enforcement mode in an HTTP header.

```
Content-Security-Policy: default-src https://cdn.example.net;
    child-src 'none'; object-src 'none'
```

Listing 2.15 – Delivering CSP in HTTP header

Listing 2.16 shows the same policy delivered in an HTML meta tag.

```
<meta http-equiv="Content-Security-Policy" content="default-src
    https://cdn.example.net; child-src 'none'; object-src 'none'"
    >
```

Listing 2.16 – Delivering CSP in HTML meta tag

6 Browser extensions

Browser extensions or addons are third party programs, that users can download and add to their browsers, to extend the functionality of browsers, and improve their browsing experience. Irrespective of the browser, the concept of addons or extensions always convey similar characteristics: browser extensions have access to elevated browser APIs, that are not accessible to traditional web applications. In the past, many vendors were providing specific technologies for building extensions that would only run on their own browsers [25, 110, 134, 156]. The WebExtensions API [100] is a cross-browser system, for developing extensions using standard HTML, JavaScript and CSS languages, that can run on many browsers including Google Chrome, Opera, Mozilla Firefox and Microsoft Edge. Interestingly, their specific extensions APIs [2, 25, 100, 110] are compatible with each other to some extent, making it easy to migrate extensions written for a specific browser to other browsers with just a few changes. In the rest of this thesis, unless otherwise specified, when we talk about extensions or addons, we mean cross-browser WebExtensions.
Extensions execute in browsers with elevated privileges. For instance, they can inject scripts (content scripts) to manipulate the DOM of web pages running in the user browser. They are not subject to the Same Origin Policy [125] with respect to their ability to make cross-origin requests. Hence, they can make requests with user credentials to get data from any web application server (See Section 3 on cross-origin requests). They have access to user information stored in the browser such as their cookies, browsing history, bookmarks,etc. They have access to a permanent storage in which data can be persistently stored as long as the extension is installed in the user browser. Examples of popular browser extensions are adblockers, such as AdBlock [6] and password managers, such as LastPass [86].

6.1 Security considerations

Because of the privileged browser features they have access to, extensions represent valuable targets for attackers. To limit the harm that attackers could cause if they compromise an extension, extensions must declare, in a mandatory `manifest.json` file, permissions for the APIs that they effectively use in the extension code. Listing 2.17 shows an example of a manifest file and the `permissions` (features and APIs) the extension will be granted access to at runtime.

```
{
    "permissions": [
      "<all_urls>",
      "storage",
      "management",
      "cookies",
      "history",
      "bookmarks",
      "downloads",
      "webRequest",
      "webRequestBlocking"
    ]
}
```

Listing 2.17 – Permissions declaration in a manifest file

These are only a subset of all the capabilities provided by browsers to extensions. When installed, this extension will be granted full access (read/write) to data on any web application, thanks to the permission `<all_urls>`, called the `host` permission. This implies that if the user is logged into a web application (mailing, banking, social networks, ...), the extension also has access to the user's private data on that application. The rest of the permissions read straightforwardly. The `storage` permission allows the extension to store and retrieve data in the browser. The permissions `management`, `cookies`, `history`, and `bookmarks` give the extension the permission to access and manage the list of installed extensions, cookies, the user's browsing history and bookmarks respectively. With the `downloads` permission, the extension can download and save arbitrary files in the user's device. The `webRequest` and `webRequestBlocking` permissions give an extension the ability to intercept and tamper with HTTP communications between web applications and web servers. In particular, it can add or remove HTTP headers in requests and responses [28,59].

6.2 Architecture

Extensions are made up of in 3 main parts or components.

```
  ...
  "background": {
    "scripts": ["background.js"]
  },
  "content_scripts": [{
    "matches": ["<all_urls>"],
    "scripts": ["content_scripts.js"]
  }],
  "browser_action": {
    "default_icon": "icon.png",
    "default_popup": "popup.htm"
  }
```

```
...
```

Listing 2.18 – Declaring different components of an extension in the manifest file

Listing 2.18 shows the declaration of different components of an extension: the scripts that will execute in the extension background page, the content scripts that will be injected in all web pages, and a UI page (browser action) with an icon, which once clicked, will display a popup for the user to customize the extension for example.
There is a separation in privileges among the different components of an extension. More importantly, webpages and extensions execute in separate contexts. In the contexts of extensions components, extensions specific APIs are all accessible via the `chrome` object in Chrome and Opera browsers [25,110], and via the `browser` object in Firefox and Microsoft Edge [2,100]. The `chrome` or `browser` object are properties of the global object `window` (See Section 1.5). The `document` object allows background scripts to manipulate the background page DOM, the UI pages scripts to manipulate the UI page DOM, and the content scripts to manipulate the webpage DOM

— The background page runs the main logic of the extension. It is composed of a set of scripts that execute in the `background`, without any visual UI. Scripts running in the background page have access to all the permissions of the extension. The background page is in fact an HTML page, which can be directly declared in the `manifest.json` file [14]. When only its scripts are declared (as in our example), the browser generates an HTML page in which the scripts are executed.

— UI pages are meant for the user to interact with the extension, in order for instance, to enable, disable it or customize its behavior with specific settings, etc. Background and UI pages have direct access to each other's DOM and execution contexts.

— Content scripts are injected by the browser to run along webpages. Content scripts run in a separate context, different from the context of background pages, and different also from the context of web pages in which they are injected. Even though they are not granted access to all the extension capabilities, they can directly use the `host` and `storage` permissions to access user data on any web application or to store and retrieve data from the extension storage. Content scripts can manipulate the webpage DOM, and the changes they make to the DOM are visible to scripts executing in the context of webpages. Changes made to the DOM by the webpage are also visible to content scripts. They share the same HTML5 localStorage as webpages (note that this is different from the extension `storage` which they also share with the rest of the extensions components).

6.3 Extensions injected content

Listing 2.19 shows the injection of a script in a webpage, done by the content script.

```
var script = document.createElement("script");
script.src = "https://example.com/script.js"
document.body.appendChild(script)
```

Listing 2.19 – Content injection in webpages DOM

Content injected in the DOM of webpages are visible to the page. In the particular case of scripts, they are executed in the context of the page and not that of the content scripts. Content injected by the content scripts are not restricted by the CSP of the page. That is, the script injected as shown in Listing 2.19 will execute in any page, regardless whether the CSP of the page does not allow content from `https://example.com`. This is the case

for any other remote content injected in the webpage by the extension. However, let's take an example wherein, a content scripts injects a script `A` in the page. `A` will load because it is not applied the CSP of the page. Any content that `A` in turn attempts to inject will be applied the CSP of the page [26, 200].

6.4 Extensions identification

Each extension in browsers supporting the WebExtensions API [100], is assigned a unique identifier that helps to distinguish it from other extensions the user has installed. In Chrome and Opera (and related browsers), the unique identifier is the same and is permanent for the extension regardless of the browser in which it is installed. For instance, the `uBlock Origin` [140] extension is assigned the identifier `cjpalhdlnbpafiamejdnhcphjbkeiagm` in any Chrome browser in which it is installed. On any Opera browser, its unique identifier is `kccohkcpppjjkkjppopfnflnebibpida` [142]. On the contrary, Firefox has adopted a completely different approach: an extension is assigned a randomly unique identifier, called the `UUID`, when it is installed in a Firefox browser [13]. As a consequence, the `uBlock Origin` extension in Firefox will be assigned a different browser-specific identifier for each browser in which it is installed. This identifier remains unique as long as the extension is installed in the browser. In our browser, `a813af59-53ff-4845-bc98-1b820a790ff5` is the identifier of `uBlock Origin` [141].

6.5 Web Accessible Resources

Web accessible resources (WARs) are content in the extension bundle (package) that the extension intents to inject in the DOM of web pages [57, 63]. As we have mentioned previously, content scripts have access to the page DOM, which they can modify by injecting content. Such content can be located on remote servers, or located in the extension package on the user browser. Browsers impose that the extension explicitly declares the content of its package that can be injected in web pages. In the `manifest.json` file, these content are declared using the `web_accessible_resources` key as shown by Listing 2.20.

```
{
    "web_accessible_resources": ["images/*.png", "scripts/*"]
}
```

Listing 2.20 – Declaring web accessible resources (WARs) in the manifest file

Web accessible resources injected in web pages have the following schema.

```
[Ext-Scheme]://[Ext-ID]/[path]
```

Listing 2.21 – Scheme of extensions web accessible resources URLs

The `Ext-Scheme` is the scheme used for extension bundle resources. Chrome and Opera use `chrome-extension` scheme (protocol) while Firefox uses `moz-extension`. The `Ext-ID` is the unique identifier of the extension. Finally `path` is the path to the web accessible resource itself in the extension bundle.
Browsers provide a convenient way for content scripts to inject resources in web pages by specifying only the path to the resource. By invoking `chrome.runtime.getURL("icon.png")`, the browser will generate the URL of the resource `icon.png`, by prepending it with the scheme, and the extension identifier as shown in Listing 2.21.

13. In reality, Firefox extensions also have another identifier, which is permanent as in the case of Chrome. It is used among other things, to update extensions [52] and not for the extension (resources) identification we are going to discuss here

If web accessible resources can be injected by content scripts in web pages, nothing prevents a script in a webpage from also loading a WAR. On Chrome and Opera, extensions identifiers are publicly known. All browsers have a centralized place where extensions can be downloaded and installed in the user's browser [24, 58, 94, 108].
By downloading an extension, one has access to its source code and one can know whether the extension has web accessible resources or not. There are extensions such as the CRX Extension Viewer [277] which lets one display and navigate the source code of other extensions, directly in a browser. Knowing the extension unique identifier and web accessible resources allow a webpage to also load web accessible resources and detect the presence of extensions that a user has installed in her browser [198, 245, 249].

Part I

Content Security Policy

Introduction

Content Security Policy has been first proposed by Stamm et al. [258] and standardized by the W3C, as a refinement of SOP [125], in order to help mitigate Cross-Site-Scripting [278] and data exfiltration attacks.
Since its introduction [275], CSP has gained significant consideration from the research community, with propositions aimed at improving its effectiveness and security [177, 267]. The second version [275] of the specification is supported by all major browsers, and the third version [272] is in an advanced development state. CSP adoption on websites in the wild is growing, even though slowly [177, 255, 267, 269]. To help improve CSP adoption, many tools have been proposed [214, 235, 236].
CSP adoption measurements Even though CSP is well supported by browsers [177], its endorsement by web sites is rather slow. Weissbacher et al. [269] performed the first large scale study of CSP deployment in top Alexa sites, and found that around 1% of sites were using CSP at the time. Calzavara et al. [177] found that nearly 8% of Alexa top sites had CSP deployed in their front pages in 2016. Another study, by Weichselbaum et al. [267] come with similar results to the study of Weissbacher et al. [269].
Tools to ease CSP adoption Almost all authors agree that CSP adoption is not a straightforward task, and lots of (manual) effort are needed in order to reorganize and modify web pages to support CSP. Therefore, in order to help web site developers in adopting CSP, Javed proposed CSP Aider [209] that automatically crawls a set of pages from a site and proposes a site-wide CSP. Patil and Frederik [236] proposed UserCSP, a framework that monitors the browser internal events in order to automatically infer a CSP for a web page based on the loaded resources. Pan et al. [235] proposed CSPAuto-Gen, to enforce CSP in real-time on web pages, by rewriting them on the fly client-side. Weissbacher et al. [269] have evaluated the feasibility of using CSP in report-only mode in order to generate a CSP based on reported violations, or semi-automatically inferring a CSP policy based on the resources that are loaded in web pages. They concluded that automatically generating a CSP is ineffective. Another difficulty is the use of inline scripts in many pages. The first solution is to externalize inline scripts, as can be done by systems like deDacota [184]. Kerschbaumer et al. [214] find that too many pages are still using `'unsafe-inline'`in their CSPs. They propose a system to automatically identify legitimate inline scripts in a page, thereby whitelisting them in the CSP of the underlying page, using script hashes.
Evaluating CSP effectiveness Another direction of research on CSP, has been evaluating its effectiveness at successfully preventing content injection attacks. Calzavara et al. [177] found out that many CSP policies in real web sites had errors including typos, ill-formed or harsh policies. Even when the policies were well formed, they found that almost all deployed CSP policies were bypassable because of a misunderstanding of the CSP language itself. Johns [212] first demonstrated that insecure JSONP endpoints can lead to bypasses, and proposed a server-side templating mechanism for safely assembling code and data to

prevent such attacks. Weichselbaum et al. [267] also showed many other subtle bypasses which make CSP ineffective at preventing attacks. Patil and Frederik found similar errors in their study [236]. Van Acker et al. [168] have shown that CSP fails at preventing data exfiltration specially when resources are prefetched, or in presence of a CSP policy in the HTML meta tag, because the order in which resources are loaded in a web application is hard to predict. Hausknecht et al. [200] found that some browser extensions, modified the CSP policy headers, in order to whitelist more resources and origins. This can potentially alter the effectiveness of CSP at mitigating attacks.

Improving CSP Expressiveness Johns [211] proposed hashes for static scripts, and PreparedJS, an extension for CSP, in order to securely handle server-side dynamically generated scripts based on user input. Weichselbaum et al. [267] have extended nonces and hashes, introduced in CSP level 2 [275], to remote scripts URLs, specially to tackle the high prevalence of insecure hosts in current CSP policies. They proposed whitelisting scripts with nonces and hashes instead of origins, to prevent bypasses due to JSONP and open redirects. They first introduced strict CSP, and more specifically the `'strict-dynamic'` keyword for easily loading dynamic content. This keyword states that any additional script loaded by a whitelisted script is considered a trusted script as well. They also provide guidelines on how to build an effective CSP. Nonces are included in the DOM, and their security is questionable. Furthermore, the trust propagation enabled by `'strict-dynamic'` only applies to scripts and stylesheets, and is too liberal since it allows any whitelisted script to further inject any other script without restrictions. To limit this trust propagation, Calzavara et al. [178] proposed Compositional Content Security Policy (CCSP). In their proposal, scripts which are included in the application are individually whitelisted in the CSP of the application, instead of whitelisting their origins. Furthermore, each of them is assigned an upper bound in the additional content it can further inject in the application. The upper bound is a CSP specifying which additional content a whitelisted script can further load. Besides requiring adoption by the CSP specification, CCSP also requires content providers to declare all the dependencies needed by content that they host. This helps developers build the upper bounds when including such content in their policies.

CSP and SOP

CSP is a page-specific policy. Jackson and Barth [208] have shown that page-specific policies can be bypassed by origin-wide policies. Though, their work predates CSP. In our work [255] presented in Chapter 3 we demonstrate that this also applies to CSP, by analyzing the interactions between CSP (a page-specific policy) and the SOP (an origin-wide policy). The SOP allows same-origin pages to directly access each other execution contexts. When same-origin pages do not set the same CSP restrictions on scripts they load, then when one page embeds another as an iframe, CSP violations can occur. In these settings, the deployed CSP becomes ineffective against attacks propagating from same-origin pages not protected with CSP. To effectively protect a web page against attacks with a CSP, one then has to ensure that same-origin pages it embeds are also protected against attacks, otherwise, an attacker can target such pages, and propagate the attack to CSP-protected pages thanks to the Same Origin Policy. We also extend previous results on CSP measurements by analyzing the adoption of CSP by site, not only considering front pages but all the pages in a site. We have been regularly (monthly) collecting statistics about CSP adoption on top 10k Alexa sites. Figure 2.1 shows the evolution on CSP adoption. It is interesting to note a constant growth in CSP adoption among top sites homepages. In April 2016, only 2.1% of them had adopted CSP. A year later, in April

2017, it is 3.8% of them which had CSP deployed. Finally, in April 2018, 7.3% of top 10k Alexa sites deploy CSP. This result is very encouraging from a security perspective. Even though they may be changes in the sites which are in the top 10k sites, we can say that popular websites owners are more and more aware of CSP.

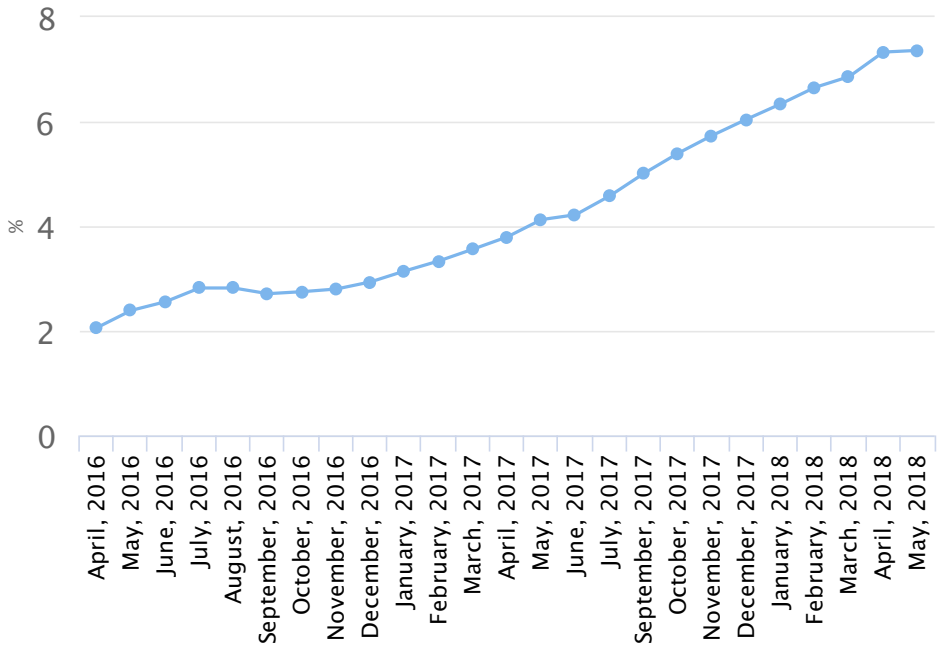


Figure 2.1 – Evolution of CSP adoption among top 10,000 Alexa Sites between April 2016 and April 2018 - Source [153]

Dependency-Free CSP

CSP has three versions: CSP1 [261], CSP2 [275] and CSP3 [272]. Each version builds on the previous one, adding more features, modifying the semantics of some or removing others. Furthermore, not all browser vendors are implementing the same version of CSP, and not all implementations are compliant with the specification.
As an application developer, one has to ensure that a CSP deployed with a page, will successfully protect it against attacks, and preserve the functionality of the application, no matter the browser in which the application executes, and the specific implementation of CSP in the browser. This area has so far received no attention. In Chapter 4, we fill this gap by formalizing the differences in CSP versions and browsers implementations as CSP directives dependencies. Then we propose a set of rewriting rules and a tool for developers to build dependency-free policies (`DF-CSP`) whose semantics are independent of CSP versions and browsers implementations. Such policies help to protect applications against attacks while preserving their functionality in all browsers.
Dependency-free policies are also useful to reason about CSP policies with the formal semantics of Calzavara et al. [179] that calculates the global meaning of CSP policies by adding the meanings of each individual directive. In this formal semantics, the global semantics of a CSP policy follows from the semantics of individual directive values. This is however not correct unless the CSP does not present any dependencies, because the meaning of individual directives could be altered by other directives.

Finally, we discuss the security implications of the use of `'strict-dynamic'` in policies [267, 272]. In fact, the use of `'strict-dynamic'` in backwards compatible policies such as `DF-CSP`, can give attackers different attack power depending on the version of CSP considered, especially in CSP3 where an attacker could potentially inject arbitrary content in the application. We show that automatically generating a second policy out of a policy that makes use of `'strict-dynamic'`, successfully ensures that an attacker who compromises a script allowed by a `DF-CSP`, does not gain more power, irrespective of the browser in which the application executes. This limits the trust propagation mechanism of CSP [267] with a global upper bound for all scripts in the page. This is different from the proposal of individual upper bounds proposed by Calzavara et al. [178] with the advantage of requiring no modification to the current CSP specification.

CSP limitations: proposals for extending the specification

Previous works have demonstrated limitations of CSP [177, 178, 211, 267], as a whitelisting mechanism. An origin added to a policy implies that any content from that origin is trusted. In fact, it is not possible to exclude (blacklist) specific content, even though one knows that they are potentially malicious. The only solution would be, to individually whitelist the trusted content, except the untrusted ones. In addition to being impossible in case only specific content are untrusted, partially whitelisting an origin is bypassable using HTTP redirections [267, 272, 275]. Moreover URLs parameters are considered safe by default. Nonetheless, many attacks have been demonstrated, where attackers leverage URL parameters to bypass CSP [211, 267]. Browser extensions are widespread and can alter the CSP of webpages, introducing vulnerabilities in webpages [200]. The use of `'strict-dynamic'` makes it difficult to assess what content will effectively load in the page.
Weichselbaum et al. [267] proposed the use of nonces to mitigate CSP bypasses. This solution has however many shortcomings. First, the security of nonces has been questioned [178, 272, 275]. They do not mitigate attacks done by whitelisted scripts, especially when they get compromised. Finally, nonces apply only to scripts and stylesheets. The CSP violations reporting mechanism fails at successfully reporting all content that effectively load in a webpage, because it is inefficient (require the deployment of two policies) and incomplete (do not report content injected by browser extensions).
In Chapter 5, we propose extending CSP specification in order to successfully address the aforementioned issues. To do so, we discuss 4 extensions to the current CSP specification: the ability to blacklist content, express more fine grained checks on URL arguments, explicitly prevent redirections to partially whitelisted origins, and a reporting mechanism for content that are allowed by a CSP enforced on a webpage. While requiring few changes to the specification, these extensions successfully mitigate the shortcomings of CSP in its current state and attacks that have been demonstrated in the wild. The reporting mechanism provides an efficient way for collecting useful feedback about the runtime enforcement of CSP as done by the browser, and can help improve the effectiveness of policies deployed to protect webpages. Finally, we demonstrate an implementation of the proposed extensions using service workers.

Chapter 3

Content Security Policy and the Same Origin Policy

Preamble

This chapter describes the interplay between the Content Security Policy (CSP) mechanism and the Same Origin Policy. In particular, CSP is a page-specific policy, while the SOP applies to all same-origin pages. We show how CSP may be violated due to the SOP when a page has a CSP, but other same-origin pages it can directly interact with (say, a parent page and its same-origin iframes) do not have CSP. This chapter is a replication of the paper titled "On the Content Security Policy Violations Due to the Same-Origin Policy" which was published in the proceedings of the 26th International Conference on World Wide Web (WWW) in 2017.

1 Introduction

In this chapter, we report on a fundamental problem of CSP. CSP [275] defines how to protect content in an isolated page. However, it does not take into consideration the page's context, that is its embedder or embedded iframes. In particular, CSP is unable to protect content of its corresponding page if the page embeds (using the `src` attribute) an iframe of the same origin. The CSP policy of a page will not be applied to an embedded iframe. However, due to SOP, the iframe has complete access to the content of its embedder. Because same-origin iframes are transparent due to SOP, this opens loopholes to attackers whenever the CSP policy of an iframe and that of its embedder page are not compatible (see Figure. 3.1).

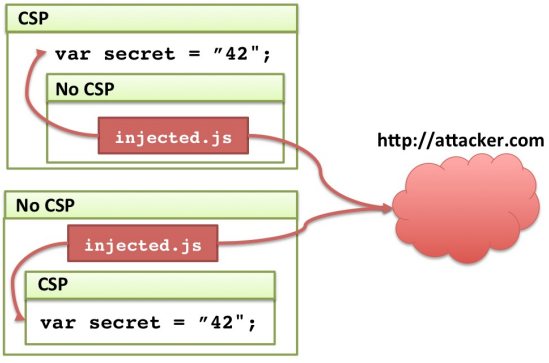


Figure 3.1 – An XSS attack despite CSP.

We analyzed 1 million pages from the top 10,000 Alexa sites and found that 5.29% of sites contain some pages with CSPs (as opposed to 2% of home pages in previous studies [177]). We identified that in 94% of cases, CSP may be violated in presence of the document.domain API and in 23.5% of cases CSP may be violated without any assumptions (see Table 3.2).
We also identified a divergence among browsers implementation of the enforcement of CSP [275] in sandboxed iframes embedded with `srcdoc`. This actually reveals an inconsistency between the CSP and HTML5 sandbox attribute specification for iframes.
We identify and discuss possible solutions from the developer point of view as well as new security specifications that can help prevent this kind of CSP violations. We have made publicly available the dataset that we used for our results in [42]. We have installed an automatic crawler to recover the same dataset every month to repeat the experiment taking into account the time variable. An accompanying technical report with a complete account of our analyses can be found at [253].
In summary, our contributions are: (i) We describe a new class of vulnerabilities that lead to CSP violations. (Section 1). (ii) We perform a large and depth scale crawl of top sites, highlighting CSP adoption at sites-level, as well as sites origins levels. Using this dataset, we report on the possibilities of CSP violations between the SOP and CSP in the wild. (Section 3). (iii) We propose guidelines in the design and deployment of CSP. (Section 6). (iv) We reveal an inconsistency between the CSP specification and HTML5 sandbox attribute specification for iframes. Different browsers choose to follow different specifications, and we explain how any of these choices can lead to new vulnerabilities. (Section 5).

2 Content Security Policy and SOP

CSP is a page-specific policy. A CSP delivered with a page controls the resources of the page. However it does not apply to the page's embedding resources [275]. As such, CSP does not control the content of an iframe even if the iframe is from the same origin as the main page according to SOP. Instead, the content of the iframe is controlled by the CSP delivered with it, that can be different from the CSP of the main page.

2.1 CSP violations due to SOP

Consider a web application, where the main page `A.html` and its iframe `B.html` are located at `http://main.com`, and therefore belong to the same origin according to the Same Origin Policy [125]. `A.html`, shown in Listing 3.1, contains a script and an iframe from `main.com`. The local script `secret.js` contains sensitive information given in Listing 3.2. To protect against XSS, the developer has installed the CSP for its main page `A.html`, shown in Listing 3.3.

```
<html>
  <script src="secret.js"></script>
   ...
  <iframe src="B.html"></iframe>
</html>
```

Listing 3.1 – Source code of `http://main.com/A.html`.

```
var secret = "42";
```

Listing 3.2 – Source code of `secret.js`.

```
default-src 'none'; script-src 'self'; child-src 'self'
```

Listing 3.3 – CSP of `http://main.com/A.html`.

This CSP provides an effective protection against XSS:

Only the parent page has CSP

According to CSP2[1], only the CSP of the iframe applies to its content, and it completely ignores the CSP of the including page. In our case, if there is no CSP in `B.html` then its resource loading is not restricted. As a result, an iframe `B.html` without CSP is potentially vulnerable to XSS, since any injected code may be executed within `B.html` with no restrictions. Assume `B.html` was exploited by an attacker injecting a script `injected.js`. Besides taking control over `B.html`, this attack now propagates to the including page `A.html`, as we show in Fig. 3.1. The XSS attack extends to the including parent page because of the inconsistency between the CSP and SOP. When a parent page and an iframe are from the same origin according to SOP, a parent and an iframe share the same privileges and can access each other's code and resources.
As per our example, `injected.js` is shown in Listing 3.4.
This script executed in `B.html` retrieves the secret value from its parent page (`parent.secret`) and transmits it to an attacker's server `http://attacker.com` via XMLHttpRequest[2].

```
function sendData(obj, url){
  var req = new XMLHttpRequest();
  req.open('POST', url, true);
  req.send(JSON.stringify(obj));
}
sendData({secret: parent.secret}, 'http://attacker.com/send.php'
    );
```

Listing 3.4 – Source code of `injected.js`.

A straightforward solution to this problem is to ensure that the protection mechanism for the parent page also propagates to the iframes from the same domain. Technically, it means that the CSP of the iframe should be the same or more restrictive than the CSP of the parent. In the next example we show that this requirement does not necessarily prevent possible CSP violations due to SOP.

Only the iframe has CSP

Consider a different web application, where the including parent page `A.html` does not have a CSP, while its iframe `B.html` contains a CSP from Listing 3.3. In this example, `B.html`, shown in Listing 3.5 now contains some sensitive information stored in `secret.js` (see Listing 3.2).

```
<html>
   ...
  <script src="secret.js"></script>
</html>
```

Listing 3.5 – Source code of `http://main.com/B.html`.

1. https://www.w3.org/TR/CSP2/#which-policy-applies
2. The XMLHttpRequest is not forbidden by the SOP for `B.html` because an attacker has activated the Cross-Origin Resource Sharing mechanism [265] on her server `http://attacker.com`.

Since the including page `A.html` now has no CSP, it is potentially vulnerable to XSS, and therefore may have a malicious script `injected.js`. The iframe `B.html` has a restrictive CSP, that effectively contributes to protection against XSS. Since `A.html` and `B.html` are from the same origin, the malicious injected script can profit from this and steal sensitive information from `B.html`. For example, the script may call the `sendData` function with the secret information:

```
sendData({secret: children[0].secret}, 'http://attacker.com/
    send.php');
```

Thanks to SOP, the script `injected.js` fetches the secret from it's child iframe `B.html` and sends it to `http://attacker.com`.

CSP violations due to origin relaxation

A page may change its own origin with some limitations. By using the `document.domain` API, the script can change its current domain to a superdomain. As a result, a shorter domain is used for the subsequent origin checks [3].
Consider a slightly modified scenario, where the main page `A.html` from `http://main.com` includes an iframe `B.html` from its sub-domain `http://sub.main.com`. Any script in `B.html` is able to change the origin to `http://main.com` by executing the following line:

```
document.domain = "main.com";
```

If `A.com` is willing to communicate with this iframe, it should also execute the above-written code so that the communication with `B.html` will be possible. The content of `B.html` is now treated by the web browser as the same-origin content with `A.html`, and therefore any of the previously described attacks become possible.

Categories of CSP violations due to SOP

We distinguish three different cases when the CSP violation might occur because of SOP:

Only the parent page or iframe has CSP : a parent page and an iframe page are from the same origin, but only one of them contains a CSP. The CSP may be violated due to the unrestricted access of a page without CSP to the content of the page with CSP. We demonstrated this example in Sections 2.1 and 2.1.

Parent and iframe have different CSPs A parent page and an iframe page are from the same origin, but they have different CSPs. Due to SOP, the scripts from one page can interfere with the content of another page thus violating the CSP.

CSP violation due to origin relaxation A parent page and an iframe page have the same higher level domain, port and scheme, but however they are not from the same origin. Either CSP is absent in one of them, or they have different CSPs – in both cases CSP may be violated because the pages can relax their origin to the high level domain by using `document.domain` API, as we have shown in Section 2.1.

3 Empirical study of CSP violations

We performed a large-scale study on the top 10,000 Alexa sites to detect whether CSP may be violated due to an inconsistency between CSP and SOP. To collect the data, we

3. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Changing_origin

used CasperJS [238] on top of PhantomJS headless browser [205]. The User-Agent HTTP header was instantiated as a Google Chrome browser version 51.

3.1 Methodology

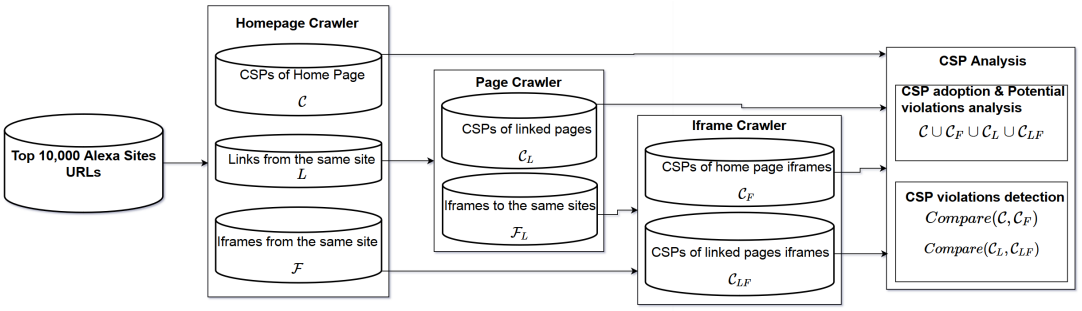


Figure 3.2 – Data Collection and Analysis Process

The overview of our data collection and CSP comparison process is given in Figure 3.2. The main difference in our data collection process from previous works on CSP measurements in the wild [177, 267] is that we crawled not only the main pages of each site, but also pages with at least the same TLD+2 component as the site. First, we collected pages accessible through links of the main page and pointing to the same site. Second, to detect possible CSP violations due to SOP, we collected all the iframes present on the home pages and linked pages.

Data collection

Home page crawler For each site in the top 10,000 Alexa list, we crawled the home page, parsed its source code and extracted three elements:

(1) a CSP of the site's home page stored in HTTP header as well as in `<meta>` HTML tag; we denoted the CSPs of the home page by $\mathcal{C}$;

(2) to extract more pages from the same site, we analyzed the source of the links via `<a href=...>` tag and extracted URLs that point to the same site, we denoted this list by L.

(3) we collected URLs of iframes present on the home page via `<iframe src=...>` tag and recorded only those belonging to the same site, we denoted this set by $\mathcal{F}$.

Page crawler We crawled all the URLs from the list of pages L, and for each page we repeated the process of extraction of CSP and relevant iframes, similar to the steps (1) and (3) of the home page crawler. As a result, we got a set of CSPs of linked pages $\mathcal{C}_L$ and a set of iframes URLs $\mathcal{F}_L$ that we extracted from the linked pages in L.

Iframe crawler

For every iframe URL present in the list of home page iframes $\mathcal{F}_H$, and in the list of linked pages iframes $\mathcal{F}_L$, we extracted their corresponding CSPs and stored in two sets: $\mathcal{C}_F$ for home page iframes and $\mathcal{C}_{LF}$ for linked page iframes.

CSP adoption analysis

Since CSP is considered an effective countermeasure for a number of web attacks, programmers may use it to mitigate such attacks on the main pages of their sites. However, if CSP

is not installed on some pages of the same site, this can potentially leak to CSP violations due to the inconsistency with SOP when another page from the same origin is included as an iframe (see Figure 3.1). In our database, for each site, we recorded its home page, a number of linked pages and iframes from the same site. This allowed us to analyse how CSP is adopted at every popular site by checking the presence of CSP on every crawled page and iframe of each site. To do so, we analyzed the extracted CSPs: $\mathcal{C}$ for the home page, $\mathcal{C}_L$ for linked pages, $\mathcal{C}_F$ for home page iframes, and $\mathcal{C}_{LF}$ for linked pages iframes.

CSP violations detection

To detect possible CSP violations due to SOP, we analyzed home pages and linked pages from the same site, as well as iframes embedded into them.

CSP selection

To detect CSP violations, we first removed all the sites where no parent page and no iframe page contained a CSP. For the remaining sites, we pointwise compared (1) the CSPs of the home pages $\mathcal{C}$ and CSPs of iframes present on these pages $\mathcal{C}_F$; (2) the CSPs of the linked pages $\mathcal{C}_L$ and CSPs of their iframes $\mathcal{C}_{LF}$. To check whether a parent page CSP and an iframe CSP are equivalent, we applied the CSP comparison algorithm (Figure 3.2)

CSP preprocessing We first normalized each CSP, by splitting it into its directives.

— If **default-src** directive was present (**default-src** is a fallback for most of the other directives), we extracted the source list s of **default-src**. We analyzed which directives were missing in the CSP, and explicitly added them with the source list s.

— If **default-src** directive was missing, we computed the list of directives not present in the CSP. In this case, there are no restrictions in CSP for every absent directive. We therefore explicitly added them with the most permissive source list. A missing **script-src** is assigned *** 'unsafe-inline' 'unsafe-eval'** as the most permissive source list [275].

— In each source list, we modified the special keywords: (i) **'self'** was replaced with the origin of the page containing the CSP; (ii) in the case both **'unsafe-inline'** and hashes or nonces are in the source list, we removed **'unsafe-inline'** from the directive since it will be ignored by the CSP2 [275]. (iii) **'none'** keywords were removed from all the directives; (iv) nonces and hashes were removed from all the directives since they cannot be compared; (iv) each whitelisted domain was extended with a list of schemes and port numbers from the URL of the page where the CSP was deployed [4].

CSP comparison We compared all the directives present in the two CSPs to identify whether the two policies required the same restrictions. Whenever the two CSPs were different, our algorithm returned the names of directives that did not match. The demonstration of the comparison is accessible on [42]. For each directive in the policies we compared the source lists and the algorithm proceeded if the elements of the lists were identical in the normalized CSPs.

Limitations

Our methodology and results have two(2) limitations that we explain here.

4. For example, according to CSP2, if the page scheme is `https`, and a CSP contains a source `example.com`, then the user agent should allow content only from `https://example.com`, while if the current scheme is `http`, it would allow both `http://example.com` and `https://example.com`.

Sites successfully crawled	9,885
Pages visited	1,090,226
Pages with iframe(s) from the same site	648,324
Pages with same-origin iframe(s)	92,430
Pages with same-origin iframe(s) where page and/or iframe has CSP	692
Pages with CSP	21,961 (2.00%)
Sites with CSP on home page	228 (2.3%)
Sites with CSP on some pages	523 (5.29%)

Table 3.1 – Crawling statistics

User interactions The automatic crawling process did not include any real-user-like interactions with top sites. As such, the set of iframes and links URLs we analyzed is an underestimate of all links and iframes a site may contain.
Pairs of (parent-iframe) In this work, we considered CSP violations in same origin (parent, iframe) couples only. There are further combinations such as couples of sibling iframes in a parent page that we could have considered. Overall, our results are conservative, since the problem might have been worst without those limitations.

3.2 Results on CSP adoption

The crawling of Alexa top 10,000 sites was performed in the end of August, 2016. To extract several pages from the same site, we also crawled all the links and iframes on a page that pointed to the same site. In total, we gathered 1,090,226 pages from 9,885 different sites. As a median, from each site we extracted 45 pages, with a maximum number of 9,055 pages found on `tuberel.com`. Our crawling statistics is presented in Table 3.1. More than half of the pages contained an iframe, and 13% of pages did contain an iframe from the same site. This indicates the potential surface for the CSP violations, when at least one page on the site has a CSP installed. We discuss such potential CSP violation in details in Section 3.3. Similarly to previous works on CSP adoption [177, 267], we found that CSP was present on only 228 out of 9,885 home pages (2.31%). Extending this analysis to almost a million pages, we found a similar rate of CSP adoption (2.00%).

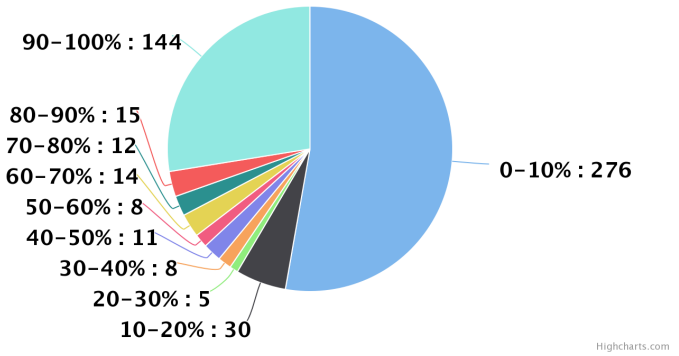


Figure 3.3 – Percentage of pages with CSP per site

Differently from previous studies that analyzed only home pages, or only pages in separation, we analyzed how many sites have at least some pages that adopted CSP. We grouped

	Same-origin parent-iframe	Possible to relax origin	Total
Only parent page has CSP	**8**3	**1**,388	1,471
Only iframe has CSP	**1**6	**2**40	256
Different CSPs in parent page and iframe	**7**0	**4**4	114
No CSP violations	551	109	660
CSP violations total	**1**69 (23.5%)	**1**,672 (94%)	1841

Table 3.2 – Statistics CSP violations due to Same-Origin Policy

	Same-origin parent-iframe	Possible to relax origin
Only parent page CSP	yandex.ru	twitter.com, yandex.ru, mail.ru
Only iframe CSP	amazon.com, imdb.com	–*
Different CSP	twitter.com	–*

*Not found in top 100 Alexa sites.

Table 3.3 – Sample of sites with CSP violations due to Same-Origin Policy

all pages by sites, and found that 5.29% of sites contain some pages with CSPs. This means website developers are aware of CSP which for some reasons is not widely adopted on all the pages of the site.
We then analyzed how many pages on each site have adopted CSPs. For each of the 523 sites, we counted how many pages (including home page, linked pages and iframes) have CSPs. Figure 3.3 shows that more than half of the sites had a very low CSP adoption on their pages: on 276 sites out of 529, CSP is installed on only 0-10% of their pages. This becomes problematic if other pages without CSP are not XSS-free. However, it is interesting to note that around a quarter of sites do profit from CSP by installing it on 90-100% of their pages.

3.3 Results on CSP violations due to SOP

As described in Section 2.1, we distinguish several categories of CSP violations when a parent page and an iframe on this page are from the same origin according to SOP. To account for possible CSP violations, we only considered cases when either parent, or iframe, or both have a CSP installed. From all the 21,961 pages that have CSP installed, we removed the pages, where CSPs are in report-only mode, leaving 18,035 pages with CSPs in enforcement mode.
Table 3.2 presents possible CSP violations due to SOP.
We extracted the parent-iframe couples that might cause a CSP violation either because (1) only the parent or the iframe installed a CSP, or (2) both installed different CSPs. First, to account for direct violations because of SOP, we distinguished couples where parent and iframe were from the same origin (columns 2,3), we found 720 cases of such couples. Second, we analyzed possible CSP violations due to origin relaxation: we have collected 1,781 couples that are from different origins but their origins could be relaxed by `document.domain` API (see more in Section 2.1) – these results are shown in column 3.
In Table 3.3 we present the names of the domains out of the top 100 Alexa sites, where we found different CSP violations. Each company in this table has been notified about the possible CSP violations. Concrete examples of the page and iframe URLs and their corre-

sponding CSPs for such violations can be found in the corresponding technical report [253]. All the collected data is available online [42].

CSP violations in presence of `document.domain` According to our results, in presence of `document.domain`, 94% of (parent, iframe) pages can have their CSP violated. Those violations can occur only if both parent and iframes pages execute `document.domain` to the same top level domain. Thus, our result is an over-approximation, assuming that `document.domain` is used in all of those pages and iframes. According to [27], `document.domain` is used in less than 3% of web pages.

Only the parent page or the iframe has CSP

We first considered a scenario where a parent page and an iframe are from the same origin, but only one of them contains a CSP. Intuitively, if only a parent page has CSP, then an iframe can violate CSP by executing any code and accessing the parent page's DOM, inserting content, access cookies etc. Among 720 parent-iframe couples from `the same origin`, we found 83 cases (11.5%) where only the parent had a CSP, and 16 cases (2.2%) where only the iframe had a CSP. These CSP violations originated from 13 (for parent) and 4 (for iframe) sites. For example, such possible violations are found on some pages of `amazon.com`, `yandex.ru` and `imdb.com` (see Table 3.3). CSP of a parent or iframe may also be violated because of `origin relaxation`. We identified 1,388 cases (78%) of parent-iframe couples where such violation may occur because CSP is present only in the parent page. This was observed on 20 different sites, including twitter.com, yandex.ru and others. Finally, in 240 cases (13.5%) only the iframe had CSP installed, which was found on 11 different sites. We manually checked the parent and iframes involved in CSP violations for sites in Table 3.3. In all of those sites, either the parent or the iframe page was a login page [42]. We further checked how effective the CSP of those pages were, using CSPEvaluator [5], proposed by Weichselbaum et al. [267]. We found out that the CSPs involved in these pages are all bypassable.

Parent and iframe have different CSPs

In a case where a page and iframe are from the same origin, but their corresponding CSPs are different, this may also cause a violation of CSP. From the 720 `same-origin` parent-iframe couples, we found 70 cases (9.7%) (from 3 sites) where their CSPs differed, and for `an origin relaxation` (from 6 sites) case, we identified only 44 such cases (2.5%). This setting was found on some pages of `twitter.com` for instance.

We further analyzed the differences in CSPs found on parent and iframe pages. For all the 114 pairs of parent-iframe (either same-origin or possible origin relaxation), we compared the policies they installed, directive-by-directive. Figure 3.4 shows that every parent CSP and iframe CSP differed on almost every directive – between 90% and 100%. The only exception is `frame-ancestors` directive, which is almost the same in different parent pages and iframes. If properly set, this directive gives a strong protection against clickjacking attacks [244], therefore equally protecting all the pages from the same origin.

Potential CSP violations

A `potential CSP violation` may happen in a site, where some pages have CSP and some others do not, or pages have different CSPs. When those pages get nested as parent-iframe, we can run into CSP violations, just like in the direct CSP violation cases we

5. https://csp-evaluator.withgoogle.com/

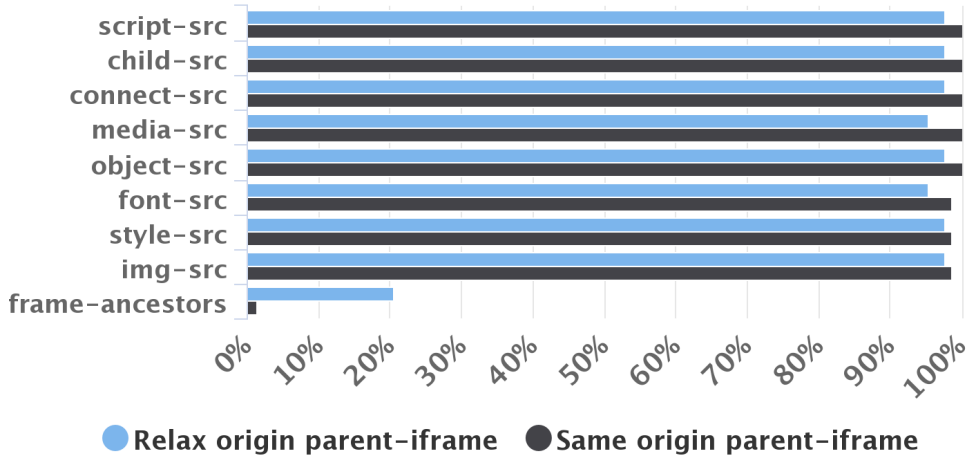


Figure 3.4 – Differences in CSP directives for parent and iframe pages

	Pages	Origins	Sites
A same origin page has no CSP	4381	197	197
A same origin page has a different CSP	1223	23	23
Total Potential violations due to same origin pages	**5**604 (31.1%)	-	-
A same origin (after relaxation) page has no CSP	4728	340	183
A same origin (after relaxation) has a different CSP	2567	135	44
Total Potential violations due to same origin (after relaxation)	**7**295(40.4%)	-	-
Potential violations total	12899 (72%)	591 (81%)	379 (52%)

Table 3.4 – Potential CSP violations in pages with CSP

reported above. To assess how often such violations may occur, we analyzed the 18,035 pages that had CSP in enforcement mode. These pages originated from 729 different origins spread over 442 sites. Table 3.4 shows that 72% of CSPs (12,899 pages) could be potentially violated, and these CSPs originated from pages of 379 different sites (85.75%). To detect these violations, for each page with a CSP in our database, we checked whether there existed another page from the same origin that did not have CSP. The page without a CSP could embed the page with CSP, leading to CSP violations because of SOP. We detected 4,381 such pages (24%) from 197 origins. Similarly, we detected 1,223 same-origin pages (7%) with different CSPs. We also analyzed cases where potential CSP violations may happen due to origin relaxation. We detected 4,728 pages (26%), whose CSP may be violated because of other pages with no CSP, and 2,567 pages (14%), whose CSP may be violated because of different CSPs on other relaxable-origin pages.
For the pages that have different CSPs, we compared how much their CSPs differed. Figure 3.5 shows that CSPs mostly differed in the `script-src` directive, which protects pages from XSS attacks [41]. This means, that if one page in the origin does whitelist an attacker's domain or an insecure endpoint [267], all the other pages in the same origin become vulnerable because they may be inserted as an iframe to the vulnerable page and their CSPs could be easily violated.

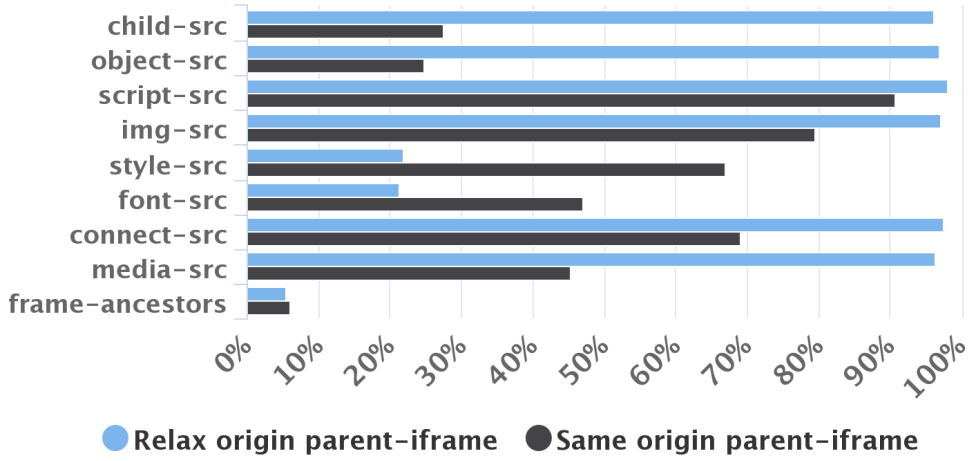


Figure 3.5 – Differences in CSP directives for same-origin and relaxed origin pages

3.4 Responses of websites owners

We reported those issues to a sample of site owners, using either HackerOne [6], or contact forms when available. Here are some selected quotes from our discussions with them.

> "*Yes, of course we understand the risk that under some circumstances XSS on one domain can be used to bypass CSP on another domain, but it's simply impossible to implement CSP across all (few hundreds) domains at once on the same level. We are implementing strongest CSP currently possible for different pages on different domains and keep going with this process to protect all pages, after that we will strengthen the CSP. We believe it's better to have stronger CSP policy where possible rather than have same weak CSP on all pages or not having CSP at all. Having in mind there are hundreds of domains within mail.ru, at least few years are required before all pages on all domains can have strong CSP.*" – `Mail.ru`
>
> "*[...]the sandbox is a defense in depth mitigation[...]. We definitely don't allow relaxing document.domain on www.dropbox.com[...]*" – `Dropbox.com`
>
> "*While this is an interesting area of research, are you able to demonstrate that this behavior is currently exploitable on Twitter? It appears that the behavior you have described can increase the severity of other vulnerabilities but does not pose a security risk by itself. Is our understanding correct? [...]We consider this to be more of a defensive in depth and will take into account with our continual effort to improve our CSP policy*" – `Twitter.com`
>
> "*I believe we understand the risk as you've described it.*" – `Imdb.com`

4 Avoiding CSP violations

Preventing CSP violations due to SOP can be achieved by having the **same** effective CSP for **all** same-origin pages in a site, and preventing origin relaxation.

Origin-wide CSP: Using CSP for all same-origin pages can be manually done but this solution is error-prone. A more effective solution is the use of a specification such as Origin Policy [274] in order to set a header for the whole origin.

6. https://hackerone.com

Preventing origin relaxation: Having an origin-wide CSP is not enough to prevent CSP violations. By using origin relaxation, pages from different origins can bypass the SOP [248]. Many authors provide guidelines on how to design an effective CSP [267]. Nonetheless, even with an effective CSP, an embedded page from a different origin in the same site can use `document.domain` to relax its origin. Preventing origin relaxation is therefore trickier.
Programmatically, one could prevent other scripts from modifying `document.domain` by making a script run first in a page [262]. The first script that runs on the page would be:

```
Object.defineProperty(document, "domain", { __proto__: null,
    writable: false, configurable: false});
```

A parent page can also indirectly disable origin relaxation in iframes by sandboxing them. This can be achieved by using **sandbox** as an attribute for iframes or as directive for the parent page CSP. Unfortunately, an iframe cannot indirectly disable origin relaxation in the page that embeds it. However, the **frame-ancestors** directive of CSP gives an iframe control over the hosts that can embed it. Finally, a more robust solution is the use of a policy to deprecate `document.domain` as proposed in the draft of Feature policy [276]. The feature policy defines a mechanism that allows developers to selectively enable and disable the use of various browser features and APIs.
Iframe sandboxing: Combining attribute **allow-scripts** and **allow-same-origin** as values for **sandbox** successfully disables `document.domain` in an iframe[7]. We recommend the use of **sandbox** as a CSP directive, instead of an HTML iframe attribute. The first reason is that **sandbox** as a CSP directive, automatically applies to all iframes that are in a page, avoiding the need to manually modify all HTML iframe tags. Second, the **sandbox** directive is not programmatically accessible to potentially malicious scripts in the page, as is the case for the **sandbox** attribute (which can be removed from an iframe programmatically, replacing the sandboxed iframe with another identical iframe but without the **sandbox** attribute).
Limitations An origin-wide CSP (the same CSP for all same origin pages) can become very liberal if all same origin pages do not require the same restrictions. In order to implement the solution we propose, one needs to consider the intended relation between a parent page and an iframe page, in presence of CSP. In the case where the two(2) pages should be allowed direct access to each other's context, then, since same origin pages can bypass page-specific security characteristics [208], the solution is to have the same CSP for both the page and the iframe. However, if direct access to each other's context is not a required feature, one can keep different CSPs in parent and iframe, or have no CSP at all in one of the parties, but their contents should be isolated from each other. The solution here is to use sandboxing. Nonetheless, there are other means (such as `postMessage`) by which one can securely achieve communication between the pages.

5 Inconsistent implementations

Combining origin-wide CSP with **allow-scripts sandbox** directive would have been sufficient at preventing the inconsistencies between CSP and the same origin policy. Unfortunately, we have discovered that for some browsers, this solution is not sufficient. Starting

7. We found out that `dropbox.com` actually puts **sandbox** attribute for all its iframes, and therefore avoids the possible CSP violations. We have had a very interesting discussion on `Hackerone.com` with Devdatta Akhawe, a Security Engineer at Dropbox, who told us more about their security practices regarding CSP in particular.

from HTML5, major browsers, apart from Internet Explorer, support the new **srcdoc** attribute for iframes [203]. Instead of providing a URL whose content will be loaded in an iframe (using the `src` attribute), one provides directly the HTML content of the iframe in the **srcdoc** attribute. According to CSP2 [275], §5.2, the CSP of a page should apply to an iframe whose content is supplied in a **srcdoc** attribute. This is actually the case for all majors browsers, which support the **srcdoc** attribute. However, there is a problem when the **sandbox** attribute is set to an **srcdoc** iframe.
Webkit-based[8] and **Blink**-based[9] browsers (Chrome, Chromium, Opera) always comply with CSP. The CSP of a page will apply to all **srcdoc** iframes, even in those iframes which have a different origin than that of the page, because they are sandboxed without **allow-same-origin** .
In contrast, we noticed that in Gecko-based browsers (Mozilla Firefox), the CSP of the page applies to that of the srcdoc iframe if and only if allow-same-origin is present as value for the attribute. Otherwise it does not apply. The problem with this choice is the following. A third party script, whitelisted by the CSP of the page, can create a **srcdoc** iframe, sandboxing it with **allow-scripts** only, and load any resource that would normally be blocked by the CSP of the page if applied in this iframe. This way, the third party script successfully bypasses the restrictions of the CSP of the page. Even though loading additional scripts is considered harmless in the upcoming version 3 [267, 272] of CSP, this specification says nothing about violations that could occur due to the loading of other resources inside a **srcdoc** sandboxed iframe, like resources whitelisted by **object-src** directive for instance, additional iframes etc.
We have notified the W3C, and the Mozilla Security Group. Daniel Veditz, a lead at Mozilla Security Group, recognizes this as a bug and explains:

> "*Our internal model only inherits CSP into same-origin frames (because in theory you're otherwise leaking info across origin boundaries) and iframe sandbox creates a unique origin. Obviously we need to make an exception here (I think we manage to do the same thing for src=data: sandboxed frames).*"

CSP specification and srcdoc iframes The problem of imposing a CSP to an unknown page is illustrated by the following example [271]. If a trusted third party library, whitelisted by the CSP of the page, uses security libraries inside an isolated context (by sandboxing them in a **srcdoc** iframe, setting **allow-scripts** as sole value for the **sandbox**) then, the page's CSP will block the security libraries and possibly introduce new vulnerabilities. Because of this, it was unclear to us what the intent of CSP designers regarding srcdoc iframes was. Mike West, one of the CSP editors at the W3C and also Developer Advocate in Google Chrome's team, clarified this to us:

> "*I think your objection rests on the notion of the same-origin policy preventing the top-level document from reaching into it's sandboxed child. That seems accurate, but it neglects the bigger picture:* ***srcdoc*** *documents are produced entirely from the top-level document context. Since those kinds of documents are not delivered over the network, they don't have the opportunity to deliver headers which might configure their settings. We impose the parent's policy in these cases, because for all intents and purposes, the* ***srcdoc*** *document* `is` *the parent document.*"

8. https://en.wikipedia.org/wiki/WebKit
9. https://en.wikipedia.org/wiki/Blink_(web_engine)

6 Conclusion

In this work, we have revealed a new problem that can lead to violations of CSP. We have performed an in-depth analysis of the inconsistency that arises due to CSP and SOP and identified three cases when `CSP may be violated`.
To evaluate how often such violations happen, we performed a large-scale analysis of more than 1 million pages from 10,000 Alexa top sites. We found that 5.29% of sites contain pages with CSPs (as opposed to 2% of home pages in previous studies).
We also found out that 72% of current web pages with CSP, are potentially vulnerable to CSP violations. This concerns 379 (72.46%) sites that deploy CSP. Further analyzing the contexts in which those web pages are used, our results show that when a parent page includes an iframe from the same origin according to SOP, in 23.5% of cases their CSPs may be violated. And in the cases where `document.domain` is required in both parent and iframes, we identified that such violations may occur in 94% of the cases.
We discussed measures to avoid CSP violations in web applications by installing an origin-wide CSP and using sandboxed iframes. Finally, our study reveals an inconsistency in browsers implementation of CSP for `srcdoc` iframes, that appeared to be a bug in Mozilla Firefox browsers.

Chapter 4

DF-CSP: Dependency-Free Content Security Policy

Preamble

In this chapter, we analyze CSP versions and browsers implementations, formalize and propose rules and rules for building dependency-free policies (`DF-CSP`).
This chapter is currently under submission.

1 Introduction

From a web application developer's perspective, deploying a CSP effective at mitigating content injection attacks and preserving the full functionality of an application can become quickly challenging for many reasons.
First of all, a good understanding of the global meaning of CSP is important. For instance, in order to set restrictions on the origins of trusted scripts, one uses the `script-src` directive. Nonetheless, it is known that plugins, in particular Adobe Flash plugins, can execute scripts in web pages. Hence, if no restrictions are set on plugins (if plugins can load from any origin), then an attacker can inject a malicious plugin, from an origin not whitelisted by the `script-src` directive, and execute scripts in the page after the plugin loads, as it has been demonstrated by Weichselbaum et al. [267]. Also important is the case of the `sandbox` directive, which when used in a policy alters the semantics of many other directives. For instance, its mere use in a policy, automatically prevents plugins, even if the `object-src` directive is used to specify a set of trusted origins for plugins.

	New directives	Deprecated
CSP1	`connect-src`, `default-src`, `font-src`, `frame-src`, `img-src`, `media-src`, `object-src`, `script-src`, `style-src`, `sandbox`, `report-uri`	n/a
CSP2	`base-uri`, `child-src`, `form-action`, `frame-ancestors`, `plugin-types`	`frame-src`
CSP3	`disown-opener`, `manifest-src`, `report-to`, `worker-src`	`child-src`, `report-uri`

Table 4.1 – CSP directives by version

The meaning of a CSP policy also depends on the version of the standard that the browser implements. There are three versions of the CSP standard [261, 272, 275], with CSP3 [272]

being the most recent one (See Table 4.1 for a summary of the changes among versions). Each new version builds on anterior ones, with its set of changes, potentially backwards incompatible with anterior versions, but with the aim at improving the effectiveness of the specification and ease of adoption by web application developers. In particular, a major feature at the heart of CSP3 is the concept of trust propagation for easily loading dynamic scripts. This is achieved with the introduction of the new `'strict-dynamic'` keyword to be used with the `script-src` directive [267]. Its semantics is that, a script which is whitelisted with a nonce or a hash, is allowed to further load any additional scripts even though such scripts are not explicitly whitelisted in the policy. From a security perspective, the use of `'strict-dynamic'` can give attackers different power depending on the version of CSP considered. In fact, an attacker who compromises a trusted script, can load arbitrary script if the underlying browser supports CSP3, but not in CSP1 and CSP2-compliant browsers where the attacker is bound by the whitelisted origins [1].

Moreover, browser vendors follow the evolution of CSP specifications at their own pace. While some of them quickly take up the latest improvements to the specification, others do not simply have a support of it all, or have a limited support of it. To add to this complexity, CSP does not offer to developers the possibility to deliver different CSP policies according to the version of CSP that the client's browser implements. Rather, the policy which is deployed by the application will be interpreted by the browser according to which version of CSP it supports. This leads to different meanings for a single CSP, as a policy which is deployed will be interpreted by a browser according to its implementation of CSP.

Ultimately, web applications would have to maintain multiple policies, one per browser. Then when a web page is accessed, the user agent will be detected, in order to serve the appropriate CSP. Maintaining multiple policies is potentially error-prone, as one has to keep all of them updated, effective against attacks while preserving the full functionality of web applications. Moreover, correctly detecting the user browser is crucial, as not delivering the right policy may potentially break the application's normal functionality or fail at mitigating content injection attacks. Unfortunately, detecting the user browser is not trivial, as this information can be potentially controlled by an attacker. For instance, browser extensions, which are very popular among users, are able of modifying HTTP headers. Thus, they can present to web servers a user agent which has nothing to do with the effective browser of the user. As a matter of fact, we installed the `User-Agent Switcher for Chrome` [147] extension on a `Chrome/68.0.3440.75`. By changing the `User-Agent` to `Opera 12.14`, Facebook and Twitter stopped sending any CSP with their responses, while they sent a CSP in a normal setting. This leaves the application unprotected against content injection attacks, while the underlying browser is fully CSP-compliant.

In this work, we introduce the notion of dependency-free policies (`DF-CSP`), for web applications developers to write and deploy policies that preserve the full functionality of web applications and that are effective at mitigating attacks, irrespective of the browser in which the application runs and the version of CSP supported by the browser. By building and deploying a `DF-CSP`, a web application developer maintains only a single policy and always serves the same CSP to all browsers without relying on user agent detection, as it could be potentially controlled by an adversary.

We scrutinize the CSP standard to find all dependencies and perform tests in browsers to assess their implementation of the specification. We then formalize and refer to these as `dependencies`. We formally define the notion of dependency-free policies (`DF-CSP`) and propose a set of provable correct rewriting rules that can be used to resolve dependencies in order to build dependency-free policies. These rules are mostly meant for developers

1. The attacker cannot read nonces, but can control the URLs of parser-inserted scripts

willing to ensure that their policies will be similarly enforced in different browsers. Some of our rewriting rules, in particular those related to the `sandbox` directive that alters the semantics of many directives, can also be implemented by browser vendors to comply with the specification. As a matter of fact, we found no browsers correctly implementing the `sandbox` directive.

Dependency-free policies are also useful to reason about CSP policies with the formal semantics of Calzavara et al. [179] that calculates the global meaning of CSP policies by adding the meanings of each individual directive. In this formal semantics, the global semantics of a CSP policy follows from the semantics of individual directive values. This is however not correct unless the CSP does not present any dependencies, because the meaning of individual directives could be altered by other directives.

Finally, we discuss the security implications of the use of `'strict-dynamic'` in policies. In fact, the use of `'strict-dynamic'` in backwards compatible policies such as `DF-CSP`, can give attackers different attack power depending on the version of CSP considered, especially in CSP3 where an attacker could potentially inject arbitrary content in the application. We show that automatically generating a second policy out of a policy that makes use of `'strict-dynamic'`, successfully ensures that an attacker who compromises a script allowed by a `DF-CSP`, does not gain more power, irrespective of the browser in which the application executes.

To assess how many sites in the wild could potentially benefit from building `DF-CSP`, we collected an analyzed the CSPs of top 100k Alexa sites. The results show that thousands of these websites can benefit from our rewriting rules for building dependency-free policies, either because they maintain multiple policies, or because they (see Table 4.7 for the results) deploy CSP that exhibit any of the dependencies we have formalized. To help build `DF-CSP`, we propose a new tool to assist developers in (1) building effective policies based on the state of the art and (2) understand the global meaning of their CSP policies by means of semantics and directive dependencies.

In summary, towards a better understanding of the global meaning of CSP policies, we make the following contributions:

— we identify, define, and formalize directive dependencies. These are a set of directive which implicit relations and individual meanings can lead to policies being differently interpreted depending on the version of CSP and browser implementation under consideration. In doing so, we find problems in CSP formal semantics and browsers implementation of CSP.

— we propose and implement a rewriter (a set of rewriting rules) for building dependency-free policies (`DF-CSP`) whose semantics are independent of any particular CSP version or browser implementation. At the core of `DF-CSP` is the mitigation of attacks as well as the preservation of the full functionality of web applications.

— We discuss how to deploy backwards compatible policies such as `DF-CSP`, along with the`'strict-dynamic'` keyword, without giving attackers more power in case they compromise a trusted script. Automatically generating and deploying a second policy out of a `DF-CSP` that uses `'strict-dynamic'`, successfully prevents an attacker from gaining more power even in CSP3-compliant browsers.

— we collect and analyze CSPs of the top 100k Alexa sites, and find that CSP policies in the wild often deploy non dependency-free policies, giving a lower bound on the number of web applications which may benefit from our `DF-CSP`. In other words, either they serve the same CSP, in which case their policy is non `DF-CSP`, or they serve different CSP based on the `User-Agent` we sent, in which case they maintain different policies for different browsers, meaning that they do not deploy `DF-CSP`.

— we implement a new tool to assist developers in building effective policies. The tool is meant to assist developers in building DF-CSP, or refactoring their policies in order to make them DF-CSP. Since CSP is primarily meant to mitigate content injection attacks, we start with rules that focus on scripts execution, then builds around them in order to have a final CSP. Even though the refactored CSP could be different from the original one, the rewriting rules do not introduce new vulnerabilities, as we put the mitigation of malicious scripts execution at the core of these rules.

2 Context and problems

2.1 Directives and their values in different CSP versions

Directives in CSP versions
Table 4.1 present CSP directives and the version in which they have been initially introduced. To start with, CSP1 [261] introduced 11 directives. Each directive targets a specific type of content, and can thus be used to restrict the origins from which content of the particular type can load from. For instance, the `script-src` directive specifies trusted origins where scripts can be loaded from. The `default-src` is a directive used as a fallback for `*-src` directives (directives which names end with `-src`). When `default-src` is present in a policy, and any of the directives which fallback to it is not specified, then the missing directive implicitly inherit the restrictions (values) of `default-src`. The directive helps for instance to apply the same restrictions on many directives at a time, without explicitly specifying them. CSP2 [275] introduced some additional directives, in particular `child-src`, to replace `frame-src`. Starting from this version, it is possible to set restrictions on trusted origins for form submission (`form-action` directive), origins of other web applications allowed to embed another application as an iframe (`frame-ancestors`), origins of URLs that can be used as values for the `<base>` tag (`base-uri`). In this version, the `plugin-types` directive has also been introduced. The `plugin-types` directive expresses the `types` of plugins that the application trusts (PDF, Java applets, Adobe Flash plugins, etc.). The trusted origins for plugins themselves are specified with `object-src` directive. CSP3 [272], currently under development, introduces some changes w.r.t to CSP2. In particular, it splits the `child-src` directive into `frame-src` (for frames) and a new directive `worker-src` (for workers), then deprecates `child-src` itself. The `report-to` directive has also been introduced to replace `report-uri`. The `manifest-src` directive makes it possible to specify the trusted origins of web applications manifests.
Directive values The semantics of the directive values (trusted origins) has evolved between previous versions and CSP3. In CSP1 and CSP2, insecure HTTP origins allow content only from the exact origin. In CSP3, HTTP origin also allows content from its secure HTTPS counterpart.
Nonces and hashes have been introduced in CSP2 to allow the whitelisting of individual inline scripts and stylesheets, instead of using the `'unsafe-inline'` keyword which removes any protection against attacks. Nonces can also be used to whitelist individual URLs. In CSP2, a script is allowed to load if its origin is whitelisted in the policy, or if the script has a valid nonce or hash. A major feature at the heart of CSP3 is the concept of trust propagation to easily load dynamic scripts. This is achieved with the introduction of the new `'strict-dynamic'` keyword to be used with the `script-src` directive. With `'strict-dynamic'`, a script which is whitelisted with a nonce or a hash, is allowed to further load any additional scripts even though such scripts are not explicitly whitelisted in the policy. This has implications from a security perspective. If the following policy is

deployed

```
script-src 'nonce-abcdef' 'strict-dynamic' 'self' https://
    trusted.com; object-src 'none';
```

— Browsers supporting CSP3, will enforce `'nonce-abcdef' 'strict-dynamic'`. An attacker, who can control the URLs of dynamically injected scripts, can inject and execute arbitrary script in the application, including from any attacker-controlled origins. This is due to the use of `'strict-dynamic'`, which enables scripts to load any additional scripts they require.

— CSP2-compliant browsers will enforce `'nonce-abcdef' 'self'`. The attacker can only inject content from the page own origin (`'self'`) which is anyway already trusted since it is whitelisted in the CSP of the page. The attacker cannot inject arbitrary script, as in the case of CSP3. Note that if we consider an attacker, who is able to read the DOM, and therefore the nonces, then this attacker can also inject arbitrary scripts as in the case of CSP3.

— Finally, in CSP1-compliant browsers, `'self'` will be enforced. As in the case of CSP2, the attacker can only inject scripts from the page own origin (`'self'`). Therefore, he cannot inject arbitrary script as in the case of CSP3.

Therefore, an attacker who manages to compromise a trusted script gains different power in the content that he can further inject. As one can see, even though `'strict-dynamic'` eases CSP adoption by allowing to quickly load dynamic content, the attacker power in a nonce-based policy (CSP3) is unlimited, compared to origin-based policies (CSP2, CSP1) in which the attacker is bound by the explicit permissiveness of the policy.

2.2 Problems with browsers support

As mentioned in the introduction, browsers implementations can lead to different interpretations of CSP.

The `sandbox` directive is not well supported in browsers We found no browser correctly supporting the `sandbox` directive. We filed bugs to the vendors of all the browsers we tested (Chrome 66, Chromium 60, Firefox 59, Opera 52, Safari 9.1.3). The `sandbox` directive has been introduced in CSP1, to provide protected applications with the same restrictions as the `sandbox` attribute for iframes in the HTML specification [2]. Depending on the presence or absence of its related flags, the `sandbox` directive alters the semantics of many other directives. First of all, its mere presence in a CSP prevents plugins from loading (`object-src`, `plugin-types`). By default, scripts execution (`script-src`), forms submission (`form-action`) are also prevented, and the `'self'` keyword in directives does not allow content from the page own origin, because the `sandbox` directive creates a unique origin. Restrictions on scripts, forms and `'self'` can be relaxed if the `allow-scripts`, `allow-forms`, and `allow-same-origin` flags (values) of the `sandbox` directive are specified respectively.
All the browsers we have tested misimplemented the `sandbox` directive when it lacks the `allow-same-origin` flag. In this situation, `'self'` must not match the page own origin. In Firefox, Opera, Chromium, and Safari, `'self'` keyword in all directives would still match the page own origin. In Chrome, only `'self'` in `script-src` will still match the page own origin.

2. https://www.w3.org/TR/html50/embedded-content-0.html#attr-iframe-sandbox

Firefox does not support the `plugin-types` directive and other problems Firefox does not support the `plugin-types` directive, that was introduced in CSP2. On Firefox then, restrictions on the types of plugins will be ignored, therefore allowing all types of plugins from origins whitelisted by the `object-src` directive the policy. CSP3 is still a working draft, but Chrome, Opera and Firefox already implements some of its features, in particular, the `'strict-dynamic'` keyword. Nonetheless, we found and reported to Mozilla, that the `'strict-dynamic'` keyword is ignored when it is specified in `default-src`. Since `'strict-dynamic'` is meant for the `script-src` directive, so directly adding it to the `script-src`, or to `default-src` should result in the same effect, since `default-src` is a fallback directive for `script-src`. Finally, IE Explorer 10 only supports the `sandbox` directive, and not other CSP directives [21].

Flash plugins can load scripts Plugins extend browsers capabilities by allowing them to render content which are not traditional HTML documents. Well known plugins are Adobe Flash and Java Applets. It is well known that Flash plugins in particular can execute scripts in the context of web applications. Therefore, in browsers which allow Flash plugins, the restrictions set on scripts can be understood as those allowed by the `script-src` and `object-src` directives, in case Flash plugins are allowed to execute. This has been particularly demonstrated by Weichselbaum et al. [267]. However, while the authors suggest that CSPs must not allow plugins, we rather argue that, plugins can be allowed to load, even Flash plugins, as long as the `object-src` directive does not allow more origins than the `script-src` directive. In this case, even though Flash plugins can execute scripts, this is done from origins which are already allowed by the `script-src` directive.
The fact that plugins can execute scripts also has an impact on workers (`child-src`, `worker-src`) and connections (`connect-src`). In fact, connections and workers are all JavaScript APIs that require script execution to be enabled before they can load. Hence, in browsers not supporting Flash plugins, when (normal) scripts (`script-src`) cannot execute, then workers cannot load either, and connections cannot be made. However, as we have shown, if scripts are not allowed, while plugins are allowed, browsers supporting Flash can still execute scripts, and consequently load workers or make connections to origins that are whitelisted in CSP.

Scripts can load fonts Traditionally, fonts (`font-src`) are included in web applications via the `@font-face` property of stylesheets (`style-src`). Hence, if stylesheets cannot load (`style-src 'none'`), consequently fonts loading via stylesheets is not either allowed, even though the `font-src` directive whitelists origins for loading fonts from. However, the W3C is currently working on a API for allowing fonts to also be loaded via scripts (`script-src`, and `object-src` because of Flash plugins). Known as CSS Font Loading API or `FontFace` API [43], many browsers (Chrome, Firefox, Safari, Opera) already provide it for scripts to load fonts. Hence, fonts can be loaded with stylesheets in all browsers and also via scripts only in browsers supporting the `FontFace` API.

2.3 Goal: is my CSP effective?

CSP has 3 versions, and even when browsers support the same version, they provide different implementation of it. Moreover, CSP does not make it possible to deploy different CSPs and clearly state to each browser, which one it should enforce, according to its implementation and the version of CSP it supports. Rather, the policy which is deployed will

be interpreted by different browsers according to their implementation and the version of CSP. It is the responsibility of the developer to ensure that irrespective of the browser in which her application will run, and which version of CSP it implements, the CSP deployed will preserve the functionality of the application and effectively protect the application against content injection attacks. This is a particularly daunting task, with regards to the differences (and sometimes incompatibilities) in the semantics of CSP between CSP versions, the versions that browsers support, and how well they support it.
Our goal is to address the challenges in deploying backwards compatible and effective CSPs, considering the differences and incompatibilities of semantics between CSP versions, browsers supports and implementations. We propose tools to assist in building and enforcing such policies.
To address the semantics challenges, we introduce the concept of CSP directives dependencies, and dependency-free policies (`DF-CSP`). By directives dependencies, we formalize the differences in semantics, the relations and influences between directives and their values, considering the different versions of the specification, and browsers implementations. Then, we introduce a rewriter, and a set of rules for resolving dependencies. We prove that the rewriter successfully produces dependency-free policies. These are policies which semantics are preserved accross different CSP versions and browsers implementations. We also discuss the security of CSP and `DF-CSP`, especially in presence of `'strict-dynamic'` in the `script-src` directive. While the use of this keyword eases CSP adoption by allowing to load dynamic scripts, it also gives an attacker unlimited power in case he can control the URLs of dynamically injected scripts. We demonstrate that deploying 2 policies limits an attacker power, even in case of a compromise of a trusted script.

3 Directives dependencies

In this section, we identify and formalize the different dependencies between directives. We consider 3 scenarios. First of all, to be comprehensive, we consider all the 3 CSP versions and their current implementations in browsers as we know of them. In a second scenario, we consider only CSP2 and CSP3, which are widely supported by major browsers. Finally, we consider CSP2 and CSP3 according to their specifications only, without considering the different browsers implementations.

3.1 CSP core syntax

For formalization purposes, we provide a CSP core syntax that represent a core of all three CSP versions. For the sake of simplicity, we consider only `well-formed` policies, which is defined below. Dealing with `well-formed` helps us to abstract from the complexity of CSP syntax. Finally, we define a directive lookup operator which given a policy, and a directive name, returns its set of values (i.e trusted origins associated to `script-src` directive if specified in a policy).
We borrow some of the notation and terms from a formalization of the semantics of CSP2 provided by Calzavara et al. [179]. However, in many cases the definitions of the concepts differed because the scope of our work is different from theirs. While they studied the semantics of individual directive values, we are interested in the global meaning of CSP, and the dependencies between directives themselves. Understanding the global meaning of CSP and resolving directives dependencies is useful prior to accurately analyzing the formal semantics of CSP as done in [179].
The core CSP syntax is shown in Table 4.2. This syntax is sufficient to illustrate our

formalization.

Source expressions	se	::=	`https:`|`'self'`|...
Directive name	t	::=	`script-src`|`object-src`|...
Directive values	v	::=	$\{se_1, ..., se_n\}$|{`'none'`}
Directive	d	::=	$t\ v$
Policy	p	::=	$\overrightarrow{d}$

Table 4.2 – CSP Core Syntax

A CSP policy p is a set of directives $\overrightarrow{d}$. CSP directives are all predefined, and include for instance `script-src`, `object-src`, `img-src`. (See also Table 4.1). Directives values are a set of source expressions se_i, that depend on the type of directive. They include origins (e.g. `https://trusted.com, trusted.com, *.trusted.com`), schemes (e.g. `https:`), keywords (e.g. `'self'`, `'none'`, `allow-scripts`). The special value {`'none'`} means that the directive does not allow any content. The special directive `default-src` is a fallback for many other directives (`script-src`, `img-src`, `object-src`, `connect-src`, ...). In other words, when a directive that falls back to `default-src` is not specified in a policy, its values resolve to the `default-src` directive values. We assume that there is a set f_d of directives that fallback to `default-src`.

Well-formed policies

By well-formed policies, we mean policies which:

— contain only known directives, given in the specification. In reality, nothing prevents a policy from including unknown directives. They will be ignored by browsers when the policy is enforced. We also consider only directives that set restrictions on the origins where content can be loaded from. This explains why we do not consider the `disown-opener` directive of CSP3 for instance [272].

— directives values are only those allowed by the specification. In practice, directive values which are not known will be ignored by browsers when enforcing the policy. For the sake of simplicity, we do not model nonces and hashes in the formalization, as they can refer to content from an arbitrary origin. This is difficult to reason about, in particular it is impossible to statically compare the restrictions on 2 directives which uses nonces or hashes, without any information on the content they will be referring to at runtime.

— directives are not duplicated. The specification makes it possible to have a directive repeated multiple times within a single policy. In this case, browsers will consider only the first instance of the directive and its related values. Other occurrences will be ignored.

— policies are not a conjunction of multiple policies. The specification allows to specify more than one policy to be enforced. In practice, browsers will enforce both of them, but separately. In this case, content should be allowed by both policies, in order to load in an application.

— directives values are expanded to accommodate the modifications introduced by CSP3, in particular that insecure origins also match their secure counterparts. CSP3 enforcement allows secure counterparts of insecure origins. For instance, the origin `http://example.com` allows both content from `http://example.com` and `https://example.com`.

— directive values are not redundant. For instance, a directive with `*.example.com*.sub.example.com` as values, presents redundancies. This is because `*.sub.example.com` is already allowed by `*.example.com`.

We assume the existence of a function for building well-formed policies. In practice, we have provided an implementation of such a function (see Section 5).

Lookup operator

We define a lookup operator which, given a policy and a directive, returns the allowed values corresponding to that individual directive only.

Definition 1 (Lookup)**.** *Given a policy expressed with a list of directives* $\overrightarrow{d}$*; and* t *a directive name, the lookup operator* $\overrightarrow{d} \downarrow t$ *is defined as follows:*

$$\overrightarrow{d} \downarrow t = \begin{cases} v & \text{if } t\ v \in \overrightarrow{d} \\ v & \text{if } \texttt{default-src}\ v \in \overrightarrow{d} \ \wedge\ t \in f_d \\ \emptyset & \text{if } t \in \{\texttt{plugin-types}, \texttt{sandbox}\} \\ \{*\} & \text{otherwise} \end{cases}$$

When a directive is present, the lookup operator returns its set of values as defined in the policy. Otherwise, if the `default-src` directive is present in the policy, and the directive being looked up falls back to `default-src`, then it returns the `default-src` directive values. If none of the two cases is true, then no particular restrictions are set on the directive. We resolve the value of the missing directive to a special value depending on the type of directive:

— $\emptyset$ means that the directive is missing in the policy (possible cases are `plugin-types` or `sandbox` directive).

— $\{*\}$ is a special value that means any content is allowed. It is returned when the missing directive allows content of a certain type (scripts, images, stylesheets). These are basically directives that may fallback to `default-src` (if `default-src` is present in $\overrightarrow{d}$), in addition to `form-action`, `frame-ancestors`, and `base-uri`.

We distinguish `plugin-types` and `sandbox` from other directives first because the values of the directives `plugin-types` and `sandbox` are only composed with keywords such as `allow-same-origin`, `application/pdf`, while other directives values can also be origins. Additionally, the other directives can be present in a policy with {*} as a value to mean that they allow content of any type. In other words, to express that for instance images are allowed from any origin, one can either omit the `img-src` directive (and the `default-src` directive) from a policy or add the `img-src` directive (or the `default-src` directive) in the policy by setting its value to {*}.
However, the only way to allow any type of plugins is to omit the `plugin-types` directive because there is no special value like {*} to be set to the `plugin-types` directive in a policy in order to allow plugins of any type. Similarly, once the `sandbox` directive is used in a policy, there is no special value like {*} to relax all the restrictions of the directive. As a matter of fact, if `allow-scripts` renable scripts that the `sandbox` directive had prevent, there is no similar flag to renable plugins once the `sandbox` directive is used, apart from removing it from a policy.

(D_1) $\forall\, t \in \{\texttt{frame-ancestors}, \texttt{base-uri}, \texttt{manifest-src}\},\ v = \overrightarrow{d} \downarrow \mathrm{t} \Rightarrow v = \{*\}$
(D_2) $\texttt{sandbox}\ v \in \overrightarrow{d}\ \wedge\ v' = \overrightarrow{d} \downarrow \texttt{object-src} \Rightarrow v' = \{\texttt{'none'}\}$
(D_3) $\texttt{sandbox}\ v \in \overrightarrow{d}\ \wedge\ \texttt{allow-scripts} \notin v\ \wedge v' = \overrightarrow{d} \downarrow \texttt{script-src} \Rightarrow v' = \{\texttt{'none'}\}$
(D_4) $v' = \overrightarrow{d} \downarrow \texttt{form-action} \Rightarrow (\texttt{sandbox}\ v \in \overrightarrow{d}\ \wedge\ \texttt{allow-forms} \notin v \wedge v' = \{\texttt{'none'}\})$
$\vee\ (((\texttt{sandbox}\ v \in \overrightarrow{d}\ \wedge\ \texttt{allow-forms} \in v)\ \vee \texttt{sandbox}\ v \notin \overrightarrow{d}) \wedge v' = \{*\})$
(D_5) $\texttt{sandbox}\ v \in \overrightarrow{d} \wedge\ \texttt{allow-same-origin} \notin v \Rightarrow \forall t\ v' \in \overrightarrow{d}, \texttt{'self'} \notin v'$
(D_6) $v = \overrightarrow{d} \downarrow \texttt{frame-src}\ \wedge\ v' = \overrightarrow{d} \downarrow \texttt{child-src} \Rightarrow v = v'$
(D_7) $v = \overrightarrow{d} \downarrow \texttt{script-src}\ \wedge\ v' = \overrightarrow{d} \downarrow \texttt{child-src}\ \wedge v'' = \overrightarrow{d} \downarrow \texttt{worker-src} \Rightarrow v = v' \wedge v = v''$
(D_8) $\texttt{plugin-types}\ v \in \overrightarrow{d}\ \wedge\ v' = \overrightarrow{d} \downarrow \texttt{object-src} \Rightarrow v' = \{\texttt{'none'}\}$
(D_9) $v = \overrightarrow{d} \downarrow \texttt{object-src}\ \wedge\ v' = \overrightarrow{d} \downarrow \texttt{script-src} \Rightarrow v - v' = \{\}$
(D_{10}) $v = \overrightarrow{d} \downarrow \texttt{script-src}, v' = \overrightarrow{d} \downarrow \texttt{object-src}.\ v = \{\texttt{'none'}\} \Rightarrow v' = \{\texttt{'none'}\}$
$\vee\ (\forall\, t \in \{\texttt{connect-src}, \texttt{child-src}, \texttt{worker-src}\}, v'' = \overrightarrow{d} \downarrow \mathrm{t}\ \wedge\ v'' = \{\texttt{'none'}\})$
(D_{11}) $v = \overrightarrow{d} \downarrow \texttt{style-src}, v' = \overrightarrow{d} \downarrow \texttt{font-src}.\ v = \{\texttt{'none'}\} \Rightarrow v' = \{\texttt{'none'}\}$
$\vee\ (\forall t \in \{\texttt{script-src}, \texttt{object-src}\}, v'' = \overrightarrow{d} \downarrow \mathrm{t} \wedge v'' = \{\texttt{'none'}\})$

Table 4.3 – Formalization of Dependency-Free Policies (DF-CSP) considering CSP1, CSP2 and CSP3 versions and their implementations in browsers.

3.2 Formalization of DF-CSP considering CSP1, CSP2, CSP3 and browsers implementations

Table 4.3, which is explained in the following subsections, shows a formalization of sufficient conditions for CSP policies to be directive dependency free. Symbols $\in$, $\notin$, and $\Rightarrow$,$\wedge$, $\forall$ are standard and have the usual meanings of set inclusion operators and logical implication, conjunction, and quantification. Using the conditions of Table 4.3, we can define a dependency-free policy (DF-CSP). Intuitively, a policy is dependency-free when all the rules $D_1, ..., D_{11}$ hold.

Definition 2 (DF-CSP)**.** *A policy* $\overrightarrow{d}$ *is dependency-free (DF-CSP) iff* $\forall\, i \in \{1, ..., 11\}, D_i$ *holds.*

The table implicitly defines dependencies: there is a dependency between directives in a CSP policy $\overrightarrow{d}$ when a condition D_i does not hold. For example, D_9 states that there is a dependency between object-src and script-src if the object-src directive is more permissive than the script-src directive.

We classify the dependencies in 4 categories. Backwards incompatible directives are those which do not have an equivalent directive in other versions of the specification. For instance, form-action was introduced in CSP starting from CSP2, and is therefore not known in CSP1. We then provide the semantics of the sandbox directive as it should be implemented by browsers. Then, we turn to directives which in different CSP versions have different meanings, and content types which in different versions of the specification are governed by different directives. Finally, we discuss dependencies due to browsers implementations of CSP, APIs they provide, and their influence on the semantics of a policy.

Backwards incompatible directives

This concerns directives which are only known in a specific version of CSP specification, and do not have any equivalence in other versions. When these directives are present in a policy, they will be enforced only in the versions of CSP in which they are known.

In other versions, they will be ignored, leading to a different semantics. This includes, `frame-ancestors`, and `base-uri` which where introduced in CSP2, and `manifest-src` which is newly introduced in CSP3[3].
Ignoring a directive is equivalent to having it allow every content. So the values of directives `frame-ancestors`, `base-uri`, and `manifest-src` must always resolve to $\{*\}$ in order for a policy to be dependency-free. This is formalized as D_1. It is worth noting that the `manifest-src` directive is a special case. In fact, it falls back to `default-src`. Hence, even when it is not specified in a policy, then if the `default-src` directive is present, the value of `default-src` should be $\{*\}$, otherwise the policy is not `DF-CSP`.

```
frame-ancestors 'self' trusted.com;
```

Listing 4.1 – D_1 problem: in CSP1, the page can be embedded by `untrusted.com`, in spite of CSP

The policy in the example above contains a dependency because the directive `frame-ancestors` is not backwards compatible. The directive will be ignored in CSP1. Only CSP2 and CSP3 will correctly enforce it.

Semantics of the `sandbox` directive

When the `sandbox` directive is present in a policy, it automatically prevents plugins from loading, no matter the restrictions set on the plugins directive (`object-src`). This is formalized as D_2. Scripts execution (`script-src`) and forms submission (`form-action`) are also prevented when the `sandbox` directive is present, and does not include `allow-scripts` and `allow-forms` in its values set. These are formalized as D_3 and D_4 respectively. Finally, when the `sandbox` directive is present, and `allow-same-origin` is not in its values set, the `'self'` keyword present in other directives is ignored. We formalize this as D_5. It is worth mentioning the case of forms submission directive `form-action`. It has only been introduced in CSP2. As formalized by D_4, in a `DF-CSP`, form submission can be disallowed (by not including `allow-forms` in `sandbox`). Otherwise if the `sandbox` directive includes `allow-forms` or is not present, then the form submission directive (`form-action`) should be treated as the backwards-incompatible directives discussed in the previous paragraph: it must allow any origin ($\{*\}$), otherwise the policy is not `DF-CSP`. This is because the directive is not supported in CSP1. Below is an example of a policy with dependencies related to the `sandbox` directive.

```
sandbox allow-scripts allow-forms;
script-src 'self' trusted.com;
form-action 'self' trusted.com;
object-src 'self' trusted.com;
```

Listing 4.2 – D_2, D_3, D_4, D_5 problems: `sandbox` alters the semantics of other directives.

Disregarding the `sandbox` directive, this policy allows scripts, plugins and forms submission to the page own origin (`'self'`) and `trusted.com`. Now considering the policy as a whole, the presence of the `sandbox` directive no longer allows plugins. The absence of the `allow-same-origin` in `sandbox` values set, creates a unique origin. Unique origins are incomparable to any other origin [76]. Therefore, a unique origin does not match `'self'`, the origin of a webpage where a policy is enforced [261,272,275]. Therefore even though the `sandbox` directive allows scripts and forms (`allow-scripts` and `allow-forms`), requests to

3. Note that `plugin-types` and `form-action` directives are also backwards incompatible directives, but we discuss them in different categories of dependencies, because the former is not supported in Firefox, and the latter is related to the `sandbox` directive

load scripts or submit forms can only be made to `trusted.com`, and not to the page own origin, despite the presence of `'self'` in the values of these 2 directives. Many browsers we have tested would still allow content from the page own origin, some allow only scripts, and others allow any type of content (scripts, forms, etc.). Additionally, the `form-action` directive is not known in CSP1, and will be ignored in this version, as is the case for backwards incompatible directives (See Section 3.2), leading to differences in semantics.

Different directives, different meanings

When specific directives are used in different versions of CSP for a single content type, setting different restrictions on these directives results in different restrictions being enforced for the same content type, depending on the version of the specification under consideration. This is the case for frames (D_6) and workers (D_7). In CSP1 and CSP3, frames are specified with the `frame-src` directive, while in CSP2, the `child-src` is used instead. The directives `script-src`, `child-src`, and `worker-src` are used to specify restrictions on workers in CSP1, CSP2 and CSP3 respectively. D_6 checks that restrictions on frames directives (`frame-src`, `child-src`) are the same, and D_7 that the restrictions on workers directives (`script-src`, `child-src`, `worker-src`) are also the same[4]. Otherwise the policy is not `DF-CSP`. The following Listing 4.3 shows a non-`DF-CSP` due to frames and workers.

```
script-src trusted.com blob:;
frame-src 'self';
worker-src 'self';
```

Listing 4.3 – D_6, D_7 problems: in CSP2, workers and frames can be loaded from any domain, in spite of CSP

Workers are concerned with `script-src`, `child-src`, and `worker-src` directives in CSP1, CSP2 and CSP3 respectively. Workers can be loaded from `trusted.com` in CSP1, any domain in CSP2 (because `child-src` is missing), and the page own origin in CSP3. Frames are concerned with `frame-src` (CSP1, CSP3) and `child-src` (CSP2) directives. Therefore in CSP1 and CSP3, frames can load only from the page own origin, while in CSP2 they can load from any origin (because `child-src` directive is missing).

Browser specific APIs

We consider the following CSP in order to specify dependencies due to browser specific APIs.

```
script-src 'self';
object-src trusted.com;
style-src 'none';
plugin-types application/pdf;
```

Listing 4.4 – D_8, D_9, D_{10}, D_{11} problems: plugins can load scripts and scripts can load fonts, in spite of CSP

The `plugin-types` directive The `plugin-types` directive is not part of CSP1. Moreover, it is not supported in Firefox, which otherwise supports other directives of CSP2 and some features of CSP3. So Firefox and CSP1-compliant browsers will ignore this directive, meaning they will always allow any `type` of plugins to load from all origins specified in

4. In practice, only same origin or blob workers are allowed. Therefore, it would have been sufficient to ensure that the effective restrictions on workers (page origin and/or blob) are equal in the three directives

the plugins directive (object-src). According to the specification [272, 275], when the plugin-types directive is present in a policy, it further restricts plugins (object-src), by specifying precisely which types of plugins are allowed in an application (PDF, Java, Adobe Flash, etc.). Unfortunately, it is not possible to include plugin-types directive in a policy, to allow any type of plugin (there is not a default value which resolves to all types of plugins, as in the case for instance for {*}, which in a directive such as img-src, would allow images from all origins). Hence, when this directive is used in a policy, plugins (object-src) should not be allowed, otherwise the policy is not DF-CSP. This is expressed as D_8. Listing 4.4 shows an example of a non dependency-free policy because of D_8. According to this policy, in browsers such as Chrome, Opera, only PDF documents are allowed as plugins, while in Firefox and CSP1-compliant browsers, any type of plugin can load, because these browsers do not support the plugin-types directive.

Loading Scripts via Flash Plugins In CSP, the script-src directive normally restricts the origins of scripts, and object-src restricts the origins of plugins. However, Flash plugins can also execute scripts. So, scripts execution in an application concerns both script-src as well as object-src (when Flash plugins can execute). By default, Firefox does not allow the execution of Flash plugins. Users requiring Flash plugins have to manually install Adobe Flash Reader to do so. Thus, for users who do not install Flash in their browsers, scripts execution is limited to script-src directive only, while in other browsers, the Flash plugins allowed by the object-src, can also execute scripts. As long as object-src directive only allows origins which are already permitted by script-src directive, then, even though Flash plugins can load (allowing them to execute scripts), the policy is DF-CSP. This is because (normal) scripts are allowed to load from the same origins. This is formalized as D_9. The object-src directive should not allow origins which are not already allowed by the script-src (See Listing 4.4 for an example). This is different from the suggestion of Weichselbaum et al. [267] that one must always prevent plugins in a policy. We rather safely argue that, plugins can be allowed to load, even Flash plugins, as long as the object-src directive does not allow more origins than the script-src directive. In this case, even though Flash plugins can load scripts, this is done from origins that are already allowed by the script-src directive.

Workers and connections depends on scripts execution Workers and connections are JavaScript APIs [12, 128, 150], and can only load when scripts execution is enabled. In browsers where plugins cannot execute scripts, if normal scripts (script-src) cannot load, then workers cannot load either, no matter the origins whitelisted in their related directives. However, in browsers where plugins can also execute scripts, it is not sufficient that normal scripts execution is not allowed to consequently prevent workers or connections. If plugins are allowed, workers can also load and connections can be made, leading to a different semantics, depending on the browser under consideration. Therefore, when normal scripts execution is not enabled, one has to ensure that either plugins cannot load, or loading workers and making connections is explicitly not allowed. This is expressed as D_{10}, and Listing 4.4 shows an example.

Fonts depends on stylesheets and also scripts execution Traditionally, fonts were loaded via stylesheets (style-src). But with the introduction of the CSS Font Loading API [43], already supported by major browsers, fonts can also be loaded via script execution. So, in browsers not supporting the API, when stylesheets cannot be loaded, fonts cannot be loaded either. However, in browsers supporting the API, if scripts execution

(R_1) $\forall t \in \{$`frame-ancestors`, `base-uri`, `manifest-src`$\}$, if $\overrightarrow{d} \downarrow \mathrm{t} \neq \{*\}$ then $\overrightarrow{d} \downarrow \mathrm{t} = \{*\}$
(R_2) if `sandbox` $v \in \overrightarrow{d}$ then $\overrightarrow{d} \downarrow$ `object-src` $= \{$`'none'`$\}$
(R_3) if `sandbox` $v \in \overrightarrow{d}$ and `allow-scripts` $\notin v$, then $\overrightarrow{d} \downarrow$ `script-src` $= \{$`'none'`$\}$
(R_4) if `sandbox` $v \in \overrightarrow{d}$ and `allow-forms` $\notin v$, then $\overrightarrow{d} \downarrow$ `form-action` $= \{$`'none'`$\}$ else $\overrightarrow{d} \downarrow$ `form-action` $= \{*\}$
(R_5) if `sandbox` $v \in \overrightarrow{d}$ and `allow-same-origin` $\notin v$, then $\forall t\ v \in \overrightarrow{d}, \overrightarrow{d} \downarrow \mathrm{t} = v - \{$`'self'`$\}$
(R_6) $T = \{$`frame-src`, `child-src`$\}, t_1, t_2 \in T, v_1 = \overrightarrow{d} \downarrow t_1$ and $v_2 = \overrightarrow{d} \downarrow t_2$. $\forall t \in T, \overrightarrow{d} \downarrow \mathrm{t} = v_1 \cup v_2$
(R_7) $T = \{$`script-src`, `child-src`, `worker-src`$\}, t_1, t_2, t_3 \in T$, $v_1 = \overrightarrow{d} \downarrow t_1$ and $v_2 = \overrightarrow{d} \downarrow t_2$ and $v_3 = \overrightarrow{d} \downarrow t_3$. $\forall t \in T, \overrightarrow{d} \downarrow \mathrm{t} = v_1 \cup v_2 \cup v_3$
(R_8) if `plugin-types` $v \in \overrightarrow{d}$ then $\overrightarrow{d} \downarrow$ `plugin-types` $= \emptyset$ or $\overrightarrow{d} \downarrow$ `object-src` $= \{$`'none'`$\}$
(R_9) $o =$ `object-src`, $s =$ `script-src`, $T = \{o, s\}, v_1 = \overrightarrow{d} \downarrow \mathrm{o}$ and $v_2 = \overrightarrow{d} \downarrow \mathrm{s}$. $\overrightarrow{d} \downarrow \mathrm{o} = \{$`'none'`$\}$ or $\overrightarrow{d} \downarrow \mathrm{o} = v_1 \cap v_2$ or $\forall t \in T, \overrightarrow{d} \downarrow \mathrm{t} = v_1 \cup v_2$
(R_{10}) $v = \overrightarrow{d} \downarrow$ `script-src`. if $v = \{$`'none'`$\}$ then $\overrightarrow{d} \downarrow$ `object-src` $= \{$`'none'`$\}$ or $\forall t \in \{$`connect-src`, `child-src`, `worker-src`$\}, \overrightarrow{d} \downarrow \mathrm{t} = \{$`'none'`$\}$
(R_{11})$v = \overrightarrow{d} \downarrow$ `style-src`. if $v = \{$`'none'`$\}$ then $\overrightarrow{d} \downarrow$ `font-src` $= \{$`'none'`$\}$ or $\forall t \in \{$`script-src`, `object-src`$\}, \overrightarrow{d} \downarrow \mathrm{t} = \{$`'none'`$\}$

Table 4.4 – Rewriting Rules

is enabled, fonts can also be loaded even though stylesheets cannot be loaded. This is expressed with D_{11}. Therefore, when stylesheets cannot be loaded, one has to ensure that fonts cannot be loaded either (i.e fonts or scripts execution are explicitly not allowed by the CSP policy). An example is given in Listing 4.4.

3.3 Rewriter for building `DF-CSP` for CSP1, CSP2, and CSP3

The goal of the rewriter is to transform a CSP policy into a `DF-CSP`. The rules are presented in Table 4.4. Each rewriting rule R_i resolves a related dependency D_i of the same number. Intuitively, each rewriting rule R_i applies a set of guidelines and modifications to a policy, in order to make the condition D_i hold. Now, we prove that each rewriting rule effectively resolves the related dependency.
The rules are meant for developers willing to build or refactor policies to be equally enforced in different browsers.
(R_1) Applying R_1 results in removing any restrictions on directives `frame-ancestors`, `base-uri`, and `manifest-src`. For `frame-ancestors` and `base-uri` directives, one can simply remove them from the policy. If the `default-src`, is not specified, one can also remove the `manifest-src` directive from the policy. Otherwise, one has to explicitly add it to the policy, setting its values to {*}. This basically means that no restrictions are set on the directive. R_1 ensures that no restrictions are set on directives `frame-ancestors`, `base-uri`, and `manifest-src`, thereby making D_1 to hold.
(R_2) When the `sandbox` directive is present in a policy, R_2 sets the `object-src` directives values to {`'none'`}. This is exactly the semantics of the `sandbox` directive regarding plugins as stated in HTML5 standard [70]. This makes D_2 holds.
(R_3) When the `sandbox` directive is present, while not specifying `allow-scripts` in its values, then scripts execution is not allowed. R_3 changes the `script-src` directive to {`'none'`}. This makes D_3 hold.
(R_4) Similarly, when the `sandbox` directive is present, while not specifying `allow-forms`,

then forms submission is not allowed. R_4 sets the `form-action` directive to {`'none'`}. Otherwise, forms submission must be allowed to any origin, because the directive is ignored in CSP1. In this case R_4 sets the `form-action` directive values to {*}. Either of these rewriting make D_4 to hold.
(R_5) The absence of the `allow-same-origin` in the `sandbox` directive results in a mismatch between `'self'` and a page own origin. This is is equivalent to not having the `'self'` keyword in any directive. To do so, R_5 removes this keyword from any directive values where it is found, making D_5 to hold.
(R_6) `frame-src`, and `child-src` directives both concern frames inclusion. R_6 rewrites them so that they have the same restrictions. Either both directives are set to the values of one of them, or to the union of the values of both of them. In any case, the result is that they will have the same values, which makes D_6 hold.
(R_7) Similarly, `script-src`, `child-src`, and `worker-src` directives concern workers. R_7 rewrites these directives so that they have the same values. All directives can either be assigned the values of one of them, or the union of the values of 2 of them, or the union of the values of all the 3 directives. In any case, these rewriting make D_7 hold.
(R_8) Since `plugin-types` directive is not supported in all browsers, when it is present in a policy, R_8 either removes it ($\overrightarrow{d} \downarrow \texttt{plugin-types} = \emptyset$ means that the `plugin-types` is not present (removed) from the policy (See Section 3.1 for more details), or it sets the `object-src` directive to {`'none'`}. Either of the rewriting makes D_8 hold. As with the case of directives not being backwards compatible, it is recommended not to use `plugin-types` in `DF-CSP`, as it leads to different semantics.
(R_9) Plugins directive `object-src` should not be more permissive than `script-src` directive. To so do, R_9 either sets `object-src` directive to {`'none'`} or to the intersection of `object-src` and `script-src`. Otherwise it assigns both directives the union of their respective values. All these rewriting ensures that `object-src` is not more permissive than `script-src`.
(R_{10}) When scripts are not allowed and if connections, workers and plugins are allowed, then connections can be made and workers can load in browsers allowing script execution via plugins, while in others, this will not be the case. So, when scripts are not allowed, R_{10} rewrites the policy so as to prevent plugins, or connections and workers. This makes D_{10} hold.
(R_{11}) When stylesheets are not allowed while fonts are still allowed, if scripts execution is allowed (either via `script-src` or `object-src`), then fonts can still load via scripts execution in browsers supporting the CSS Font Loading API [43]. To make D_{11} hold, R_{11} rewrites the policy so as to prevent fonts from loading, or to prevent scripts execution.

3.4 Resolving all Dependencies

Applying the rules in any order leads to a policy not being `DF-CSP`, because of conflicts between them. Conflicts arise between 2 rules when both of them modify the same directive. Below are the directives whose modifications introduce conflicts.

`object-src`	$R_2, R_8, R_9, R_{10}, R_{11}$
`script-src`	$R_3, R_7, R_9, R_{10}, R_{11}$
`child-src`	R_6, R_7, R_{10}
`worker-src`	R_7, R_{10}

A rule of thumb when applying these rules is that a directive value should not change twice. Considering `child-src`, if R_6 modifies its value to v, while R_7 modifies it to another value v' different from v, then R_7 reintroduces in the policy the dependency D_6 that R_6 has just resolved. When a directive is modified, then it is better to fix its value, and no longer alter

it in subsequent rewriting rules, in order to avoid entering into infinite loops, and not being able to make the policy `DF-CSP`.
Since content injection attacks in web applications is done through scripts execution, then we propose the following order in applying the rewriting rules.

— The first 5 rules (R_1, R_2, R_3, R_4, R_5) can be applied in any order. They are independent from one another.

— Resolve the dependency between plugins and plugin types directives, by applying R_8.

— Then apply rules related to scripts execution. R_9, R_{10}, R_{11}. Then fix the values

— Then apply R_7, for workers, then R_6 for frames.

The rule of thumb applies here. When a directive is modified, then fix its value and no longer alter it in subsequent rules.
One of the rewriting options we propose in rules R_6, R_7, R_9 is to assign the set of directives under consideration (`frame-src` and `child-src` in the case of R_6), a new set of values, computed as the union of the values of all the directives or a subset of them. We recommend to always compute the union of all of the directives in the set, instead of selecting only a subset of them. Even though the resulting CSP may be more permissive, at least it preserves the semantics of the policy, and does not break the application.

Correctness

Dependencies D_2, D_3, D_4, D_5 are related to the `sandbox` directive. The rewriting rules R_2, R_3, R_4, R_5 strictly follow from the specification [261, 272, 275] and the semantics of `sandbox` [76]. Applying these rules does not modify the semantics of a policy and is compliant with CSP specification. Since most browsers do not properly support the `sandbox` directive, implementing our rewriting rules can help them comply with the specification regarding this directive and its influence on other directives. Developers can also apply these rules to their policies prior to deploying them, in order to ensure that the `sandbox` directive will be correctly enforced in all browsers, with respect to its influence on other directives.
The remaining rules are meant as guidelines for developers willing to build policies which are dependency-free, and with equivalent semantics in all browsers. Dependency D_1 concerns directives which are not backwards compatible with the different CSP versions. In other words, they do not have equivalence in all versions of the specification, meaning they will be ignored in such versions. These directives should not set any restrictions on the type of content they are related to, as suggested in R_1. Basically, a policy should not set restrictions on forms, application manifests, etc.
R_6 ensures that restrictions on frames are the same in all versions of CSP. R_7 concerns workers. R_8 relates to the fact that `plugin-types` directive is not supported in all versions of the specification. The best solution is not to use this directive at all in a policy. R_9 ensures that restrictions set on plugins are not permissive than those on scripts, since (Flash) plugins could execute additional scripts in some browsers, and not in others. R_{10} ensures that when normal scripts execution is not allowed, plugins cannot load either. Otherwise, in some browsers, plugins could still make connections, load workers, while in other browsers, this would not be the case. Finally, R_{11} ensures that when stylesheets cannot load, fonts cannot load either. Otherwise, if scripts execution is enabled, fonts can still load in some browsers, while in others this would not be the case.

Example of dependency-free policies

To summarize, a dependency-free policy is a policy which

- does not set any restriction on, `frame-ancestors`, `base-uri`, `plugin-types`, `manifest-src`.
- either prevents forms submission or allow forms submission to any origin.
- applies the same restrictions on frames directives (`frame-src`, `child-src`) and workers directives (`script-src`, `child-src`, and `worker-src`).
- when stylesheets are not allowed, fonts should not also be allowed. Otherwise scripts can load fonts.
- `object-src` directive should be less or as permissive as the `script-src` directive. Otherwise, plugins can execute additional scripts in browsers supporting Flash.

Below are examples of `DF-CSP` policies.

```
default-src trusted.com;
manifest-src *;
```

Listing 4.5 – Policies should not set any restriction on manifests, and other backwards incompatible directives

```
script-src 'self' trusted.com;
child-src 'self' trusted.com;
frame-src 'self' trusted.com;
worker-src 'self' trusted.com;
object-src trusted.com;
```

Listing 4.6 – Scripts, frames, and workers should have the same restrictions. Plugins directive should be less permissive than the scripts directive

```
sandbox allow-forms;
object-src 'none';
script-src 'none';
child-src 'none';
worker-src 'none';
frame-src 'none';
form-action *;
```

Listing 4.7 – Sandbox prevents plugins and scripts execution. So, frames and workers should also be prevented. `'self'` should not be used in directives values. Forms can be submitted to any origin

```
style-src 'none';
font-src 'none'
```

Listing 4.8 – Policies should explicitly prevent fonts when stylesheets are not allowed

3.5 Dependencies between CSP2 and CSP3 implementations

We want to assess how the dependencies and rewriting rules change if we consider only CSP2 and CSP3. In fact, CSP2 has a lot in common with CSP3, and we are not aware of any modern browser fully supporting CSP1. A scenario in which only CSP2 and CSP3 are considered is more realistic, and representative of current CSP implementations by browsers in the wild [21].
The changes are the following w.r.t dependencies and rewriting rules for the versions (CSP1, CSP2, CSP3) presented in Table 4.5.

$(D_1) v = \overrightarrow{d} \downarrow \texttt{manifest-src} \Rightarrow v = \{*\}$
(D_4) `sandbox` $v \in \overrightarrow{d} \ \wedge$ `allow-forms` $\notin v \ \wedge \ v' = \overrightarrow{d} \downarrow \texttt{form-action} \Rightarrow v' = \{\texttt{'none'}\}$
$(D_7) \ v = \overrightarrow{d} \downarrow \texttt{child-src} \ \wedge v' = \overrightarrow{d} \downarrow \texttt{worker-src} \Rightarrow v = v'$

(R_1) if $\overrightarrow{d} \downarrow \texttt{manifest-src} \neq \{*\}$ then $\overrightarrow{d} \downarrow \texttt{manifest-src} = \{*\}$
(R_4) if `sandbox` $v \in \overrightarrow{d}$ and `allow-forms` $\notin v$, then $\overrightarrow{d} \downarrow \texttt{form-action} = \{\texttt{'none'}\}$
$(R_7) \ T = \{\texttt{child-src}, \texttt{worker-src}\}, t_1, t_2 \in T, v_1 = \overrightarrow{d} \downarrow t_1$ and
$v_2 = \overrightarrow{d} \downarrow t_2. \ \forall t \in T, \overrightarrow{d} \downarrow \mathrm{t} = v_1 \cup v_2$

Table 4.5 – Dependencies and rewriting rules considering only CSP2 and CSP3 and their implementations in browsers

— Only the `manifest-src` directive of CSP3 is backwards incompatible, since it does not have an equivalence in CSP2. So, D_1 and R_1 are changed to contain only this directive.

— Since `form-action` is part of CSP2 and CSP3, therefore D_4 and R_4 are modified accordingly. Specifically, the `form-action` directive is set to {`'none'`} if the `sandbox` directive is present and does not include the `allow-forms` value.

— `script-src` directive is no longer linked to workers. The directives related to workers are `child-src`, and `worker-src` in CSP2 and CSP3 respectively. Therefore, `script-src` is removed from D_7 and R_7.

— Other dependencies (See Table 4.3) and rules 4.4 remain unchanged.

From the examples of `DF-CSP` presented in Section 3.4, only the policy shown in Listing 4.7 changes as show in the example below.

```
sandbox allow-forms;
object-src 'none';
script-src 'none';
child-src 'none';
worker-src 'none';
frame-src 'none';
form-action 'self' trusted.com;
```

The `form-action` directive is no longer limited to $\{*\}$ when the `sandbox` directive is specified with `allow-forms`. It can take any set of trusted origins, while still making the policy `DF-CSP`.

3.6 Dependencies between CSP2 and CSP3 specifications

In previous sections, we focused on the differences in CSP specifications as well as the peculiarities of their implementations in browsers. For instance, the dependencies and rewriting rules take into consideration the fact that Firefox does not support the `plugin-types` directive. Here, we consider only CSP2 and CSP3, as they are described in the specification, and discuss the dependencies and rewriting rules in this scope only. Table 4.6 presents the modifications that this implies in the original dependencies and rewriting rules presented in Tables 4.3 and 4.4

— The changes in dependencies D_1, D_4, D_7, and the related rewriting rules R_1, R_4, R_7 are the same as described in Table 4.5 for the case where CSP2 and CSP3 are considered as well as their implementations in browsers.

(D_8) $v = \overrightarrow{d} \downarrow$ plugin-types, $v' = \overrightarrow{d} \downarrow$ object-src. $v = \{$'none'$\} \Rightarrow v' = \{$'none'$\}$
(D_9) $v = \overrightarrow{d} \downarrow$ object-src $\wedge v' = \overrightarrow{d} \downarrow$ script-src $\Rightarrow v \subseteq v'$
$\vee$(plugin-types $v'' \in \overrightarrow{d} \wedge$ application/x-shockwave-flash $\notin v''$)
(D_{10}) $v = \overrightarrow{d} \downarrow$ object-src $\wedge v' = \overrightarrow{d} \downarrow$ script-src $\wedge$ $v' = \{$'none'$\} \Rightarrow v = \{$'none'$\}$
$\vee$ (plugin-types $v'' \in \overrightarrow{d} \wedge$ application/x-shockwave-flash $\notin v''$)
$\vee$ ($\forall t \in \{$connect-src, worker-src, child-src$\}, v''' = \overrightarrow{d} \downarrow$ t $\wedge$ $v''' = \{$'none'$\}$)
(D_{11}) $v_1 = \overrightarrow{d} \downarrow$ style-src, $v_2 = \overrightarrow{d} \downarrow$ font-src, $v_3 = \overrightarrow{d} \downarrow$ script-src, $v_4 = \overrightarrow{d} \downarrow$ object-src.
$v_1 = \{$'none'$\} \Rightarrow v_2 = \{$'none'$\} \vee$ ($v_3 = \{$'none'$\}$
$\wedge$ ($v_4 = \{$'none'$\} \vee$ (plugin-types $v \in \overrightarrow{d} \wedge$ application/x-shockwave-flash $\notin v$)))

(R_8) $v = \overrightarrow{d} \downarrow$ plugin-types, $v' = \overrightarrow{d} \downarrow$ object-src. if $v = \{$'none'$\}$ $\wedge$ $v' \neq \{$'none'$\}$ then
($\exists v'' \neq \{$'none'$\} \wedge \overrightarrow{d} \downarrow$ plugin-types $= v''$) $\vee \overrightarrow{d} \downarrow$ object-src $= \{$'none'$\}$
(R_9) $v = \overrightarrow{d} \downarrow$ script-src, $v' = \overrightarrow{d} \downarrow$ object-src, $v'' = \overrightarrow{d} \downarrow$ plugin-types. if $v \subset v'$ then
$\overrightarrow{d} \downarrow$ plugin-types $= v'' - \{$application/x-shockwave-flash$\}$
$\vee \overrightarrow{d} \downarrow$ object-src $= \{$'none'$\} \vee \overrightarrow{d} \downarrow$ object-src $= v \cap v'$
$\vee \forall t \in \{$script-src, object-src$\}, v_1, v_2 \in \{v, v'\}, \overrightarrow{d} \downarrow$ t $= v_1 \cup v_2$
(R_{10}) $v = \overrightarrow{d} \downarrow$ script-src, $v' = \overrightarrow{d} \downarrow$ object-src, $v'' = \overrightarrow{d} \downarrow$ plugin-types.
if $v = \{$'none'$\}$ then $\overrightarrow{d} \downarrow$ object-src $= \{$'none'$\}$
$\vee \overrightarrow{d} \downarrow$ plugin-types $= v'' - \{$application/x-shockwave-flash$\}$
$\vee \forall t \in \{$child-src, worker-src, connect-src$\}, \overrightarrow{d} \downarrow$ t $= \{$'none'$\}$
(R_{11}) $v = \overrightarrow{d} \downarrow$ script-src, $v' = \overrightarrow{d} \downarrow$ object-src, $v'' = \overrightarrow{d} \downarrow$ plugin-types.
if $\overrightarrow{d} \downarrow$ style-src $= \{$'none'$\}$ then $\overrightarrow{d} \downarrow$ font-src $= \{$'none'$\}$
$\vee$ (($\overrightarrow{d} \downarrow$ plugin-types $= v'' - \{$application/x-shockwave-flash$\}$
$\vee \overrightarrow{d} \downarrow$ object-src $= \{$'none'$\}$) $\wedge \overrightarrow{d} \downarrow$ script-src $= \{$'none'$\}$)

Table 4.6 – Dependencies and rewriting rules for CSP2 and CSP3, according to the specifications. We consider only browsers which implementations are compliant with the specifications

— According to the specification, when no specific `type` of plugins is allowed, then plugins themselves are not allowed. Normally, the specification requires that the plugin-types directive always specify at least one value when it is included in a policy [275]. But nothing prevents one from adding no values to this directive, making it {'none'}. When it is the case, then plugins (object-src) are not allowed to execute. D_8 and R_8 are modified accordingly.

— Regarding scripts execution via plugins, since only Flash plugins can execute scripts, the plugin-types directive does not include a dependency, unless it explicitly allows Flash plugins (application/x-shockwave-flash), or it is absent (which means that any type of plugins is allowed). This consequently changes dependencies D_9, D_{10}, D_{11} and rewriting rules R_9, R_{10}, R_{11} that describe the fact that plugins can execute scripts, and therefore make connections, load workers or fonts.

Other directives are unchanged.
Below are examples of DF-CSP in CSP2 and CSP3 specifications, assuming that they are correctly implemented by browsers.

Table 4.7 – Dependencies in the wild, considering CSP1, CSP2, CSP3 and their implementations in browsers.

Dependency	**#Pages**	**#Origins**	**#Sites**
D_1	134,339 (**60.61%**)	6,282	3,814
D_2	23	5	5
D_3	43	1	1
D_4	49	2	2
D_5	43	1	1
D_6	48,068 (**21.69%**)	1,726	1,302
D_7	71,161 (**32.11%**)	2,888	2,076
D_8	206	8	8
D_9	26,739 (**12.08%**)	905	695
D_{10}	0	0	0
D_{11}	0	0	0

```
script-src 'self';
object-src *;
plugin-types application/pdf;
```

This policy allows scripts from the page own origin, and plugins from any origin. Nonetheless, only PDF plugins are allowed, and not Flash which can execute scripts. This policy is `DF-CSP`, according to CSP2 and CSP3 specifications, even though the `object-src` directive is more permissive than the `script-src` directive.

4 Dependencies in the wild

We collected and analyzed the CSP of pages from top 100k Alexa sites, in order to assess the prevalence of dependencies (Table 4.3). To collect CSP, we used SlimerJS [130] on `Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:57.0) Gecko/20100101 Firefox/57.0`. We visited the homepages, and also links that we found on the homepage, pointing to the site or its subdomains.

To analyze policies, we implemented a tool with a full CSP parser according to the specification [275] (for checking whether a directive allows a URL to load). It can compare the permissiveness of 2 policies, and detect when one CSP directive allows more origins than the other [255]. It can also detect dependencies as discussed in this work.

We found 221,638 pages deploying CSP. They are spread over 18,673 origins from 13,226 sites out of 100k Alexa sites (13.22%). Policies with dependencies (for the scenario where all CSP versions and browsers implementations are considered) are presented in Table 4.7. As one can observe from Table 4.7, the most prevalent reason why policies are not `DF-CSP` is because of D_1, which relates to the use of backwards-incompatible directives in policies. Recall that the mere presence of these directives in a policy introduces dependencies, when they do not have their equivalence in other versions of the specification. Hence, 60.61% of the CSP deployed use directives (`frame-ancestors`, `base-uri`, `manifest-src`). They are not known to all versions of CSP. This result also include policies which set to the `form-action` directive, values which are not {`'none'`} nor {$*$}. Since this directive is not part of CSP1, restrictions set on it get ignored, leading to dependencies.

Another dependency widely present in policies is D_7, which concerns workers. It has 3 different directives, one per CSP version (`script-src`, `child-src`, `worker-src`). Hence, 32.11% of the policies we have analyzed do not set the same restrictions on the workers

directives.
As the results show for D_6, 21.69% of the policies deployed set different restrictions on the frames directives (`frame-src`, `child-src`).
We found 12.08% of pages allowing scripts executions via (flash) plugins (`object-src`), from origins not whitelisted in the `script-src` directive.
Some 206 pages use the `plugin-types` directive, which results in different enforcement because the directive is not supported in CSP1 and Firefox. The results show that the `sandbox` (D_2, D_3, D_4, D_5) is not widespread among policies. In particular, a single site is making use of `sandbox` directive, without `allow-same-origin`, but with `'self'` in its CSP directives values (D_5). Finally, we found no policy in which scripts (`script-src`) are not allowed, and allow workers and connections via plugins (D_{10}). Nor did we find policies which do not allow stylesheets, and allow fonts to be loaded via scripts (D_{11}).

4.1 Validity of the statistics

A criticism that could be directed to the results presented here would be about their validity, because we collected the policies using a specific browser. In particular, if indeed a web server checks the `User-Agent` string to send a specific CSP, then for all the requests, it is clear that we have obtained a policy for `Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:57.0) Gecko/20100101 Firefox/57.0`. However, this exactly means that the related server is not maintaining a single policy, but rather multiple policies, one per browser. This implies that the policy we obtained from the server is not `DF-CSP`. Otherwise, if the policy sent by the server is the same for any browser, then it is not `DF-CSP`, according to our analysis. Therefore, all the results reported here are valid, even though the server served a specific CSP based on the `User-Agent` string we sent during the crawling process.

5 Tool for building effective policies

The main goal of the tool is to assist developers in building effective policies. It is able to detect common errors reported by Calzavara et al. [177], resolve dependencies, reduce policies by removing semantic redundancies in directive values, and provide the semantics of the policy. The tool also checks that policies are `well-formed` as specified in Section 3.1. To do so, it removes unknown directives and their values; removes unknown directive values keeping only those allowed by the specification; removes duplication of directives keeping only the first occurrence; expands policies by explicitly adding directives which fallback to `default-src`, by explicitly adding them in the policy, when the directive is missing and `default-src` is present. Building a single policy from a conjunction of multiple policies is achieved by computing the intersection of the set of whitelisted origins of each directive in both policies. Check out the tool here [5].
All of its features are described below for the dependencies and rewriting rules in Tables 4.3 and 4.4 for all the versions of CSP.

Errors and misconfigurations This includes misspelled directive names or values (i.e `defalt-src` instead of `default-src`), quoting (using double quotes instead of single one for `'self'`, or not quoting values when they should be quoted), and missing colon after a scheme name (`https` instead of `https:`) [177]. For all these cases, the tool suggests changes to the developer. To make a suggestion, we compute the Levenshtein distance [6]

5. https://swexts.000webhostapp.com/dependencies/
6. https://en.wikipedia.org/wiki/Levenshtein_distance

between known directive names and values and misspelled ones. Then the suggestions are shown to the developer who can change the policy accordingly. The list of directive values that should be quoted is given by CSP specification (`'self'`, `'none'`, nonces, hashes, `'strict-dynamic'`). Other values (origins, schemes, and `sandbox` directive flags) should not be quoted. Only single quotes are accepted in the specification. The tool also ensures that schemes always end with a colon (https:), otherwise suggestions are made to the developer to fix them.

Dependencies Directives dependencies are described in Section 3. The tool detects and proposes to resolve them. For instance, when scripts are specified, while the `sandbox` directive does not include the `allow-scripts` flag, this is detected as a dependency issue, and it is suggested to the developer to either allow scripts execution by adding the `allow-scripts` flag to the `sandbox` directive, or disallow scripts execution by setting the `script-src` directive values to {`'none'`}. Other dependencies are treated similarly.
It is worth mentioning the ability of executing scripts with Flash plugins. So, whenever `object-src` directive can be used to execute scripts from additional origins not included in the `script-src` directive (using Flash), the tool would suggest to either remove the additional origins from the `object-src` directive or simply set its value to `'none'` [267], or remove from it the additional origins not whitelisted in `script-src` directive, etc (See Table 4.4 for more details).

Semantics The tool generates the semantics of the policy, which is all the origins from which content can be loaded for each type of content according to the CSP. In particular, the `default-src` directive is expanded by adding directives which fallback to it when they are not clearly specified. Any other missing directive (which does not fallback to `default-src`) is explicitly added, as allowing any content of the related type to load.

Redundancies The tool is also able to detect origins redundancies in the policy, and suggest to developers to remove them, in order to improve the clarity and maintainability of the policy. For instance, the origin `*.example.com` covers `www.example.com`. When those 2 origins are whitelisted, the tool suggests to only keep the first one.

6 `DF-CSP` and strict CSP

In this section, we discuss the use of the keyword `'strict-dynamic'` in backwards compatible policies such as `DF-CSP`. To improve the protection of an application deploying CSP in backwards compatible fashion, we propose that web applications always accompany a nonce-based policy that makes use of `'strict-dynamic'`, with an origin-based policy to further limits an attacker's power. This policy is not a new policy, but one which is automatically generated from a policy that makes use of `'strict-dynamic'`. The second policy is exactly the same as the first one, except that it does not include the `'strict-dynamic'` keyword.
One can still benefit from `'strict-dynamic'`, by enabling whitelisted scripts to further load all their dependencies (additional scripts), without having to set individual nonces or hashes to these dependencies. When the policy is enforced in a CSP2 or CSP1-compliant browser, all the dependencies can still load thanks to the origins (CSP1) and/or nonces (CSP2). Regardless of the version of CSP supported by the browser, an attacker who manages to compromise a trusted script, cannot load more scripts than what is declared in the origin-based policy. In fact, browsers will enforce both policies. A content is allowed

to load if it is allowed by both policies. Since the origin-based policy clearly states the origins from which trusted scripts can load, the attacker can then only load content from these origins, which are anyway already trusted.

6.1 Attacker model

The attacker here is able to control the URLs of non-parser scripts dynamically injected from a trusted script [267, 272]. Below is an example of a dynamic script execution.

```
script = document.createElement("script");
script.src = "http://attacker.com/x.js"
document.body.appendChild(script)
```

We consider that the attacker can control the value of the `src` attribute of the dynamically injected script. For instance, this URL is retrieved from a database, or as a result of any XSS attack.

6.2 Design

To deploy multiple CSPs, one simply separate them with commas. Hence, the single policy in Listing 2.1 is rewritten in two policies as follows:

```
script-src 'strict-dynamic' 'nonce-abcdef' 'self' https://
    trusted.com; object-src 'none';,
script-src 'nonce-abcdef' 'self' https://trusted.com; object-src
    'none';
```

The first policy is exactly the same as the one in Listing 2.1. The second one is also exactly the same except that it does not have the keyword `'strict-dynamic'` in its `script-src` directive. The second policy is automatically generated from the first one, which when deployed together successfully provide the same protection against attacks. Let's stress again that the second policy is automatically generated from the first one, as we will demonstrate below. This is the sole difference between the two policies. Regarding only the `script-src` directive, in CSP3, the 2 policies `'strict-dynamic' 'nonce-abcdef' https://trusted.com` and `'nonce-abcdef' 'self' https://trusted.com` will be enforced. Even if an attacker manages to compromise a script, he cannot inject arbitrary content, because of the second script, which binds the origins of trusted script to only `'nonce-abcdef', 'self' and https://trusted.com`. This is exactly the same protection provided in CSP1 and CSP2 when the two policies are both enforced. This design helps preserve the protection of the application across browsers supporting different versions of CSP, and more importantly, helps prevent attackers from injecting arbitrary content when they get to compromise a trusted script, in particular in CSP3-compliant browsers.

6.3 Applications vulnerable to such attacks

We performed an analysis of the CSP policies from top 100k Alexa sites, to assess how websites using `'strict-dynamic'` are protected against such attacks. Our results show that `'strict-dynamic'` is not very widespread. We found the use of this keyword in the CSPs of pages from 24 different origins. Among those, only `https://www.dropbox.com` and `https://earn.com` deployed a policy set consisting of 2 policies. All other origins were vulnerable to this attack. It is worth noting that among those sites, 7 of them deployed very liberal policies, by allowing `https:, http:` schemes in the `script-src` directive [177]. This already allows attackers to inject arbitrary content in CSP1 or CSP2. This implies

that the attacker already gains the same power in all versions of the specification for these websites.

7 Conclusion

Following the success of CSP1 and CSP2, the W3C is currently actively working on the next version of the specification, CSP3. Each version builds on the previous one, with changes aimed at improving the security and ease of adoption by developers. In this work, we have highlighted and addressed the semantics and security challenges related to the changes introduced in each version of the specification, in particular when a single policy has to be enforced in different browsers providing implementations which are not always compliant with the specification. We formalize the differences between the specifications as dependencies, and propose a set of rewriting rules for building dependency-free policies (`DF-CSP`). To the best of our knowledge, this work is the first comprehensive study of CSP dependencies and the first to propose a tool for resolving these dependencies in order to obtain effective CSP policies.

Chapter 5

Extending CSP: Blacklisting, URL arguments Filtering and Monitoring

Preamble

This chapter presents proposals for extending the CSP specification to address different limitations of CSP that have been demonstrated in the literature or identified by us. In particular, we propose and implement a blacklisting mode to CSP, a mechanism for filtering URLs parameters, prevent redirections and an efficient reporting mechanism for collecting feedback about the runtime enforcement of CSP.
This chapter has been submitted for review.

1 Introduction

Previous studies have demonstrated the limitations of CSP as a whitelisting mechanism, and the attacks that can be mounted to bypass CSP [212, 267]. To illustrate these limitations, we pose the following research questions regarding important security requirements for an application.

How do we effectively whitelist a precise set of trusted content from a domain ? In this scenario, a developer does not trust a whole third party domain, but only a specific content, or set of content. CSP makes it possible to express this by partially whitelisting an origin. In other words, instead of declaring the whole (third party) origin in a policy, one declares only the specific trusted content or set of content from the domain. As such, the origin is partially whitelisted, and only the whitelisted content are allowed to load. Nonetheless, by using HTTP redirections, any content from the partially whitelisted origin can be loaded [272, 275]. In particular, when the CSP whitelists origins that host insecure open redirects endpoints, the CSP can be bypassed by loading any content from the partially whitelisted origin, as if the origin was fully whitelisted [267].

How do we exclude an untrusted content from a whitelisted origin ? In this second scenario, the developer needs to whitelist an origin but exclude a specific content or set of content that it hosts. For instance, the content to exclude is an untrusted content known for introducing a threat in the application. This is the case of the popular AngularJS JavaScript library [8]. Weichselbaum et al. [267] reported that it allows to execute arbitrary scripts in a webpage, despite CSP. Another case we found interesting is a webpage that loads

scripts from its own origin and also from a third party domain. However, the developer may want to prevent the third party from loading scripts located at the developer's website under the path `/admin/` because it contains sensitive scripts. The developer may also want to prevent the third party script from discovering that there is a user logged into the current web application. In fact, in a recent study, Gulyás et al. [198] demonstrated that by loading a resource such as an image which is only accessible once a user is logged into a website, a third party script can discover that the user is logged into the website and use such information for tracking purposes for instance. Unfortunately, CSP does not allow to blacklist a specific content or set of content of an origin. When an origin is declared in a policy, it is considered trusted in its entirety.

How do we filter out unsafe URL parameters ? In this scenario, we consider that an origin is whitelisted, but one would like to ensure that URLs injected in the webpage do not have parameters. That is, when an origin hosts content which are insecure JSONP endpoints, URL parameters provided to requests to load such content can be leveraged by an attacker to execute arbitrary content [212, 267]. Bypassing partially whitelisted origins by HTTP redirections is also done by leveraging parameters of open redirects [267]. Unfortunately, URL parameters are ignored by browsers when they match a URL against a policy. Additionally, to exfiltrate user data, attackers usually pass them to URLs as parameters of HTTP requests to load content. Hence, even if a CSP tries to prevent data exfiltration by restricting the endpoints to which AJAX requests can be made to [275], attackers can still exfiltrate data by passing them as parameters of URLs of other types of content, considered less security critical, such as images for instance.

How do we efficiently collect feedback about the runtime enforcement of CSP in a webpage? A developer would like to know which content are allowed by the CSP deployed to protect webpages of her application. This feedback can be useful for many reasons. First, even if an application has been heavily tested, it is not excluded that an attacker can find a vulnerability and inject malicious content in the application. Moreover, an error in a CSP may result in the policy being more permissive than expected, allowing attacker-injected content to load [177]. Furthermore, browser extensions are widespread on major browsers [24,58,94,108]. In particular, they have the ability to intercept and modify CSPs deployed to protect webpages, and inject in webpages their own content that is not always required to comply with the CSP of the page [26, 200]. Nonetheless, extensions content may further inject vulnerabilities in webpages, which are otherwise restricted by CSP. Finally, the new `'strict-dynamic'` keyword introduced in CSP3 [267,272] allows to potentially load any script in a webpage at runtime. It makes it impossible to statically know the content (scripts) that are allowed by a policy before it is effectively enforced in a browser. Hence, knowing which content is effectively injected in a webpage at runtime represents a valuable information that can help assess the security of a webpage and deploy more secure policies.
Weichselbaum et al. [267] proposed the use of nonces to mitigate CSP bypasses based on open redirects and unsafe JSONP endpoints. However, this solution comes with the following issues. First, the security of nonces is questionable because they are included in the DOM of webpages [178,272,275]. Moreover, the use of nonces does not prevent a script that is already loaded in the webpage from making requests with JSONP parameters, or from redirecting to partially whitelisted origins, especially if the script gets compromised. Finally, nonces apply only to scripts and stylesheets, and not to other types of content such as images whose URLs parameters an attacker can leverage to exfiltrate user data for

instance. CSP provides a reporting mechanism for developers to collect violations which occur in browsers during its enforcement [261,272,275]. Violations are triggered by content not matching the CSP of the page. Ultimately, deploying the most restrictive policy (a policy that does not allow any content to load) in report-only mode could potentially give feedback about content that load in an application. This method however is inefficient and incomplete. In fact, a CSP in report-only mode implies that any content in the webpage will trigger a violation and browsers will submit a report for each individual violation. If the webpage loads numerous content, then the reports sent for each content from the browsers of all the users of the application potentially introduce an overhead from the application server-side. Moreover, if a violation is triggered, is it because of a trusted content or an untrusted content ? To distinguish between trusted and untrusted content, one would have to deploy at least 2 policies, one in report-only mode, the other one in enforcement mode, collect violations in both cases, and compute the difference to get the content effectively allowed by the policy. Furthermore, content injected by browsers extensions that are not subject to the CSP of the page, will not trigger any violation report. Collecting feedback by using CSP violations is therefore incomplete. Finally, the use of `'strict-dynamic'` in CSP3 makes it difficult to know in advance the origins from which content in a webpage will be effectively loaded, before the CSP is enforced.
To fully address the aforementioned issues, we propose to extend the current CSP specification.
Adding a blacklisting mode to CSP Currently CSP specification defines 2 modes: the `report-only` mode in which policies are enforced, but browsers do not block content not allowed by the policy; and the `enforcement` mode in which content that are not allowed by the policy are effectively blocked. We refer to these two modes as CSP whitelisting modes. The proposed blacklisting mode is the exact opposite of the enforcement mode: content that match a CSP in blacklisting mode are blocked, otherwise they are allowed. We propose to introduce a new header, `Content-Security-Policy-Blacklisting` for deploying CSP in blacklisting mode. One would use this mode to exclude specific content or set of content on a domain from loading in a webpage. CSP in blacklisting mode proves useful when one knows that the domain hosts content that are potentially malicious and could introduce further vulnerabilities if loaded in a webpage. This new mode can also serve to explicitly prevent the loading of sensitive content in a webpage, as they may reveal information about a logged-in user for instance [198]. The blacklisting mode is meant to be used as a complement to CSP deployed in either of the whitelisting modes.
Filtering URLs parameters We propose extending the URL matching algorithm of CSP [275], which is used to check whether a URL is allowed by a policy or not, in order to take into consideration URL parameters. In the current status of the specification, URL parameters are considered trusted by default. The proposed extension is to enable declaring origins, paths, and specific content by specifying the URL arguments that are trusted or untrusted. This extension is meant to be used with the new CSP blacklisting mode. If an entire content can already be blacklisted, by filtering URL parameters, one can further blacklist content when they are injected in webpages with URLs that have specific unsafe parameter names, parameters with unsafe values, or even prevent URLs with any parameters. The URL filtering mechanism perfectly fits requirements where one wants to ban URLs arguments of requests to insecure JSONP endpoints, open redirects endpoints, or prevent data exfiltration via URL parameters.
Disallowing redirections to partially whitelisted origins The CSP bypass due to partially whitelisted origins can already be mitigated with the 2 previous proposals. In fact, if one identifies the open redirects endpoints of origins whitelisted in a policy, they

can either be blacklisted or their URL parameters filtered in order to prevent them from redirecting to partially whitelisted origins. Nonetheless, one has to ensure that all open redirects endpoints are identified and blacklisted. Otherwise, it is sufficient that an endpoint be missed to leave the whole CSP bypassable. We propose to introduce a new directive `disallow-redirects` that can be used in policies to instruct the browser to prevent all redirections to partially whitelisted origins. In the current CSP URL matching algorithm [275], browsers would allow loading any content from a partially whitelisted origin if the content is the result of an HTTP redirection [272,275]. This new directive is meant to alter this precise part of the algorithm: when it is used in a policy, browsers should prevent from loading, any content not explicitly whitelisted on a partial origin. This ensures that once an origin is partially whitelisted, browsers strictly enforce it.

Efficient feedback reporting mechanism Finally, we propose extending CSP with an efficient reporting mechanism for content that match a policy, similarly to CSP violations reports. While a browser enforces a policy, it can keep track of all content matching the policy, and report this information to an endpoint specified in the CSP of the page. To specify the endpoints for collecting feedback, we introduce the directives `monitor-uri` and `monitor-to`, similar to the `report-uri` and `report-to` directives currently used for collecting violations. To make the mechanism efficient, reports may be sent after all content are loaded and the page enters a stable state. Content injected thereafter could be submitted at regular intervals according to a delay defined by the browser. The web application developer can analyze this feedback, and look for potentially malicious content that loaded because of errors or misconfigurations in the policy, content injected by browser extensions, or dynamic content loaded by scripts when the `'strict-dynamic'` keyword is used in a CSP. Therefore, the policy can be updated to improve its effectiveness.

As we have shown, these extensions improve on previous proposals for fighting against CSP bypasses. Nonces that have been proposed to mitigate JSONP and open redirects have been criticized in the literature, mostly because they are included in the DOM of web applications, and attackers can use scriptless attacks to read nonces and inject arbitrary content [178]. Filtering URLs parameters can be applied to URLs to prevent data exfiltration, JSONP, and open redirects by mandating that URLs of requests cannot carry any arguments, or specific arguments. Using CSP violations reporting mechanism to get a feedback, is inefficient and incomplete, it introduces an overhead, and requires the deployment of 2 policies.

To further show that the proposed extensions require few changes from a browser perspective, we implemented them using service workers in an example web application. Service workers (we refer to them as a monitor) intercept HTTP requests initiated by browsers to load content in a webpage. As such, they act like a proxy for content included in the page [149]. We deploy a service worker with an example web application, which deploys a CSP in enforcement mode and another policy in blacklisting mode. The CSP in blacklisting mode is enforced by the monitor, and the CSP in whitelisting mode is enforced by the browser. Once the URL of a content matches a policy, the browser makes a request to fetch its content. Then, the request is sent to the service worker, which further checks its URL against the blacklisting policy. If the URL matches the blacklisting policy (either because it is a blacklisted content or carries untrusted arguments), then the request is blocked, otherwise it is effectively made. For open redirects, as HTTP redirections are not intercepted by service workers for security reasons [275], we could not fully implement the new `disallow-redirects` directive. Nevertheless, to prevent redirections to partially whitelisted origins, we assumed that all open redirects are known by the developer. Then we either used the new blacklisting mechanism to blacklist the open redirects, or prevented

them from carrying URL parameters by filtering them out with the new URL parameters filtering mechanism we introduced. Finally, the monitor also logged all the URLs of content that it intercepted and reported them to the developer, as a feedback of the runtime enforcement of CSP.
In summary, this study contributes with four new extensions to the Content Security Policy. It aims at improving the security of web applications, by (i) expressing policies in blacklisting mode, (ii) filtering URL arguments, (iii) disallowing redirects to partially whitelisted origins and finally (iv) providing developers with an efficient way for collecting feedback about the runtime enforcement of their policies in an application.

2 Problem and motivation

For the sake of simplicity and throughout the rest of this work, we describe in more details the issues we addressed by considering mostly the `script-src` directive, which sets restrictions on the origins from which trusted scripts can load. Nonetheless this work is more general, and concerns CSP as a whole, its other directives and content types. To illustrate the limitations of CSP and motivate our proposals, we consider the following policies.

```
script-src https://trusted.com https://redirect.com https://
   partials.com/scripts/;
img-src trusted.com/image.png;
```

Listing 5.1 – Example of an origin-based CSP

```
default-src 'none'; form-action 'none';
frame-ancestors 'none'; report-uri /allcontent
```

Listing 5.2 – Restrictive policy in report-only mode to get the list of content loaded in a webpage

```
script-src 'nonce-random1234' 'strict-dynamic'
```

Listing 5.3 – Example of CSP with nonces, or nonce-based policy

Listing 5.1 presents a CSP where trusted scripts are whitelisted by their origins. We refer to them as origin-based policies. Only scripts from the explicitly specified origins are allowed to load in the webpage on which this policy will be deployed. The injection of a script with the URL `https://trusted.com/script.js` in the webpage is allowed since the script comes from the whitelisted origin `https://trusted.com`. Listing 5.2 presents a restrictive policy that basically prevents a page from loading content. Listing 5.3 presents a nonce-based policy. When a nonce-based policy makes use of the `'strict-dynamic'` keyword, we refer to the overall policy as a strict CSP. Nonces are used to whitelist individual scripts. To allow a script to load, one injects `<script src="https://trusted.com/script.js" nonce="random1234"></script>` in the page. Note the use of the `nonce` attribute on the script tag. Its value is the nonce whitelisted in the policy (See Listing 5.3). With the presence of the `'strict-dynamic'` keyword, whitelisted scripts (with nonces) that load in the page can further dynamically inject additional scripts, even though the additional scripts are not assigned whitelisted nonces [1]. Hence, contrary to the origin-based CSP in Listing 5.1 where one knows the exact origins from which content can load, in the case of strict CSP in Listing 5.3, scripts that effectively load are known only at runtime (when the page is loaded in a browser and the policy enforced).

1. Nonces and hashes work quite similarly [272, 275]

2.1 Partially whitelisted origins

In the CSP of Listing 5.1, from the domain https://partials.com, only scripts with the path /scripts/, for instance https://partials.com/scripts/a.js are trusted. Hence, trying to inject https://partials.com/script.js will fail. Nonetheless, one can get this script loaded and executed if it is loaded as the result of an HTTP redirection [272, 275]. To illustrate the bypass of partially whitelisted origins, let's assume that the origin https://redirect.com, which is also whitelisted in the CSP of Listing 5.1, hosts an open redirect endpoint https://redirect.com/r. In other words, instead of directly injecting https://partials.com/script.js, one passes the URL of the script as an argument to the open redirect endpoint by injecting a script with the URL https://redirect.com/r?url=https://partials.com/script.js. Instead of returning a script to be executed, the open redirect generates an HTTP redirection Location: https://partials.com/script.js. Since this is an HTTP redirection, the browser will not check whether the whole URL matches the CSP. It is sufficient that the origin of the request be fully or partially whitelisted in the CSP of the page for the script to be allowed via the HTTP redirection. And since this is the case (See Listing 5.1), then the script https://partials.com/script.js is allowed to load, even though its URL does not match the CSP. This bypass works in CSP2 [275] and CSP3 [272].

2.2 Excluding content from whitelisted origins

Now let's assume that from the CSP of Listing 5.1, most of the scripts from the https://trusted.com origin are trusted. However, the origin also hosts the insecure script https://trusted.com/untrusted.js and hosts sensitive scripts in the /admin/ folder (scripts which paths start with https://trusted.com/admin/) that must be not be loaded in the current page. CSP does not provide a mechanism for excluding content from an origin. When an origin is whitelisted, it is trusted in its entirety.

2.3 URL parameters

URL parameters are not taken into consideration when browsers match a URL against a policy. Consider the CSP of Listing 5.1, the URLs https://trusted.com/script.js and https://trusted.com/script.js?func=eval&arg=1 all match the CSP if they are used to inject a script, and the URL https://trusted.com/image.png?data=someuserdata&cookie=usercookies matches the policy it is the URL of an image injected in the page. In the first case, the URL does not have any parameter. In the second case, the same URL is provided the parameters func with the value eval and arg with the value 1. If according to CSP, these 2 URLs are exactly the same, in practice they may result in the execution of completely different content. Considering the second case, if the parameters provided are used to generate the response which is returned back, we run into JSONP requests which can lead to CSP bypass [212, 267]. In the third case, the URL to load the image is passed some user data and cookies as parameters, so that they are exfiltrated to trusted.com.

2.4 CSP violations

CSP can be used in 2 modes. In the report-only mode, policies are delivered to browsers using the Content-Security-Policy-Report-Only header. In this mode, content not matching the policy are not blocked by the browser. They are simply reported as CSP violations to the developer. In the dual enforcement mode on the other hand, policies are delivered to browsers using the Content-Security-Policy header. When browsers

enforce such a policy, content that do not match the policy are effectively blocked, and a violation report is also sent. CSP allows to combine policies in different modes, or even deploy multiple policies in the same mode. Multiple policies are all enforced individually. In this case, a resource is allowed to load if it is allowed by all the policies. The directives `report-uri` (in CSP1, CSP2) and `report-to` (in CSP3) are used in a policy to indicate where CSP violations will be submitted to [272, 275].
The violations report mechanism of CSP can be used to build the list of content that load in a webpage. To do so, one has to deploy 2 policies: a policy in enforcement mode (as the ones in Listing 5.1 and 5.3), and a policy in report-only mode that does not allow any content, as the policy shown in Listing 5.2. Since the policy is in report only mode, any content that attempts to load in the webpage will trigger a CSP violation. Hence, every content triggers a violation. It is therefore impossible to distinguish between malicious and trusted content by analyzing the violations reported by a single policy in report-only mode. So one has to also collect violations triggered by the enforcement of the policy in enforcement mode deployed to effectively prevent malicious content from loading (i.e. policies shown in Listing 5.1 and Listing 5.3). Hence violations in this policy are triggered only by content not matching the policy. Computing the difference between the 2 reports then gives the content that effectively loaded because they are allowed by the CSP of the webpage. It is worth mentioning the case of browser extensions, whose content are not always subject to the CSP of the page [26, 200]. For instance, if the policy in Listing 5.1 is deployed on a webpage, this does not prevent a browser extension from injecting a script with the URL `https://untrusted.com/vulnerable.js`, even if this URL is not allowed by the policy. Worryingly, the browser extension may be injecting a content that introduces vulnerabilities in the webpage. Moreover, the injection of this script will not trigger a CSP violation report, even in presence of a CSP in report-only mode as the one in Listing 5.2.

2.5 Motivation

To help mitigate the CSP bypasses due to JSONP and open redirects, Weichselbaum et al. [267] suggested the use of nonces for whitelisting individual scripts instead of whitelisting the origins, URLs or path to the scripts. Nevertheless, recent studies question the security of nonces, mostly because nonces are included in the DOM of webpages, and thereby are subject to leakage by scriptless attacks [178]. Moreover, the use of nonces does not prevent a script which is already loaded in a webpage from making requests with unsafe JSONP parameters, using open redirects, or loading untrusted content. If a whitelisted script gets compromised by an attacker, then he can bypass the CSP at will. The use of nonces apply only to scripts (`script-src`) and stylesheets (`style-src`) content types, and not to other types of content. The violation reporting mechanism is more indicated for violations that occur from time to time, and is not suited for efficiently collecting feedback about content that load in a webpage. Moreover, it does not report content injected by browser extensions. Also when the page deploys a strict CSP, collecting feedback is useful because one cannot know in advance the content allowed by the strict CSP before it is enforced.
To successfully address the aforementioned issues, we propose to extend the CSP specification with (i) a blacklisting mode, (ii) a URL arguments filtering mechanism, (iii) new directives for preventing redirections to partially whitelisted origins and (iv) a mechanism for efficiently collecting feedback about the runtime enforcement of the policy of a webpage by browsers.

3 Extending CSP specification

In this section, we introduce the extensions we propose to the CSP specification.

3.1 CSP in blacklisting mode

Similarly to the `Content-Security-Policy` and `Content-Security-Policy-Report-Only` headers used for deploying a CSP in enforcement and report-only modes respectively, we propose a new header `Content-Security-Policy-Blacklisting` for deploying CSP in blacklisting mode. Semantically, the blacklisting mode is the exact opposite of the enforcement mode. Hence, when a URL matches a CSP in blacklisting mode, then it is not allowed to load. Consider the policy in Listing 5.4.

```
script-src cdn.cloudfare.com/angular.js;
```

Listing 5.4 – A CSP in blacklisting mode to exclude `angular.js`

In enforcement mode, this policy would have allowed the `angular.js` script to load. By deploying the policy in blacklisting mode, then the script is blacklisted and hence not allowed to load. One can therefore combine this policy with another policy in enforcement mode to prevent only `angular.js` from loading, while allowing any other content from `cdn.cloudfare.com` to load. Listing 5.5 presents the two policies.

```
Content-Security-Policy: script-src cnd.cloudfare.com
Content-Security-Policy-Blacklisting: script-src cdn.cloudfare.
    com/angular.js
```

Listing 5.5 – Combining 2 policies: one in enforcement mode, and the other one in blacklisting mode

3.2 Checks on URL arguments

To illustrate this proposal, let's consider the following scenario. Listing 5.6 shows an example of a JSONP endpoint which expects the parameter `callback` and uses it to generate a function call, and passes it data.

```
Content-Security-Policy: script-src jsonp.com
```

Listing 5.6 – CSP with insecure JSONP endpoint

If an attacker injects `http://jsonp.com/?callback=eval` in a webpage, the returned response is a function call to `eval(...)`. Note that the URL argument is used to generate the function call. To prevent the URL from loading when it is passed the `callback` parameter, one could also deliver a CSP in blacklisting mode as shown in Listing 5.7 in addition to the CSP in Listing 5.6 that allows parameters to be passed to the insecure JSONP endpoint.

```
Content-Security-Policy-Blacklisting: script-src jsonp.com/?
    callback
```

Listing 5.7 – Supporting URL parameters in CSP

The CSP in blacklisting mode mandates that, for URLs to load content from `jsonp.com`, they must not have the argument `callback`. While the first policy only (Listing 5.6) would have allowed `http://jsonp.com/?callback=eval` to load, deploying also the second policy in blacklisting mode (Listing 5.7) would block it. By enforcing the CSP in blacklisting mode, one detects that the URL of the resource to load has an argument, whose name is `callback`. Therefore the URL is blocked. Note that this design does not prevent the

webpage from loading other content from jsonp.com. For instance, it is completely possible to load http://jsonp.com/script.js, but not to load http://jsonp.com/script.js?callback=foo. If the script has some other arguments, then they are allowed to load. For instance, loading http://jsonp.com/script.js?foo=bar is allowed by the policy above. We have shown how to prevent URLs with a specific argument (callback in our example). Now we illustrate additional scenarios on how to filter URLs parameters.

Blocking all URLs with arguments To prevent URLs with arguments, one simply ends their origins (paths or URLs) with ? in a blacklisting CSP.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src jsonp.com/?
```

Listing 5.8 – Blocking all URLs with arguments

The policy in Listing 5.8 stipulates that arguments are not allowed on URLs of scripts from jsonp.com. Without the CSP in blacklisting mode, any URL from jsonp.com would have been allowed. Now, the second policy in blacklisting mode will block URLs with parameters.

Blacklisting URLs when they have specific argument value One can go even more fine-grained, by blocking URLs only when they have specific parameters that have specific values. Listing 5.9 shows how to blacklist URLs from jsonp.com when the have the argument callback with the value eval.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src jsonp.com/?
    callback=eval;
```

Listing 5.9 – Blacklisting URLs with specific argument names and values

While the browser would prevent http://jsonp.com/script.js?callback=eval from loading, it would not prevent http://jsonp.com/script.js?callback=alert from loading.

Specifying multiple unsafe arguments The different scenarios presented above can be combined to filter out URLs with a set of unsafe arguments. If multiple arguments are specified for an origin in a blacklisting policy, then a URL is blocked if it has all the blacklisted arguments. Listing 5.10 is a policy which blocks URLs having both the callback and arg arguments with any values.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src jsonp.com/?
    callback&arg;
```

Listing 5.10 – Blacklisting URLs with multiple unsafe arguments

This policy will block http://jsonp.com/script.js?callback=eval&arg=1, but not http://jsonp.com/script.js?callback=eval, because the first URL has both unsafe parameters, while the second one does not.

Blocking URLs with at least one untrusted argument To block URLs if they have at least one argument among a set of unsafe arguments, one can declare the blacklisting policy as shown in Listing 5.11.

```
Content-Security-Policy: script-src jsonp.com
Content-Security-Policy-Blacklisting: script-src jsonp.com/?
    callback jsonp.com/?arg;
```

Listing 5.11 – Blacklisting URLs with at least one of multiple unsafe arguments

A URL will be blocked if it has either the argument callback or the arg with any values. Hence, http://jsonp.com/script.js?callback=eval and http://jsonp.com/

script.js?arg=1 will be blocked, while http://jsonp.com/script.js?foo=bar will not be blocked.

3.3 Preventing redirections

In CSP1, when an origin is partially whitelisted, then only content that match the partially whitelisted origin can load. In CSP2 and CSP3 however, any content from partially whitelisted origins are allowed to load as the result of HTTP redirections [272, 275]. Here, we propose to extend the CSP specification so as to allow developers to explicitly prevent redirections to partially whitelisted origins. To do so, we propose a new directive `disallow-redirects`. When this directive is present in a policy, it prevents redirections to partially whitelisted origins. Consider the following policy

```
script-src https://partials.com/scripts/.js;
disallow-redirects;
```

Listing 5.12 – CSP that uses `disallow-redirects` to explicitly prevent redirections to partially whitelisted origins

In case of an HTTP redirection (using an open redirect endpoint for instance), script from https://trusted.com/script.js would be allowed in CSP2 and CSP3. The new `disallow-redirects` directive in the policy instructs the browser to prevent these redirections to the partially whitelisted origins.

3.4 Reporting runtime enforcement of CSP

In CSP, to collect violation reports sent by the browsers, one must use the `report-uri` directive in CSP1 and CSP2, and the `report-to` directive in CSP3. As we have shown, only violations are reported to developers. Nonetheless, the content that actually load in webpages represent valuable information that can be used to improve the security of the application, by helping to deploy more secure policies. Therefore, following the semantics of the `report-to` and `report-uri` directives used for reporting violations, we propose the `monitor-uri` and `monitor-to` directives for reporting to developers content that effectively load within a webpage. When content are allowed to load within the page upon enforcement of a CSP, browsers would generate a report, following similar algorithm used for generating violations [261, 272, 275], and send this to the developer to whatever endpoint is specified in the `monitor-uri` or `monitor-to` directives. Listing 5.13 shows an example of a policy deployed to get feedback on the runtime enforcement of the policy, as well as collect CSP violations.

```
script-src trusted.com; object-src 'none';
report-uri /reports/violations;
monitor-uri /reports/feedback;
```

Listing 5.13 – Policy to collect violations and feedback

In this example, the `monitor-uri` directive follows exactly the semantics of `report-uri` directive of CSP1 and CSP2.

3.5 Backwards compatibility and implementation overhead

The changes we propose are all backwards-compatible. Browsers not supporting the new extensions will simply ignore them. The extensions we propose to the CSP specification only introduce a few modifications to the implementations of CSP in browsers. For instance,

the blacklisting mode does not require browsers to support a new algorithm, but uses exactly the URL matching algorithm already implemented by browsers [272, 275]. There is only need for supporting a new CSP header. If the browser already implements CSP, it just enforces a blacklisting policy as a normal one. The sole difference is in the final decision: when a URL matches a blacklisting policy, the URL is not allowed, while in whitelisting mode, it is allowed. The only modifications needed to the CSP URL matching algorithm are those to support the URL parameters filtering mechanism and the new directives. The algorithm for filtering out unsafe URL parameters is rather simple to implement. We provided an implementation of this additional algorithm in Section 4 using the URLSearchParams JavaScript API [145]. We refer to it as the URL parameters checker. It consists of a dozen lines of code. In doing so, we did not modify the URL matching algorithm itself. We rather implemented a dedicated function for matching CSP against URLs parameters. As such, we preserve backwards compatibility in browsers. An implementation of the URL parameters checker can be plugged into an already existing implementation of the URL matching algorithm [261, 272, 275] of CSP in order to further apply filtering on URLs arguments. Hence, after matching a URL against a request, the URL is passed to the parameters checker which further checks that the URL does not carry unsafe parameters. Otherwise the related content is blocked from loading. Regarding partially whitelisted origins, when a URL is the result of a redirection, the whole URL, and not only its origin is checked against the policy if the `disallow-redirects` directive is used in the policy. If the URL is not explicitly allowed by the policy, it is prevented from loading.

4 Implementation

In this section, we demonstrate an implementation of the proposed extensions to the CSP specification and an evaluation using service workers. Our goal is to demonstrate that the CSP specification can be easily extended, in a backwards compatible way, with features that can improve the security of web applications, by allowing the expression of fine-grained policies. Even if these extensions are not supported in browsers, we argue that our implementation could even already be deployed on real world applications. To do so, we measured the overhead associated with deploying a service worker, which applies CSP in blacklisting mode, filters URLs arguments, and reports feedback, by using an example of web application.

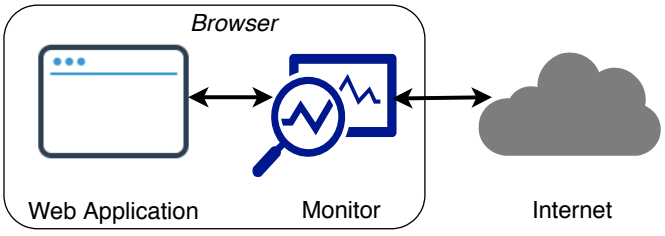


Figure 5.1 – Monitoring CSP Enforcement

In our implementation, we deploy a monitor. It acts like a proxy as shown by Figure 5.1. It intercepts requests made by the browser to load content in a web application. It is seamlessly and easily integrated to the application by the developer from the server-side, without requiring users or browsers to undertake any particular action. In addition to deploying a CSP in enforcement mode that will be enforced by the browser, the developer also

deploys a CSP in blacklisting mode. In this policy, the developer can express fine-grained policies regarding unsafe URL parameters, partially whitelisted origins, blacklisted content and provide an endpoint where to submit feedback. So the CSP in enforcement mode is enforced by the browser, while the one in blacklisting mode is enforced by the monitor. When a content is injected in a webpage, first the CSP in enforcement mode is enforced by the browser. When a URL matches the policy, the browser makes a request to fetch its content. This request is intercepted by the monitor, and the CSP in blacklisting mode is applied. Upon enforcement, the monitor checks whether the URL is not a blacklisted one and does not carry unsafe parameters. In the particular case of partially whitelisted origins, HTTP redirections are not visible to service workers for privacy reasons [275]. Nevertheless, to implement the semantics of the `disallow-redirects` directive, one can use the blacklisting and URL filtering mechanisms to either blacklist open redirects endpoints or filter out their unsafe parameters. Then once a URL is allowed by the monitor upon enforcement of the policy in blacklisting mode, the request is made. Otherwise, it is blocked. The monitor also logs all requests that it intercepts, and reports them as feedback about the enforcement of CSP in the browser.
The monitor is not meant to replace the CSP enforcement provided by browsers. It rather complements it by filling the CSP expressiveness gap at fully mitigating bypasses. It adds a reporting mechanism for developers to get the set of content being loaded in the application.

4.1 Implementation of the URL filtering algorithm

Following we provide an example of implementation of URL arguments checker algorithm.

```
function unsafeArguments(origin, url){
  if(origin.indexOf("?") == -1)
    return true;
  var oArgs = origin.split("?").slice(1).join("?"),
    uArgs = url.split("?").slice(1).join("?");
  if(!oArgs && !uArgs)
    return false;
  var oparams = new URLSearchParams(oArgs || ""),
    uparams = new URLSearchParams(uArgs || "");
  for(var it of oparams.keys()){
    if(!uparams.has(it)){
      return false;
    }else{
      var ovalue = oparams.get(it) || "",
        uvalue = uparams.get(it) || "";
      if(ovalue && ovalue != uvalue){
        return false;
      }
    }
  }
  return true;
}
```

Listing 5.14 – Implementation of the URL arguments matching algorithm using the URLSearchParams API [145] in JavaScript

The implementation is done in JavaScript, using the URLSearchParams API [145]. If the function returns `true`, then the URL is blocked, otherwise it is allowed. Recall that blacklisting URL arguments is meant to be used with CSP in blacklisting mode. So, the URL matching algorithm is first applied. Then, when the URL matches an origin in the

Table 5.1 – Matching arguments in an origin against arguments in a URL

Origin	URL	Match
a.com/?	a.com/s.js	✗
a.com/?	a.com/s.js?func=eval	✓
a.com/?	a.com/s.js?func=eval&arg=hello	✓
a.com?func	a.com/s.js	✗
a.com?func	a.com/s.js?arg=hello	✗
a.com?func	a.com/s.js?func=alert	✓
a.com?func	a.com/s.js?func=alert&arg=hello	✓
a.com?func=eval	a.com/s.js	✗
a.com?func=eval	a.com/s.js?arg=hello	✗
a.com?func=eval	a.com/s.js?func=alert	✗
a.com?func=eval	a.com/s.js?func=eval	✓
a.com?func=eval	a.com/s.js?func=eval&arg=hello	✓
a.com?func=eval&arg	a.com/s.js	✗
a.com?func=eval&arg	a.com/s.js?func=eval&arg=hello	✓
a.com?func=eval&arg	a.com/s.js?func=alert&arg=hello	✗

blacklisted CSP, it is further passed to the arguments checker. If the origin of URL does not declare any unsafe arguments, it means that the URL must be blocked since it already matches the blacklisted origin. We can say that the blacklisting is at the origin level. Otherwise if the blacklisted origin specifies unsafe arguments, then the URL is blocked if its arguments match the blacklisted arguments of the origin. In this case, we can say the blacklisting is at the arguments level. Section 3.2 gives all the details about filtering out URL parameters.
Given an origin and a URL, the URL arguments checker checks whether the blacklisted arguments of the origin are found among the arguments of the URL. Table 5.1 shows the application of this algorithm on different origins and URLs. When there is a match between the origin and the URL, then the URL is blocked.

4.2 Implementation of the URL matching algorithm

We also provide an implementation of the CSP2 URL matching algorithm [275]. We considered CSP2 since it is the latest stable version of CSP. The implementation allows us to match origins against URLs (blacklisting mode) before checking the URLs arguments against origins arguments. Our implementation follows from the specification. The code is available online at https://swexts.000webhostapp.com/monitor/.

4.3 Service workers

Service workers [149] are an experimental technology, already implemented in major browsers including Chrome, Firefox, Opera, Microsoft Edge and Safari [2]. They act as a proxy, part of the application itself, which can however intercept all HTTP requests made by the browser to load content in an application. Service workers are deployed as part of the application, but once executed, will reside in the browser and intercept all requests going out of the application, as well as all incoming responses destined to the application. Service workers have been introduced among other things, to enable web applications to provide users with

2. https://jakearchibald.github.io/isserviceworkerready/

an offline experience when network is unavailable. It appears that they perfectly fit the needs of our monitor, and we use them to implement the latter.

Description and set up

This quotation from Mozilla Developer Network defines very well service workers [149]

> A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web page/site it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to give you complete control over how your app behaves in certain situations, (the most obvious one being when the network is not available.)

First of all, the service worker itself is a JavaScript file which makes use of the specific APIs made available to it by browsers. Then the service worker is deployed by referencing it in the application whose requests it intercepts and manages.

Intercepting requests Following is how service workers intercept requests made from an application.

```
self.addEventListener('fetch', function(event) {
    url = event.request.url;
    content_type = event.request.destination;
    page = event.request.headers.referer

  }
);
```

Listing 5.15 – Intercepting requests in service workers

To intercept the URLs of requests, service workers listen for `fetch` events, which are triggered each time that a request is initiated by the browser to load content in the application. Note that those requests are done after the browser enforces the CSP of the application on the URL of the content to load. The `request` object of the event contains all the information necessary to make the request (URL of the request, type of content being loaded, the specific page from which the request is being made, data sent along the request in case of HTTP POST requests, ...) [149]. As shown in Listing 5.15, `url` represents the URL of a request intercepted by the service worker, `content_type` the type of content that the URL will load (script, image, ...)[3], and `page`, the specific page of the application from which the request is being made. This helps for instance, to deploy a single service worker for an entire application made of multiple pages. The monitor specifically makes use of this three categories of information (URL of request, content type and URL of the page), which are sufficient for it to check whether the request of the particular type should be allowed or not, in the specific page. When the request is allowed, the monitor lets it proceed using the `fetch` API [54], as shown in the following Listing 5.16.

```
event.respondWith(
  return fetch(event.request).then(function(response) {
    return response;
  })
);
```

Listing 5.16 – Making a request from the service worker

3. This information is not available on Firefox service workers

Otherwise, the request is blocked. This is achieved by generating and returning an empty response in the monitor, using the `Response` API [4].

```
event.respondWith(
  return new Response();
);
```

Listing 5.17 – Blocking a request

Deployment To deploy the service worker (monitor), one has to indicate to the browser the location (URL) of its code on the application server. Additionally, one indicates whether the monitor is deployed for a specific page or for an entire application (origin). To deploy service workers for an entire application, one can simply modify the main (HTML) page of the application, and the service worker will be deployed for the entire application.

```
...
<script>
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js', {scope: '/'})
  .then(function(reg) {
    console.log('Service Worker Started');
  }).catch(function(error) {
    console.log('Service Worker Failed');
  });
}
</script>
...
```

Listing 5.18 – Deploying a service worker for an application

In Listing 5.18 above, `sw.js` is the JavaScript file of the service worker located on the application server, and the service worker is registered for the entire application `scope: '/'`.

Enforcement of the CSP in blacklisting mode

We have already described an implementation of the URL checker algorithm (See Listing 5.14) and an implementation of CSP URL matching algorithm (Section 4.2). All these implementations are included in the service worker file that is deployed. When a web server sends a CSP in blacklisting mode (using `Content-Security-Policy-Blacklisting` HTTP header) with the response to a request to load a webpage, the monitor intercepts and saves on the fly the CSP in blacklisting mode. Then, when a request to load content on the page is intercepted (Listing 5.15), the URL of the request is checked against the blacklisting policy of the page. If the URL of request does not match the blacklisting policy, the request is normally made (Listing 5.16). Otherwise, the request is blocked. In this case, the monitor (service worker) returns an empty response (Listing 5.17).
In any case, requests that are intercepted, are logged. Requests that are blocked (because they are blacklisted content or because of their unsafe parameters) are also logged. This is submitted to the endpoint specified by the developer in the monitor policy. In our implementation, the reports are sent every 15 seconds [5]. That is the feedback about the enforcement of CSP on the application. In our implementation, we fix the URL where the feedback is sent to.

4. https://developer.mozilla.org/en-US/docs/Web/API/Response
5. We have chosen this number randomly, as all content on our example pages are loaded before this delay expires

We provide online at https://swexts.000webhostapp.com/monitor/, our ready-to-user monitor and guidelines on how to easily integrate it to web applications.

5 Evaluation

We deployed the monitor on an example web application (located on the localhost), with different types of content (scripts, images, stylesheets, fonts, XMLHttpRequests, etc.). The CSP of the page is shown in the following Listing 5.19.

```
script-src 'self' http://localhost:5000 http://localhost:7000
```

Listing 5.19 – CSP in enforcement mode deployed on the webpage

It allows scripts of the site own origin, and from 2 third party origins. To simulate third party origins, we deployed multiple example applications on different ports of the localhost (ports 5000 and 7000). The application itself is deployed on port 8000. To further apply additional checks on URLs of requests to these origins (the 2 third party origins in particular), we deployed the CSP in blacklisting mode shown in Listing 5.20

```
script-src http://localhost:5000/ajax/libs/angular.js/ http://
    localhost:7000/?
```

Listing 5.20 – Blacklisting CSP used to apply further checks on the content allowed by the CSP in enforcement mode

It blacklists (excludes) scripts whose paths start with http://localhost:5000/ajax/libs/angular.js/ from the origin http://localhost:5000 whitelisted in the CSP of Listing 5.19. URLs to http://localhost:7000 that carry any arguments will also be blocked by the monitor.
The deployed monitor has been able to successfully enforce the blacklisting policy on content we injected in the application. We also tested the monitor with nonce-based strict CSPs, and with toy browser extensions injecting content in the webpage. All such content have been successfully intercepted by the monitor and applied the CSP in blacklisting mode. Finally, the monitor was also able to log and report all content intercepted, blocked, and loaded. On a real web application, by analyzing the reported feedback, one may be able to detect potentially malicious or untrusted content loaded as a result of errors or misconfigurations, or as a result of an attacker exploiting a content injection vulnerability in the application. This also includes content dynamically injected in strict CSPs, and content injected by browser extensions.
The following is a report sent by the monitor upon enforcement of the blacklisting policy.

```
[
   {
      "url": "http://localhost:8000/script.js?BNfWB8tsrM",
      "type": "script",
      "blocked": false
   },
   {
      "url": "http://localhost:8000/script.js?K3Vc6ksIeV",
      "type": "script",
      "blocked": false
   },
   {
      "url": "http://localhost:7000/scripts/cspinclusion.js?
         callback=zleYLgrXNQ",
```

```
        "type": "script",
        "blocked": true
    },
    {
        "url": "http://localhost:5000/ajax/libs/angular.js/1.7.2/
            angular-animate.js",
        "type": "script",
        "blocked": true
    },
    {
        "url": "http://localhost:7000/scripts/cspinclusion.js",
        "type": "script",
        "blocked": false
    },
    {
        "url": "http://localhost:8000/script.js?AyhR4pkCaJ",
        "type": "script",
        "blocked": false
    },
    {
        "url": "http://localhost:8000/script.js?VPJS8xfJbn",
        "type": "script",
        "blocked": false
    },
    {
        "url": "http://localhost:7000/scripts/cspinclusion.js?
            callback=QZ4d2uiq8Y",
        "type": "script",
        "blocked": true
    },
    {
        "url": "http://localhost:7000/scripts/cspinclusion.js?
            callback=cmkRJvjPih",
        "type": "script",
        "blocked": true
    }
]
```


Listing 5.21 – Feedback reported by the monitor

Entries in the report array with the `"blocked": true` property are content that are allowed by the CSP of the page as enforced by the browser, but blocked by the monitor after applying the blacklisting CSP.

5.1 Performance overhead

We evaluated the overhead introduced in a web application, with the use of a monitor. To do so, our example webpage is embedding a set of content of different types. In particular it has 3 scripts for measuring the load time of the application:

- A first script which, when executed, registers the start time. It is the first script loaded in the webpage.
- A second script is responsible for dynamically loading in the webpage many content of different types (scripts, stylesheets, fonts, images, etc.).
- A third script injected at last, is responsible for measuring the end time. It is the last script executed in the webpage.

The page is composed of the following content:

— 20 scripts, each further making 1 synchronous XMLHttpRequest;

— 20 stylesheets, further loading 1 font each;

— 20 (JPG) images.

The application is served from the localhost to avoid latency and delay introduced with the network if it was deployed on a remote server. Time is measured using the `performance`[6] API, 100 times in different browsers (Chrome, Firefox, Opera, and Brave). All resources loaded are never cached, so that all measurements are done in the same conditions.
A first measurement is done when the application is not deploying any monitor (`No Monitor`). Then, another measurement is done when the monitor does not perform any action apart from simply forwarding all the requests that it intercepts (`Unenforced Monitor`) without applying the monitor policy. This is done to measure the overhead introduced by the use of service workers. Finally, a last measurement is done when the monitor enforces a blacklisting policy (`Enforced Monitor`).
The different times are shown in Figure 5.2 for Chrome browser, version 66, on an Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz, 64 bits, with 16Gb of RAM. Results in other browsers are similar and therefore omitted.

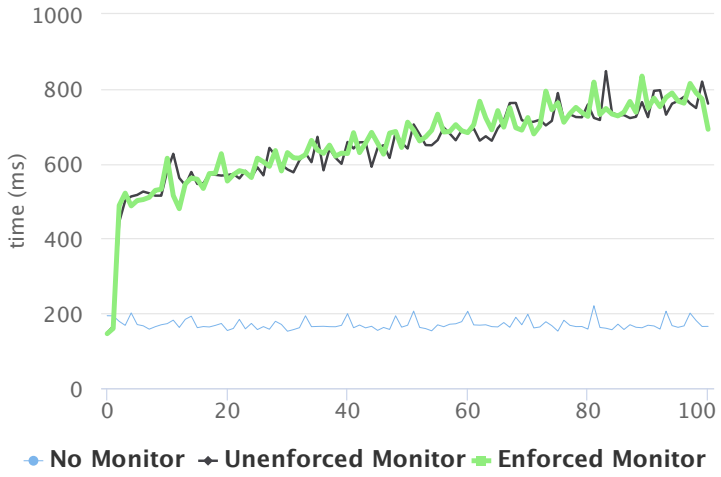


Figure 5.2 – Performance overhead of deploying the monitor

As one may observe, the main overhead is due to the use of service workers to implement the monitor (`Unenforced Monitor`). Comparatively, enforcing the monitor policy itself introduced a negligible overhead (`Enforced Monitor`). We think that this is an acceptable overhead, in comparison to the security benefits that one gains with the deployment of the monitor.

Overhead of applying the CSP in blacklisting mode

We further measured the specific overhead of applying CSP within the monitor to blacklist content. To do so, we collected the CSP and scripts of 100k Alexa sites (home pages and up to 100 pages related[7] to the site). When the page had a CSP, we extracted the whitelisted origins of its `script-src` directive. This resulted in 6,481 unique sets of `script-src`

6. https://developer.mozilla.org/en-US/docs/Web/API/Performance
7. Pages from the same origin as the site, and pages from a subdomain

directives and the associated values. We further gathered all the origins whitelisted in all `script-src` directives into a single set of unique origins, totaling 11,982 of them.
Then we randomly selected 6,481 scripts, corresponding to the number of unique `script-src` directives. To each script, using our implementation of the CSP URL matching algorithm, we applied all the 11,982 unique origins to check whether there was a match or not, and saved the time it took for the algorithm to terminate. Then, we compute the average time for applying all the 11,982 origins to the script. Figure 5.3 presents the results.

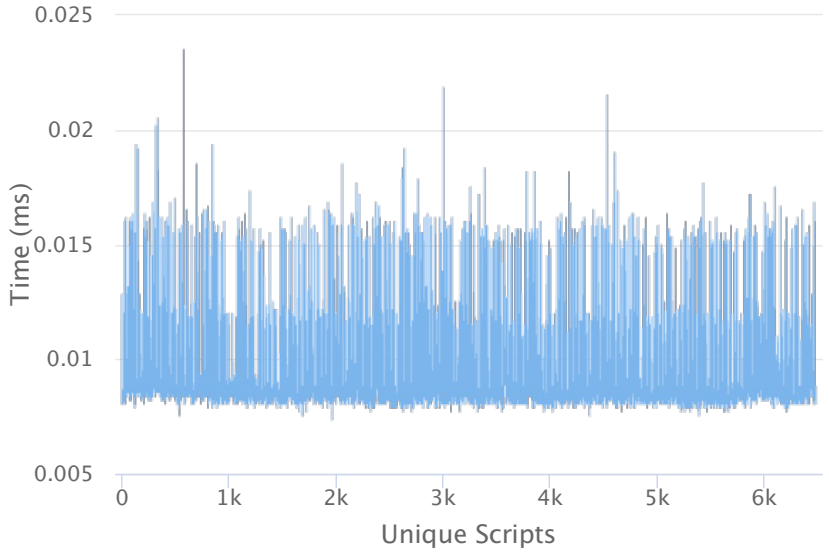


Figure 5.3 – Overhead introduced by applying CSP to content

The overhead introduced with applying CSP is really negligible. Assume that a directive contains 100 origins, in the worst case, the overhead introduced by applying CSP is less than 2.5 ms, which in our opinion, is acceptable, compared to the security benefits gained with deploying CSP in blacklisting mode.

6 Discussions and limitations

Here we discuss the limitations of the service workers we used to implement the monitor in this work.

6.1 Service workers

Service workers is still a working draft at the W3C [8], even though it is already supported by major browsers, including Firefox, Chrome, Opera, Microsoft Edge and Safari. They are backwards compatible: browsers not supporting them will not deploy the monitor, without breaking the application. Developers do not have to serve specific versions of the application for browsers not supporting service workers. Deploying the monitor as shown in Listing 5.18 (Section 4) ensures its backwards compatibility. The only modification required is in the entry page to the application, where the monitor should be indicated, using an HTML script tag. Even though, using service workers introduce an overhead, there are many improvements which can compensate this overhead, in addition to the security benefits that one would gain by deploying them. Responses to requests can be cached

8. https://w3c.github.io/ServiceWorker/

in the monitor. Later on, when the application makes a request for the same content, it is retrieved from the cache and returned to the application. In this work, we have shown an implementation of the monitor using service workers. The monitor presents the advantage of laying outside of the browser enforcement of CSP. It only intercepts requests after they are allowed by the browser upon enforcement of the CSP of the page. This allows to further check for blacklisted content or content with unsafe arguments. In the monitor, we log requests, and can delay the report time. Nonetheless, service workers have limitations. They work only for secure (HTTPS) web applications (and the localhost). They do not work in Firefox private browsing mode. Also in Firefox, it is not easy to get the type (script, image, ...) of the intercepted request. Service workers cannot intercept requests to load cross-origin iframes, for security reasons. Nonetheless, the monitor can be successfully deployed for content which executes in the context of the application such as scripts, plugins, stylesheets, images, etc.
Alternative methods can also be used to implement the monitor especially for browsers not yet supporting service workers: JavaScript proxies [82], or redefining JavaScript objects to intercept the injection of content [254]. To get all content that are injected in a webpage, similarly to a restrictive CSP in report-only mode (See Section 2), one can make use of Mutation Observers [102]. They allow to watch all changes made to the DOM of a webpage. As such, one can record all content that are injected in the webpage. These methods have many drawbacks. The first one is that they can potentially interfere with CSP enforcement as done by browsers. Moreover, contrary to these methods, service workers are easy to deploy, and can monitor all pages of entire web applications.

6.2 Browser extensions

Contrary to CSP in report-only mode, our monitor intercepts all content even those injected by browser extensions directly in the context of web applications. Content that browser extensions directly inject in the context of web applications [9], are usually not subject to the CSP of the page, and browsers would not block them. Such content are also intercepted in the monitor. If the browser allows them to load, even though they do not match the CSP of the page, the monitor instead would block them if they are not allowed by the blacklisting policy. Blocking such requests may however break extensions functionality. We did not assess how widespread this practice is among extensions, but applications developers have to take this into consideration when deploying our monitor. Should extensions content be blocked or not? Developers have to find a trade-off between the security of their applications and preserving the functionality of extensions [249].

6.3 Privacy implications of the reporting mechanism

In general, our proposal of monitoring the runtime enforcement of CSP, has to be discussed in the scope of browser extensions. Currently, a developer can observe content injected by browser extensions in web applications, by inspecting the DOM, setting up a Mutation Observer [102] or deploying a service worker as we have done. Since content injected in web pages by browser extensions are visible to web pages, we argue that browser vendors may also report such content to developers when reporting the runtime enforcement of CSP. Therefore reporting content does not leak any further information than what could

9. These are not content scripts, as content scripts execute in their own contexts. These are content further injected by content scripts directly in the context of web pages (See https://developer.chrome.com/extensions/content_scripts). In Chrome and Opera, even web accessible resources are also intercepted (See https://developer.chrome.com/extensions/manifest/web_accessible_resources)

already be obtained with the examples of techniques given above. Our proposal is just an efficient way for getting feedback, without relying on the techniques presented here, given their limitations.
There are however cases where extension developers would like to hide their injected content from web applications. For instance, in Firefox, injecting browser extensions own content, called web accessible resources, leak the extension unique identifier, which is unique on a per user basis. If this identifier is leaked to the web application, it can serve to uniquely identify and track her in future browsing sessions, as the identifier is unique for the extension and does not change throughout browsing sessions [245]. In general, it is difficult to hide this identifier from web applications. Setting up a mutation observer allows to intercept the identifier. We think that such content must also be reported as they can already be observed by different means.
There is however one case, recommended to prevent leaking extension's identifiers, in the particular case of iframes injection, as discussed on Bugzilla [20].

```
var f = document.createElement("iframe");
document.body.appendChild(f);
f.contentWindow.location = chrome.extension.getURL("iframe.htm")
    ;
```

As shown in the listing above, one can inject an iframe without leaking the unique identifier of the extension. From a mutation observer, is is not possible to observe the URL of such an iframe, and scripts running in the page cannot also observe it. Finally, service workers cannot intercept the URLs of cross-origin iframes. In this situation, and for the sake of user privacy, one may argue that the monitor of the runtime enforcement of CSP must not report the URLs of iframes included as such. We think that since such content are included in the webpage, they must also be reported.

7 Conclusion

In this work, we propose four new extensions to the current CSP specification: a new blacklisting mode, the ability to blacklist content based on unsafe URL arguments, new directives for explicitly preventing redirections to partially whitelisted origins and an efficient monitoring mechanism for collecting feedback of the runtime enforcement of CSP. These extensions are all backwards compatible, and do not break the current state of the specification, nor do they require significant modifications from current browsers implementations of the specification. We demonstrated an implementation of the new extensions using service workers, to monitor and intercept content that load on the policy, and apply additional checks on the URLs of content. We then evaluated the overhead of deploying such a policy on a web application. The monitor is easily integrated to the application by the developer from the server-side, without requiring users or browsers to undertake any particular action.

Part II

Third party web tracking

Introduction

A number of studies have demonstrated that third party tracking is very prevalent on the web today and they have analyzed the underlying tracking technologies [188, 217, 225, 242]. Lerner et al. [222] analyzed how third party tracking evolved for a period of twenty years. Trackers have been categorized either according to their business relationships with websites [225], their prominence [188, 217] or the user browsing profile that they can build [242]. Mayer and Mitchell [225] grouped tracking mechanisms into two categories called stateful (cookie-based and super-cookies) and stateless (fingerprinting). It is rather intuitive to convince ourselves of the effectiveness of a stateful tracking, since it is based on unique identifiers that are set in users' browsers. Nonetheless, the efficacy of stateless mechanisms has been extensively demonstrated. Since the pioneer work of Eckersley [185], browser fingerprinting methods have been extensively studied in the literature [163–165, 173, 180, 188, 221, 230, 259, 263]. A classification of fingerprinting techniques is provided in [264]. Those studies have contributed to raising public awareness of tracking privacy threats. Mayer and Mitchell [225] have shown that users are very sensitive to their online privacy, thus hostile to third party tracking. Englehardt et al. [189] have demonstrated that tracking can be used for surveillance purposes. The success of anti-tracking defenses is yet another illustration that users are concerned about tracking [226].

Extensions and web logins detections Since 2006, there have been multiple proposals to detect and enumerate user's browser extensions [175, 182, 193, 216]. Most of them were blog posts that were meant to raise awareness in the security community, but did not aim to scientifically evaluate extension detection at large scale, nor to perform user studies, that could explain how extensions contribute to browser fingerprinting. Similarly, there has been an ongoing discussion on Web login detection in the security community [169, 187, 194, 206, 207, 223], but no quantitative studies have been made until our work.

Sjösten et al. [249] provided the first large scale study on enumerating all free browser extensions that were available on Chrome and Firefox. The authors found that 38.96% of top 10k extensions in the Chrome Web Store were detectable with WARs. While their work lacked the evaluation of user uniqueness or fingerprintability, it disclosed the fact that 28 of the Alexa top 100k sites already used extensions detection. This finding made it clear that extension detection is more than a theoretical privacy threat, thus deserving further studying.

Starov and Nikiforakis [260] were the first to analyze fingerprintability of browser extensions and evaluating how unique users are, based on their extensions. They detected extensions based on the changes they make to the webpages. They examined the top 10,000 Chrome extensions and found that 9.2% of them were detectable on any website, and 16,6% made detectable changes on specific domains with 90% accuracy. They analyzed the stability of the proposed detection method. For a sample of 1,000 extensions, they concluded that 88% of extensions were still detectable after 4 months. To evaluate uniqueness of users based on their browser extensions, the authors collected installed extensions for 854 users.

To detect 5 extensions, their testing website needed roughly 250 ms.
Sánchez-Rola et al. [245] detected browser extensions through a timing side-channel attack, and were able to detect all extensions in Firefox and Chrome that use access control settings, regardless of the site visited. Their detection technique also relies on WARs. When querying a non-exist (fake) WAR of an extension, the authors observed a difference in the time the browser takes to respond to the query, depending on whether the extension was installed in the user's browser or not. The difference in time is caused by the access control mechanism of the browser when the concerned extension is installed or not in the browser. Because of this timing method, they had to make 10 calls per extension. To quantify the fingerprintability of users, they collected fingerprints from only 204 users and tested for 2,000 Chrome and Firefox extensions. In total, their users had 174 extensions that were fingerprintable.
Tracking protection There are a number of defenses that try to protect users against third party tracking. First, major browser vendors provide mechanisms for users to block third party cookies or browse in private/incognito mode for instance. More and more browsers even take a step further, by considering privacy as a design principle: Brave Browser [16], Tor Browser [136], TrackingFree [234], Blink [221], CLIQZ [29]. Pierre Laperdrix has done substantial work on browser fingerprinting, its stability and proposed different countermeasures as to mitigate them [191, 218–221, 266].
Well known trackers such as advertisers, which businesses heavily depend on their ability to track users, have also been taking steps towards limiting their own tracking capabilities [225]. The W3C is pushing forward the Do Not Track standard [137, 138] for users to easily express their tracking preferences so that trackers may comply with them.
But the most popular defenses are browser extensions. Being tightly integrated into browsers, they provide additional privacy features that are not by default, implemented in browsers. Well known privacy extensions are Disconnect [47], Ghostery [61], ShareMeNot [242] which is now part of PrivacyBadger [117], uBlock Origin [139] and a relatively new MyTrackingChoices [167]. Merzdovnik et al. [226] provide a large-scale evaluation of these anti-tracking defenses.

Tracking protection from the server-side

In Chapter 6, we describe and implement a privacy-preserving web architecture that gives website developers a control over third party tracking: developers are able to include functionally useful third party content, while at the same time ensuring that the end users are not tracked by the third parties. The architecture consists in two main components: a ready-to-deploy Rewrite Server, deployed by the developer server-side in order to rewrite webpages, and more precisely the URLs of third party content, by prefixing them with the URL of the second component, the Middle Party Server. Therefore, when the page loads in a browser, all third party requests are redirected to the Middle Party Server. It is a ready-to-deploy trusted third party server, under the control of the developer. When it receives requests from the browser to load third party content, it removes any tracking information from the requests and forwards them to the third party. Also, when it receives a response from the third party, it removes tracking information then returns the response to the browser.

Browser fingerprinting with extensions and web logins

Chapter 7 reports on the first large-scale behavioral uniqueness study based on 16,393. To do so, we set up a website with the aim to collect fingerprints from users, that are the browser extensions they have installed, and the websites they are logged into. We test and detect the presence of 16,743 Chrome extensions, covering 28% of all free Chrome extensions. We also detect whether the user is connected to 60 different websites. We used Web Accessible Resources [249] to detect extensions, and analyzed all free Chrome Web Store extensions. We observed that 27−28% of all free Chrome extensions were detectable on any website with 100% accuracy, and the presence of an extension can be detected in around 1ms. We analyzed the stability of the proposed detection method. In our study, we analyzed 12,164 extensions, and conclude that 72.4% of them are detectable every month during the 9-months period.

We analyze how unique users are based on their behavior, and find out that 54.86% of users that have installed at least one detectable extension are unique; 19.53% of users are unique among those who have logged into one or more detectable websites; and 89.23% are unique among users with at least one extension and one login.

We use an advanced fingerprinting algorithm and show that it is possible to identify a user in less than 625 milliseconds by selecting the most unique combinations of extensions. Because privacy extensions contribute to the uniqueness of users, we study the trade-off between the amount of trackers blocked by such extensions and how unique the users of these extensions are. We have found that privacy extensions should be considered more useful than harmful. The chapter concludes with possible countermeasures.

Chapter 6

Third party tracking protection solution for web developers

This chapter presents a server-side tracking prevention solution for web developers. It is a proposal of a web architecture that can be easily implemented by web developers, by just plugging it to existing web servers, in order to protect all their users against third party tracking. The main intuition is to redirect all third party content to a trusted Middle Party Server which removes tracking information from third party requests and responses.
The content of this chapter are replicated from the paper entitled "Control What You Include! Server-Side Protection against Third Party Web Tracking" which was published on 9th International Symposium on Engineering Secure Software and Systems (ESSoS) in 2017.

1 Introduction

Third party tracking is the practice by which third parties recognize users across different websites as they browse the web. In recent years, tracking technologies have been extensively studied and measured [185, 188, 217, 225, 230, 242] – researchers have found that third parties embedded in websites use numerous technologies, such as third-party cookies, HTML5 local storage, browser cache and device fingerprinting that allow the third party to recognize users across websites [250] and build browsing history profiles. Researchers found that more than 90% of Alexa top 500 websites [242] contain third party web tracking content, while some sites include as much as 34 distinct third party content [222].
But why do website developers include so many third party content (that may track their users)? Though some third party content, such as images and CSS [22] files can be copied to the main (first-party) site, such an approach has a number of disadvantages for other kinds of content. `Advertisement` is the base of the economic model in the web – without advertisements many website providers will not be able to financially support their website maintenance. `Third party JavaScript libraries` offer extra functionality: though copies of such libraries can be stored on the main first party site, this solution will sacrifice maintenance of these libraries when new versions are released. The developer would need to manually check the new versions. `Web mashups`, such as applications that use hotel searching together with maps, are actually based on reusing third-party content, as well as maps, and would not be able to provide their basic functionality without including the third-party content. Including JavaScript libraries, content for mashups or advertisements means that the web developers cannot provide to the users, the guarantee of non-tracking. Apart from an ethical decision not to track users, since May 2018, websites owners now

have a legal obligation as well not to track users. The ePrivacy directive (also know as 'cookie law') has been updated to a regulation, and make website owners liable for third party tracking that takes place in their websites. This regulation applied to all the services that are delivered to any individual located in the European Union. This regulation apply high penalties for any violation [161]. Hence, privacy compliance will be of high interest to all website owners and developers, and today there is no automatic tool that can help to control third party tracking.To keep a promise of non-tracking, the only solution today is to exclude any third-party content [1], thus trading functionality for privacy.
In this chapter, we present a new web application architecture that allows web developers to gain control over certain types of third party content. Our solution is based on the automatic rewriting of the web application in such a way that the third party requests are redirected to a trusted web server, with a different domain than the main site. This trusted web server may either be controlled by a trusted party, or by a main site owner – it is enough that the trusted web server has a different domain. A trusted server is needed so that the user's browser will treat all redirected requests as third party requests, like in the original web application. The trusted server automatically eliminates third-party tracking cookies and other technologies.
In summary our contributions are:

— A classification of third party content that can and cannot be controlled by the website developer.

— An analysis of third party tracking capabilities – we analyze two mechanisms: recognition of a web user, and identification of the website she is visiting [2].

— A new architecture that allows to include third party content in web applications and eliminate stateful cookie-based tracking.

— An implementation of our architecture, demonstrating its effectiveness at preventing stateful third party tracking in several websites.

2 Background and motivation

Third party web tracking is the ability of a third party to re-identify users as they browse the web and record their browsing history [225]. Tracking is often done with the purpose of web analytics, targeted advertisement, or other forms of personalization. The more a third party is prevalent among the websites a user interacts with, the more precise is the browsing history collected by the tracker. Tracking has often been conceived as the ability of a third party to recognize the web user. However, for successful tracking, each user request should contain two components:

User recognition is the information that allows tracker to recognize the user;

Website identification is the website that the user is visiting.

For example, when a user visits `news.com`, the browser may make additional requests to `facebook.com`. As a result, Facebook learns about the user's visit to `news.com`. Figure 6.1 shows a hypothetical example of such tracking where `facebook.com` is the third party.
Consider that a third party server, such as `facebook.com` hosts different content, and some of them are useful for the website developers. The web developer of another website,

1. For example, see https://duckduckgo.com/.
2. Tracking is often defined as the ability of a third party to recognize a user through different websites. However, being able to identify the websites a user is interacting with is equally crucial for the effectiveness of tracking.

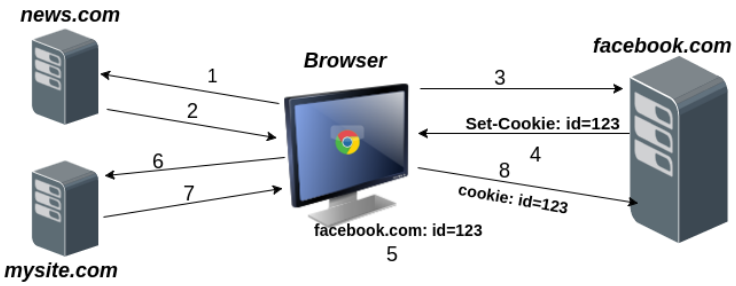


Figure 6.1 – Third Party Tracking

say `mysite.com`, would like to include such `functional` content from Facebook, such as Facebook "Like" button, an image, or a useful JavaScript library, but the developer does not want its users to be tracked by Facebook. If the web developer simply includes third party Facebook content in his application, all its users are likely to be tracked by cookie-based tracking. Notice that each request to `facebook.com` also contains an HTTP Referrer header, automatically attached by the browser. This header contains the website URL that the user is visiting, which allows Facebook to build the user's browsing history profile.
The example demonstrates cookie-based tracking, which is extremely common [242]. Other types of third party tracking, that use client-side storage mechanisms, such as HTML5 LocalStorage, or cache, and device fingerprinting that do not require any storage capabilities, are also becoming more and more popular [188].

Web developer perspective

A web developer may include third party content in her webpages, either because this content `intentionally` tracks users (for example, for targeted advertising), or because this content is important for the functioning of the web application. We therefore distinguish two kinds of third party content from a web developer perspective: `tracking` and `functional`. `Tracking` content is `intentionally` embedded by website owner for tracking purposes. `Functional` content is embedded in a webpage for other purposes than tracking: for example, JavaScript libraries that provide additional functionality, such as jQuery, or other components, such as maps. In this work, we focus on `functional` content and investigate the following questions:

— What kind of third party content can be controlled from a server-side (web developer) perspective?

— How to eliminate the two components of tracking (user recognition and website identification) from functional third party content that websites embed?

2.1 Browsing context

When a browser renders a webpage delivered by a first party, the page is placed within a `browsing context` [19]. A browsing context represents an instance of the browser in which a document such as a webpage is displayed to a user, for instance browser tabs, and popup windows. Each browsing context contains 1) a copy of the browser properties (such as browser name, version, device screen etc), stored in a specific object; 2) other objects that depend on the origin of the document according to SOP. For instance, the object `document.cookie` gives the cookies related to the domain and path of the current context.
In-context and cross-context content. Certain types of content embedded in a webpage, such as images, links, and scripts, are associated with the context of the webpage, and

we call them `in-context` content. Other types of content, such as `<iframe>`, `<embed>`, and `<object>` tags are associated with their own browsing context, and we call them `cross-context` content. Usually, cross-context content, such as `<iframe>` elements, cannot be visually distinguished from the webpage in which they are embedded, however they are as autonomous as other browsing contexts, such as tabs or windows. Table 6.1 shows different third party content and their execution contexts.

Table 6.1 – Third party content and execution context

	HTML tags	Third party content
in-context	`<link>`	stylesheets
	`<img>`	images
	`<audio>`	audios
	`<video>`	videos
	`<form>`	forms
	`<script>`	scripts
cross-context	`<(i)frame>, <frameset>, <a>, <area>`	web pages
	`<object>, <embed>, <applet>`	plugins and web pages

The Same Origin Policy manages interactions between different browsing contexts. In particular, it prevents in-context scripts from interacting with cross-context iframes in case their origins are different. To communicate, they may use inter-frame communication APIs such as `postMessage` [116].

2.2 Third party tracking

In this work, we consider only stateful tracking technologies – they require an identifier to be stored on the client-side. The most common storage mechanism is cookies, but others, such as HTML5 LocalStorage and browser cache can also be used for stateful tracking. Figure 6.2 presents the well-known stateful tracking mechanisms [225]. We distinguish two components necessary for successful tracking: user recognition and website identification. For each component, we describe the capabilities of in-context and cross-context. We also distinguish `passive tracking` (through HTTP headers) and `active tracking` (through JavaScript or plugin script).

	User recognition		**Website identification**	
	Passive	Active	Passive	Active

in-context	HTTP cookies
	Cache-Control
	Etag
	Last-Modified
		-	Referer
			Origin
				`document.URL`
				`document.location`
				`window.location`

cross-context		Flash LSOs
		`document.cookie`
		`window.localStorage`
		`window.indexedDB`
			Referer	`document.referrer`

Figure 6.2 – Stateful tracking mechanisms

In-context tracking. In-context third party content is associated with the browsing context of the webpage that embeds it (see Table 6.1).

Passively, such content may use HTTP headers to recognize a user and identify the visited website. When a webpage is rendered, the browser sends a request to fetch all third party content embedded in that page. The responses from the third party, along with the requested content, may contain HTTP headers that are used for tracking. For example, the Set-cookie HTTP header tells the browser to save third party cookies, that will be later on automatically attached to every request to that third party in the Cookie header. Etag HTTP header and other cache mechanisms like Last-Modified and Cache-Control HTTP headers may also be used to store user identifiers [250] in a browser. To identify the visited website, a third party can either check the Referer HTTP header, automatically attached by the browser, or an Origin header [3].

Actively, in-context third party content cannot use browser storage mechanisms, such as cookies or HTML5 Local Storage associated to the third party because of the limitations imposed by the SOP (see Section 2.1). For example, if a third party script from third.com uses `document.cookie` API, it will read the cookies of the main website, but not those of third.com. This allows tracking within the main website but does not allow tracking cross-sites [242]. For website identification, third party active content, such as scripts, can use several APIs, for example `document.location`.

Cross-context tracking. Cross-context content, such as iframe, is associated with the browsing context of the third party that provided this content.

Passively, the browser may transmit HTTP headers used for user recognition and website identification, just like in the case of in-context content. Every third-party request for cross-context content will contain the URL of the embedding webpage in its Referer header.

Requests to fetch third party content further embedded inside a cross-context (such as iframe) will carry, not the URL of the embedding webpage, but that of the iframe in their Referer or Origin headers (in the case of CORS requests). This prevents them from passively identifying the embedding webpage.

Actively, cross-context third party content can use a number of APIs to store user identifiers in the browser. These APIs include cookies (`document.cookie`), HTML5 LocalStorage (`document.localStorage`), IndexedDB, and Flash Local Stored Objects (LSOs). For website identification, `document.referrer` API can be used – it returns the value of the HTTP Referrer header transmitted in the request to the cross-context third party.

Combining in-context and cross-context tracking. Imagine a third party script from `third.com` embedded in a webpage – according to the context and to the SOP, it is in-context. If the same webpage embeds a third party iframe from `third.com` (cross-context), then because of SOP, such script and iframe cannot interact directly. However, they can still communicate through inter-frame communication APIs such as `postMessage` [116].

On one hand, the in-context script can easily identify the website using APIs such as `document.location`. On the other hand, the cross-context iframe can easily recognize the user by calling `document.cookie`. Therefore, if the iframe and the script are allowed to communicate, they can exchange those partial tracking information to fully track the user. For example, a social widget, such as the Facebook "Like" button, or Google "+1" button, may be included in webpages as a script. When the social widget script is executed on the client-side, it loads additional scripts, and new browsing contexts (iframes) allowing the third party to benefit from both in-context and cross-context capabilities to track users.

3. Origin header is also automatically generated by the browser when the third party content is trying to access data using Cross-Origin Resource Sharing [40] mechanism.

3 Privacy-preserving web architecture

For third party tracking to be effective, two capabilities are needed: 1) the tracker should be able to identify the website in which it is embedded, and 2) recognize the user interacting with the website. Disabling only one of these two capabilities for a given third party already prevents tracking. In order to mitigate stateful tracking (see Section 2), we make the following design choices:

1. **Preventing only user recognition for in-context**. As shown in Table 6.2, in-context content cannot perform any active user recognition. We are left with passive user recognition and (active and passive) website identification. Preventing passive user recognition for such content (images, scripts, forms) is possible by removing HTTP headers such as `Cookie, Set-cookie, ETag` that are sent along with requests/responses to fetch those content.

   Note that it is particularly difficult to prevent active website identification because trying to alter or redefine `document.location` or `window.location` APIs, will cause the main page to reload. Therefore, in-context active content (scripts) can still perform active website identification. That notwithstanding, since we remove their user recognition capability, tracking is therefore prevented for in-context content.

2. **Preventing only website identification for cross-context**. We prevent passive website identification by instructing the browser not to send the HTTP `Referer` header along with requests to fetch a cross-context content. Therefore, when the cross-context content gets loaded, the tracker is unable to identify the website in which it is embedded in. Indeed, executing `document.referrer` returns an empty string instead of the URL of the embedding page.

   Because of the limitations of the SOP, a website owner has no control over cross-context third party content, such as iframes. Therefore, active and passive user recognition can still happen in third party cross-context. We discuss other possibilities to block some active user recognition APIs in Section 4.1. Nonetheless, since website identification is not possible, tracking is therefore prevented for cross-context third party content.

3. **Preventing communication between in-context and cross-context content**. Our architecture proposes a way to block such communications that can be done by `postMessage` API. We discuss the limitations of this approach in Section 4.1.

To help web developers keep their promises of non-tracking and still include third-party content in their web applications, we propose a new web application architecture. This architecture allows web developers to 1) automatically rewrite the URLs of all in-context third party content embedded in a web application, 2) redirect those requests to a trusted third party server which 3) remove/disable known `stateful` tracking mechanisms (see Section 2) for such content; 4) rewrite and redirect cross-context requests to the trusted third party so as to prevent website identification and communication with in-context scripts.
Figure 6.3 provides an overview of our web application architecture. It introduces two new components fully controlled by the website owner.
Rewrite Server (Section 3.1) acts like a reverse proxy [122] for the original web server. It rewrites the original web pages in such a way that all the requests to fetch all the third party content that they embed are redirected through the Middle Party Server before reaching the intended third party server.

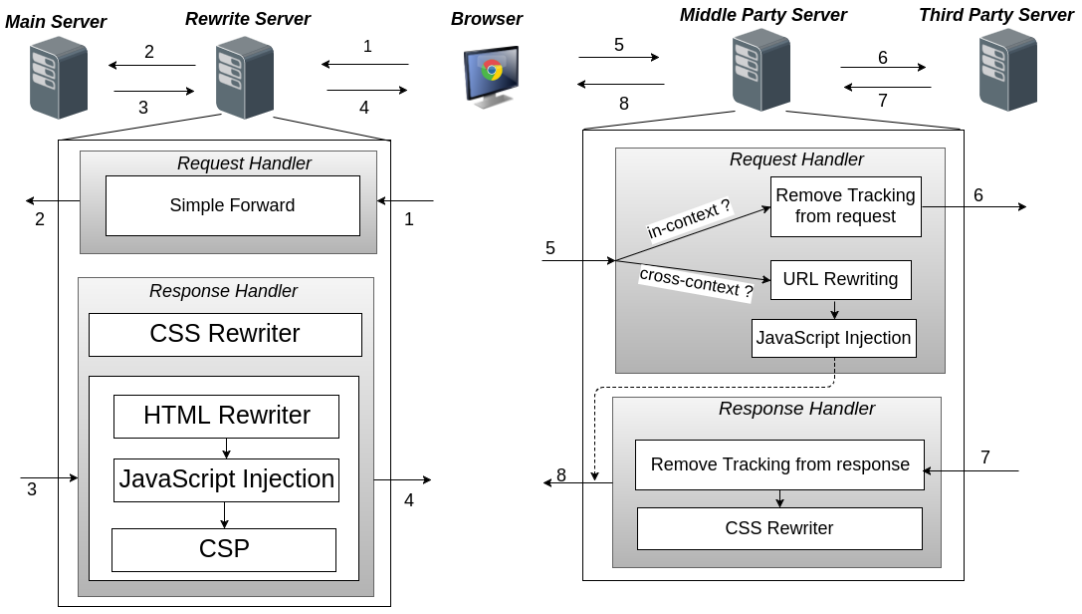


Figure 6.3 – Privacy-Preserving Web Architecture

Middle Party Server (Section 3.2) is at the core of our solution since it intercepts all browser third party requests, removes tracking, then forwards them to the intended third parties. For every response from a third party, the server removes tracking information and forwards the response back to the browser. For in-context content such as images and scripts, the Middle Party Server prevents user recognition and website identification, while for cross-context content such as iframes, it prevents website identification and communication with other in-context scripts.

3.1 Rewrite Server

The goal of the Rewrite Server is to rewrite the original content of the requested webpages in such a way that all third party requests will be redirected to the Middle Party Server. It consists of three main components: static HTML rewriter for HTML pages, static CSS rewriter and JavaScript injection component. In each webpage, a JavaScript code is loaded that ensures that all dynamically generated third party content are redirected to the Middle Party Server as well.

HTML and CSS Rewriter rewrites the URLs of static third party content embedded in original web pages and CSS files in order to redirect them to the Middle Party Server. For example, the URL of a third-party script source `http://third.com/script.js` is written so that it is instead fetched through the Middle Party Server: `http://middle.com/?src=http://third.com/script.js`. The **HTML Rewriter** component is implemented using the Jsdom HTML parser [69], and **CSS Rewriter**, using the CSS parser [44] module for Node.js.

JavaScript Injection. The Rewrite Server also injects a script in all original webpages after they are rewritten. This script controls APIs used to dynamically inject content inside a webpage once the webpage is rendered in a browser. It is available at `https://webstats.inria.fr/sstp/dynamic.js`. Table 6.2 shows APIs that can be used to dynamically inject third party content within a webpage. They are controlled using the injected script.

A **Content Security Policy (CSP)** [275] is injected in the response header of each webpage in order to prevent third parties from bypassing the rewriting and redirection to

Table 6.2 – Injecting dynamic third party content

API	Content
`document.createElement`	inject content from Table 6.1
`document.write`	any content
`window.open`	Web pages(popups)
`Image`	images
`XMLHttpRequest`	any data
`Fetch, Request`	any content
`EventSource`	stream data
`WebSocket`	websocket data

the Middle Party Server. A CSP delivered with the webpage controls the resources of that page by specifying which resources are allowed to be loaded and executed. By limiting the resource origins to only those of the Middle Party Server and the website own domain, we prevent third parties from bypassing the redirection to the Middle Party Server in order to load content directly from a third party server. Such attempts will get blocked by the browser upon enforcement of the CSP of the page. The following listing gives the CSP injected in all webpages, assuming that `middle.com` is the domain of the Middle Party Server.

```
Content-Security-Policy: default-src 'self' middle.com;
    object-src 'self';
```

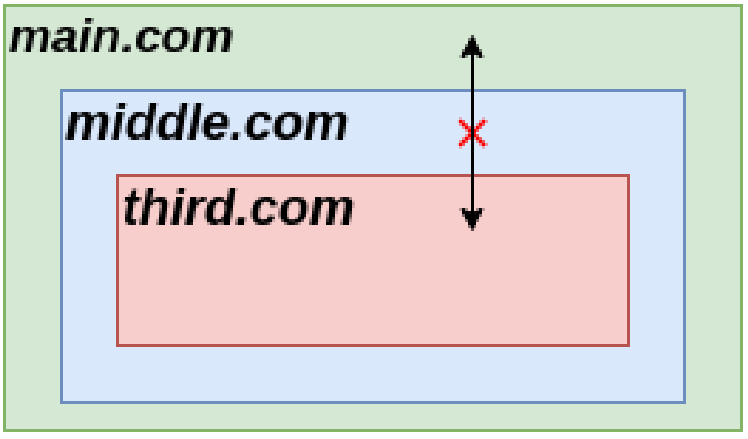


Figure 6.4 – Preventing trackers from combining in-context and cross-context tracking

3.2 Middle Party

The main goal of the Middle Party is to proxy the requests and responses between browsers and third parties in order to remove tracking information exchanged between them. It functions differently for in-context and cross-context content.

In-context content are scripts, images, etc. (see Table 6.1). Since a third party script from `http://third.com/script.js` is rewritten by the Rewrite Server to `http://middle.com/?src=http://third.com/script.js`, it is fetched through the Middle Party Server. This hides the third party destination from the browser, and therefore prevents it from attaching third party HTTP cookies to such requests. Because the browser will still

attach some tracking information to the requests, when the middle party receives a request URL from the browser, it will then take the following steps. **Remove tracking from request** that are set by the browser as HTTP headers. Among those headers are `Etag, If-Modified-Since, Cache-Control, Referer`. Next, it makes a request to the third party in order to get the content of the script `http://third.com/script.js`. **Remove tracking from response** returned by the third party. The headers that the third party may send are `Set-Cookie, Etag, Last-Modified, Cache-Control`. **C**SS Rewriter rewrites the response if the content is a CSS file, in order to also redirect to the Middle Party Server any third party content that they may embed. Finally, the response is returned back to the browser.

Cross-context content are iframes, links, popups, etc. (see Table 6.1). The Middle Party Server prevents website identification for cross-context content and communication with in-context scripts. This is done by loading cross-context content from another cross-context controlled by the Middle Party Server as illustrated in Figure 6.4.

For instance, a third party iframe from `http://third.com/page.html` is rewritten to `http:// middle.com/?emb=http://third.com/page.html`. When the Middle Party Server receives such a request URL from the browser, it takes the following actions: **URL Rewriting.** Instead of fetching directly the content of `http://third.com/page.html`, the Middle Party Server generates a content in which it puts the URL of the third party content as a hyperlink `<a href = "http://third.com/page.html" rel = "noreferrer noopener"></a>`. The most important part of this content is in the `rel` attribute value. Therefore, `noreferrer noopener` instructs the browser not to send the `Referer` header when the link `http://third.com/page.html` is navigated. **JavaScript Injection** module adds a script to the content so that the link gets automatically navigated once the content is rendered by the browser. Once the link is followed, the browser fetches the third party content directly on the third party server, without going through the Middle Party server anymore. However it does not include the `Referer` header for identifying the website. Therefore, the `document.referrer` API also returns an empty string inside the iframe context. This prevents it from identifying the website. The third party server response is placed in a new iframe nested within a context that belongs to the Middle Party, and not directly in the site webpage. This prevents in-context scripts and the cross-context content from exchanging tracking information as illustrated in Figure 6.4.

HTTPS content. We recommend deploying the Middle Party Server as an HTTPS server. Therefore, third party content originally served over HTTPS (before rewriting) still get served over HTTPS even in the presence of the Middle Party Server . Moreover, third party content originally served over HTTP would get blocked by current browsers according to the Mixed Content policy [273]. With an HTTPS Middle Party, HTTP third party requests will not be prevented from loading since they are fetched over HTTPS through the Middle Party.

Multiple redirections. A third party may attempt to circumvent our solution by performing multiple redirections. This is commonly used in advertisements (though ads are not within the scope of this work).

When a (third party) web server wants to perform a redirection to another server, it usually does so by including in the response, a special HTTP *Location* that indicates the server to which the next request will be sent. The Middle Party Server prevents such circumvention by rewriting the *Location* header so that the browser sends the next redirection request to the Middle Party Server again. As a result, all the redirections pass via the Middle Party.

4 Implementation

We have implemented both the Rewrite Server and the Middle Party Server as full Node.js [105] web servers supporting `HTTP(S)` protocols and web sockets. Implementation details are available at http://www-sop.inria.fr/members/Doliere.Some/essos/.

Rewrite Server

In our implementation, we deploy the Rewrite Server on the same physical machine as the original web application server. In order to do so, we moved the original server on a different port number, and the Rewrite Server on the initial port of the original server. Therefore, requests that are sent by browsers first reach the Rewrite Server. It then simply forwards them to the original server, which handles the request as usual and returns a response to the Rewrite Server. Then, HTML webpages, and CSS files are rewritten using the **HTML Rewriter** and **CSS Rewriter** components respectively. To handle dynamic third party content, we inject a script. And in order to prevent malicious third parties from bypassing the redirection, we inject a CSP (See Section 3.1).

Middle Party

All requests to load third party contents embedded in a website deploying our architecture will go through the Middle Party Server. in-context and cross-context contents are handled differently.
In-context content are simply stripped off tracking information that they carry from the browser to the third parties and vice versa. See Section 3 for the list of tracking information that are removed from third party requests and responses. In particular, third party CSS responses are rewritten, using the **CSS Rewriter** component, to redirect to the Middle Party Server any third party content that they may further embed. As in the case of the Rewrite Server, this component is implemented using a CSS parser [44] for Node.js
Cross-context content are handled in a way that the original website identity is not leaked to them. They are also prevented from communicating with any in-context third party content to exchange tracking information. If the cross-context URL was http://third.com/page.html, instead of making a request to third.com, the Middle Party Server returns to the browser, a response consisting of rewriting the URL to

```
<a href="http://third.com/page.html" rel="noreferrer noopener"><
    /a>.
```

and injecting the following script:

```
var third_party = document.getElementsByTagName("a")[0];
if(window.top == window.self){
  third_party.target = "_blank";
  third_party.click();
  window.close();
}else{
  var iframe = document.createElement("iframe");
      iframe.name = "iframetarget";
  document.body.appendChild(iframe);
  third_party.target = "iframetarget";
  third_party.click();
}
```

Overall, when this response is rendered, the browser will not send the `Referer` header to the third party, and the third party is prevented from communicating with in-context content, as explained in Section 3.2.

4.1 Discussion and limitations

Our approach suffers from the following limitations. First, while our implementation prevents cross-context and in-context contents from communicating with each other using `postMessage` API, in-context third party script can however identify the website a user visits via `document.location.href` API. The script can include the website URL, say `http://main.com`, as a parameter of the URL of a third party iframe, for example `http://third.com/page.html?ref=http://main.com` and dynamically embed it in the webpage. In our architecture, this URL is rewritten and routed to the Middle Party. Since, the Middle Party Server does not inspect URL parameters, this information will reach the third party even though the `Referer` is not sent with cross-context requests.
Another limitation is that of dynamic CSS changes. For instance, changing the background image via the *style* object of an element in the webpage is not captured by the dynamic rewriting script injected in webpages. Therefore, if the image was a third party image, the CSP will prevent it from loading.
Performance overhead There is a performance cost associated with the Rewrite Server, which can be evaluated as the cost of introducing any reverse proxy to a web application architecture (See Section 3.1). Rewriting contents server-side and browser-side is also expensive in terms of performance. We believe that server-side caching mechanisms, in particular for static webpages, may help speed up the responsiveness of the Rewrite Server. The Middle Party Server may also lead to performance overhead especially for webpages with numerous third party contents. Therefore, it can be provided as a service by a trusted external party, as it is the case for Content Distribution Networks (CDNs) serving contents for many websites.
Extension to stateless tracking Even though this work did not address stateless tracking such as device fingerprinting, our architecture already hides several fingerprintable device properties and can be extended to several others: 1) The redirection to the Middle Party anonymizes the real IP addresses of users; 2) Some stateless tracking APIs such as `window.navigator`, `window.screen`, and `HTMLCanvasElement` can be easily removed or randomized from the context of the webpage to mitigate in-context fingerprinting.
Possibility of blocking active user recognition in cross-context. With the prevalence of third party tracking on the web, we have shown the challenges that a developer will face towards mitigating that. The sandbox attribute for iframes help prevent access to security-sensitive APIs. As tracking has become a hot concern, we suggest that similar mechanisms can help first party websites tackle third party tracking. The sandbox attribute can for instance be extended with specific values to tackle tracking. Nonetheless, the sandbox attribute can be used to prevent cross-context from some stateful tracking mechanisms [76].

5 Evaluation and Case Study

Demo website We have set up a demo website that embeds a collection of third party content, both in-context and cross-context. In-context content include images, HTML5 audio and video, and a Google Map which further loads dynamic content such as images, fonts, scripts, and CSS files. A Youtube video is embedded as cross-context content in an

iframe. The demo website is deployed at https://sstp-rewriteproxy.inria.fr. With the deployment of our solution, there is no change from a user perspective on how the demo website is accessed. Indeed, it is still accessible at https://sstp-rewriteproxy.inria.fr. However from the server-side, it is the Rewrite Server which is now running at https://sstp-rewriteproxy.inria.fr instead of the original server. It then intercepts user requests and forwards them to the original server which has been moved on port 8080 (http://sstp-rewriteproxy.inria.fr:8080), hidden from users and the outside.
The Middle Party Server runs at https://sstp-middleparty.inria.fr. With our architecture deployed, all requests to fetch third party content embedded in the demo website are redirected to the Middle Party Server. For in-context content, it removes any tracking information in the requests sent by the browser. Then it forwards the requests to the third parties. Any tracking information set by the third parties in the responses are also removed before being forwarded to the browser. For the cross-context content (Youtube Video in our demo), it is not directly loaded as an iframe inside the demo page. Instead, an iframe from the Middle Party Server is created and embedded inside the demo webpage. Then the Youtube video is automatically loaded in another iframe inside this first iframe whose context is that of the Middle Party Server. During this process, the *Referer* header is not leaked to Youtube (Section 3.2), preventing it from identifying the demo website in which it is included.

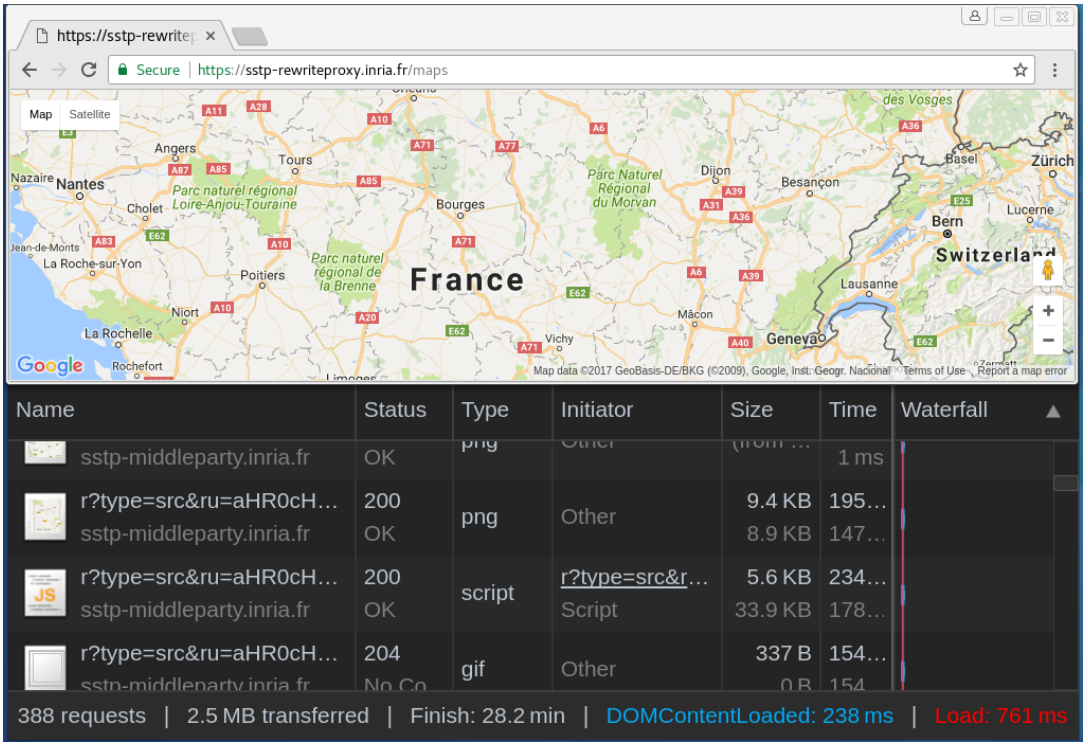


Figure 6.5 – A demo page displaying a Google Maps

Figure 6.5 shows a screenshot of the redirection of third party requests to the Middle Party Server.
Real websites. Since we did not have access to real websites, we could not install the Rewrite Server and evaluate our solution on them. We therefore implemented a browser proxy based on a Node.js proxy [106], and included all the logic of the Rewrite Server within the proxy. The proxy was deployed at https://sstp-rewriteproxy.inria.fr:5555 and

acts like the Rewrite Server for real websites intercepting and forwarding requests to them, and rewriting the responses in order to redirect them to our Middle Party Server deployed at https://sstp-middleparty.inria.fr.
We then evaluated our solution on different kinds of websites: a news website http://www.bbc.com, an entertainment website http://www.imdb.com, and a shopping website http://verbaudet.fr. All three websites load content from various third party domains. Visually, we did not notice any change in the behaviors of the websites. We also interacted with them in a standard way (clicking on links on a news website, choosing products and putting them in the basket on the shopping website) and the main functionalities of the websites were preserved.
Overall, these evaluation scenarios have helped us improve the solution, especially rewriting dynamically injected third party content. We believe that this implementation will get even mature in the future when we will be able to convince some website owners to deploy it.
Limitations of the evaluation on real websites.
The evaluation on the real websites may break some features of the website or introduce performance issues. Here, we discuss such problems and how to prevent them.
Third party identity (OpenID) providers such as Facebook or Google need to use third party cookies in order to authenticate users to third party websites. Therefore, stripping off cookies can prevent users from successfully logging in to the related websites. In a deployment scenario, we make it possible for the developer to instruct the Rewrite Server not to rewrite such third party identity provider content so that users can still log in.
Furthermore, it is common for websites to rely on Content Distribution Networks (CDNs), from which they load content for performance purposes. Therefore, rewriting and redirecting CDNs requests to the Middle Party Server can introduce performance issues. In this case also, a developer can declare a list of CDNs whose requests should not be rewritten by the Rewrite Server.
Finally, as one may have noticed, the real websites we have considered in our evaluation scenario are all HTTP websites. We could not evaluate our solution on real HTTPS websites because HTTPS requests and responses that arrive at the browser proxy are encrypted. Therefore, we could not rewrite third party content that are embedded in such websites.

6 Conclusion

Most of the previous research analyzed third party tracking mechanisms, and how to block tracking from a user perspective. In this chapter, we classified third party tracking capabilities from a website developer perspective. We proposed a new architecture for website developers that allows to embed third party content while preserving users' privacy. We implemented our solution, and evaluated it on real websites to mitigate stateful tracking.

Chapter 7

Browser fingerprinting based on extensions and web logins

Preamble

This chapter presents browser fingerprinting based on the browser extensions a user installs and the websites she is logged into. The content of this chapter are replicated from the paper title "To Extend or not to Extend: on the Uniqueness of Browser Extensions and Web Logins" that has been published in the proceedings of the 2018 Workshop on Privacy in the Electronic Society (WPES'18). It has been done in collaboration with other authors from Inria.

1 Introduction

In the last decades, researchers have been actively studying users' uniqueness in various fields, in particular biometrics and privacy communities hand-in-hand analyze various characteristics of people, their behavior and the systems they are using. Related research showed that a person can be characterized based on her typing behavior [243, 279], mouse dynamics [239], and interaction with websites [190]. Furthermore, Internet and mobile devices provide rich environment where users' habits and preferences can be automatically detected. Prior works showed that users can be uniquely identified based on websites they visit [231], smartphone apps they install [166] and mobile traces they leave behind them [183].
Since the web browser is the tool people use to navigate through the Web, privacy research community has studied various forms of `browser fingerprinting` [165, 180, 185, 188, 191, 230]. Researchers have shown that a user's browser has a number of "physical" characteristics that can be used to uniquely identify her browser and hence to track it across the Web. Fingerprinting of users' devices is similar to physical biometric traits of people, where only physical characteristics are studied.
Similar to previous demonstrations of user uniqueness based on their behavior [166, 231], `behavioral characteristics`, such as browser settings and the way people use their browsers can also help to uniquely identify Web users. For example, a user installs web browser extensions she prefers, such as AdBlock [6], LastPass [86] or Ghostery [61] to enrich her Web experience. Also, while browsing the Web, she logs into her favorite social networks, such as Gmail [66], Facebook [53] or LinkedIn [90]. In this work, we study `users' uniqueness` based on their behavior and preferences on the Web: we analyze how unique are Web users based on their `browser extensions and logins`.

In recent works, Sjösten et al. [249] and Starov and Nikiforakis [260] explored two complementary techniques to detect extensions. Sánchez-Rola et al. [245] then discovered how to detect any extension via a timing side channel attack. These works were focused on the technical mechanisms to detect extensions, but what was not studied is how browser extensions contribute to uniqueness of users at large scale. Linus [223] showed that some social websites are vulnerable to the "login-leak" attack that allows an arbitrary script to detect whether a user is logged into a vulnerable website. However, it was not studied whether Web logins can also contribute to users' uniqueness.
In this work, we performed the first large-scale study of user uniqueness based on browser extensions and Web logins, collected from more than 16,000 users who visited our website (see the breakdown in Fig. 7.6). Our experimental website identifies installed Google Chrome [62] extensions via Web Accessible Resources [249], and detects websites where the user is logged into, by methods that rely on URL redirection and CSP violation reports. Our website is able to detect the presence of 13k Chrome extensions on average per month (the number of detected extensions varied monthly between 12,164 and 13,931), covering approximately 28% of all free Chrome extensions [1] . We also detect whether the user is connected to one or more of 60 different websites. Our main contributions are:

— A large scale study on how unique users are based on their browser extensions and website logins. We discovered that 54.86% of users that have installed at least one detectable extension are unique; 19.53% of users are unique among those who have logged into one or more detectable websites; and 89.23% are unique among users with at least one extension and one login. Moreover, we discover that 22.98% of users could be uniquely identified by Web logins, even if they disable JavaScript.

— We study the privacy dilemma on Adblock and privacy extensions, that is, how well these extensions protect their users against trackers and how they also contribute to uniqueness. We evaluate the statement "the more privacy extensions you install, the more unique you are" by analyzing how users' uniqueness increases with the number of privacy extensions they install; and by evaluating the tradeoff between the privacy gain of the blocking extensions such as Ghostery [61] and Privacy Badger [117].

We furthermore show that browser extensions and Web logins can be exploited to fingerprint and track users by only checking a limited number of extensions and Web logins. We have applied an advanced fingerprinting algorithm [197] that carefully selects a limited number of extensions and logins. For example, Figure 7.1 shows the uniqueness of users we achieve by testing a limited number of extensions. The last column shows that 54.86% of users are unique based on all 16,743 detectable extensions. However, by testing 485 carefully chosen extensions we can identify more than 53.96% of users. Besides, detecting 485 extensions takes only 625ms.
Finally, we give suggestions to the end users as well as website owners and browser vendors on how to protect the users from the fingerprinting based on extensions and logins.
In our study we did not have enough data to make any claims about the stability of the browser extensions and web logins because only few users repeated an experiment on our website (to be precise, only 66 users out of 16,393 users have made more than 4 tests on our website). We leave this as a future work.

1. The list of detected extensions and websites are available on our website: https://extensions.inrialpes.fr/faq.php

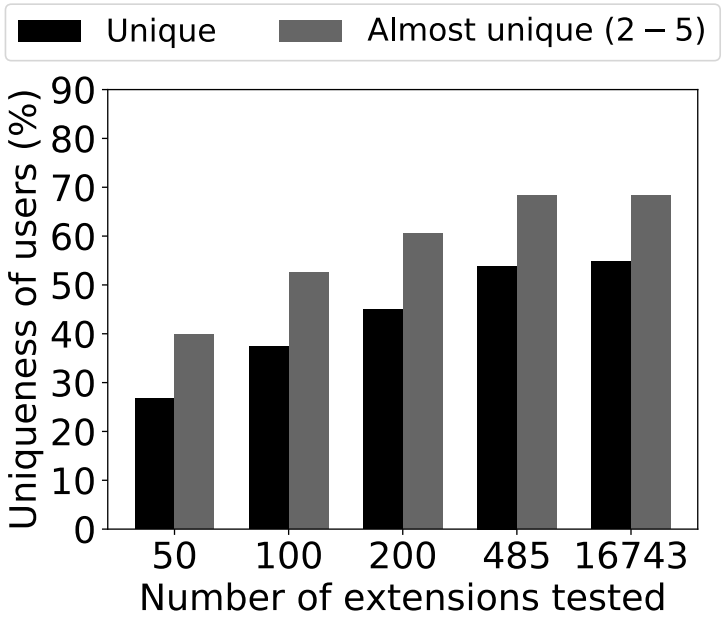


Figure 7.1 – Results of general fingerprinting algorithm. Testing 485 carefully selected extensions provides a very similar uniqueness result to testing all 16,743 extensions. Almost unique means that there are 2–5 users with the same fingerprint.

2 Background

2.1 Detection of browser extensions

In the Google Chrome web browser, each extension comes with `a manifest file` [64], which contains metadata about the extension. Each extension has a unique and permanent identifier, and the manifest file of an extension with identifier `extID` is located at `chrome-extension://[extID]/manifest.json`. The manifest file has a section called `web_accessible_resources` (WARs) that declares which resources of an extension are accessible in the content of any webpage [63]. The WARs section specifies a list of paths to such resources, presented by the following type of URL:
`chrome-extension://[extID]/[path]`, where `path` is the path to the resource in the extension.
Therefore, a script that tries to load such an accessible resource in the context of an arbitrary webpage is able to check whether an extension is installed with a 100% guarantee: if the resource is loaded, an extension is installed, otherwise it is not. Figure 7.2 shows an example of AdBlock extension detection: the script tries to load an image, which is declared in the `web_accessible_resources` section of AdBlock's manifest file. If the image from AdBlock, located at `chrome-extension://[AdBlockID]/icons/icons24.png` is successfully loaded, then AdBlock is installed in the user's browser.
Sjösten et al. [249] were the first to crawl the Google Chrome Web Store and to discover that 28% of all free Chrome extensions are detectable by WARs. An alternative method to detect extensions that was available at the beginning of our experiment, was a behavioral method from XHOUND [260], but it had a number of false positives and detected only 9.2% of top 10k extensions . Therefore, we decided to reuse the code from Sjösten et al. [249] with their permission to crawl Chrome Web Store and identify detectable extensions based on WARs. During our experiment, we discovered that WARs could be detected in other

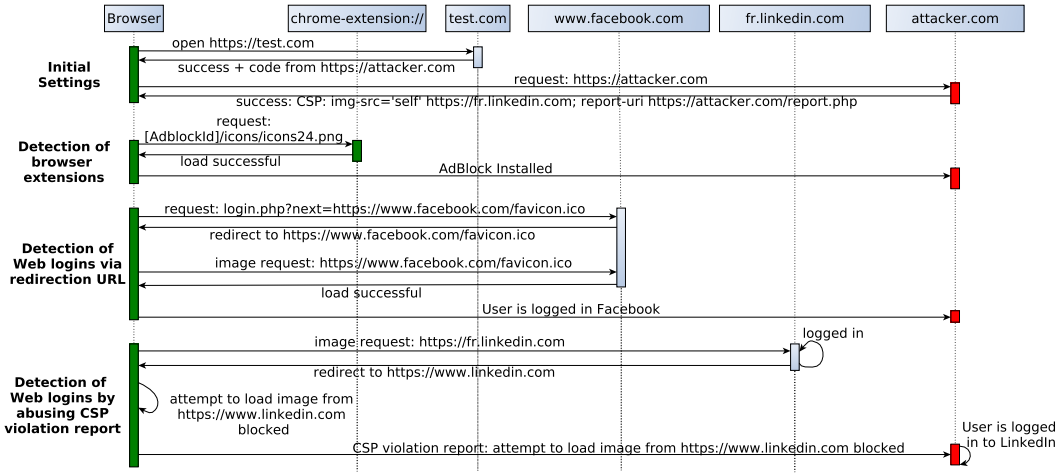


Figure 7.2 – Detection of browser extensions and Web logins. A user visits a benign website `test.com` which embeds third party code (the attacker' script) from `attacker.com`. The script detects an icon of `Adblock` extension and concludes that `Adblock` is installed. Then the script detects that the user is logged into Facebook when it successfully loads Facebook `favicon.ico`. It also detects that the user is logged into LinkdedIn through a CSP violation report triggered because of a redirection from `https://fr.linkedin.com` to `https://www.linkedin.com`. All the detection of extensions and logins are invisible to the user.

Chromium-based browsers like Opera [109] and the Brave Browser [16] (we could even detect Brave Browser since it is shipped with several default extensions detectable by WARs). We have chosen to work with Chrome, as it was the most affected.

2.2 Detection of web logins

In general, a website cannot detect whether a user is logged into other websites because of Web browser security mechanisms, such as access control and Same-Origin Policy [125]. In this section, we present two advanced methods that, despite browser security mechanisms, allow an attacker to detect the websites where the user is logged into. Figure 7.2 presents all the detection mechanisms.

Redirection URL hijacking. The first requirement for this method to work is the login redirection mechanism: when a user is not logged into Facebook, and tries to access an internal Facebook resource, she automatically gets redirected to the URL `http://www.facebook.com/login.php?next=[path]`, where `path` is the path to the resource. The second requirement is that the website should have an internal image available to all the users. In the case of Facebook, it is a `favicon.ico` image.

By dynamically embedding an image pointing to `https://www.facebook.com/login.php?next=https%3A%2F%2Fwww.facebook.com%2Ffavicon.ico` into the webpage, an attacker can detect whether the user is logged into Facebook or not. If the image loads, then the user is logged into Facebook, otherwise she is not. This method has been shown to successfully detect logins on dozens of websites [223].

Abusing CSP violation reporting.

An attacker can misuse CSP to detect redirections [206]. We extend this idea to detect logins. For this method to work, a website should redirect its logged in users to a different domain. In the case of LinkedIn, the users, who are not logged in, visit `fr.linkedin.com`,

while the users, who are logged in, are automatically redirected to a different domain `www.linkedin.com`. The lowest block of Fig. 7.2 presents an example of such attack on LinkedIn. Initially, the attacker embeds a hidden iframe from his own domain with the CSP that restricts loading images only from `fr.linkedin.com`. Then, the attacker dynamically embeds a new image on the testing website, pointing to `fr.linkedin.com`. If the user is logged in, LinkedIn will redirect her to the `www.linkedin.com`, and thus the browser will fire a CSP violation report because images can be loaded only from `fr.linkedin.com`. By receiving the CSP report, the attacker deduces that the user is logged in LinkedIn.

3 Dataset

We launched an experiment website in April 2017 to collect browser extensions and Web logins with the goal of studying users' uniqueness at a large scale. We have advertised our experiment by all possible means, including social media and in press. In this section, we first present the set of attributes that we collect in our experiment and the rules we applied to filter out irrelevant records. Then, we provide data statistics and show which extensions and logins are popular among our users.

3.1 Experiment website and data collection

The goal of our website is both to collect browser extensions and Web logins, and to inform users about privacy implications of this particular type of fingerprinting. Using the various detection techniques described in Section 2, we collect the following attributes:

— The list of installed browser extensions, using web accessible resources. For each user we tested around 13k extensions detectable at the moment of testing (see Figure 7.3).

— The list of Web logins: we test for 44 logins using redirection URL hijacking and 16 logins using CSP violation report.

— Standard fingerprinting attributes [221], such as fonts installed, Canvas fingerprint [164], and WebGL [227]. To collect these attributes, we use FingerprintJS2, which is an open-source browser fingerprinting library [199]. We collected these attributes in order to clean our data and compare entropy with other studies (see Table 7.3).

To recognize users that perform several tests on our website, we have stored a unique identifier for each user in the HTML5 localStorage. We have communicated our website via forums and social media channels related to science and technology, and got press coverage in 3 newspapers. We have collected 22,904 experiments performed by 19,814 users between April and August 2017.

Ethical concerns. Our study was validated by an IRB-equivalent service at our institution. All visitors are informed of our goal, and are provided with both Privacy Policy and FAQ sections of the website. The visitors have to explicitly click on a button to trigger the collection of their browser attributes. In our Privacy Policy, we explain what data we are collecting, and give a possibility to opt-out of our experiment. The data collected is used only for our own research, will be held until December 2019 and will not be shared with anyone.

Data cleaning. We applied a set of cleaning rules over our initial data, to improve the quality of the data. The final dataset contains 16,393 valid experiments (one per user). Table 7.1 shows the initial number of users and which users have been removed from our initial dataset. We have removed all 1,042 users with mobile browsers. At the time of writing this thesis, browser extensions were not supported on Chrome for mobiles.

Initial users	**19,814**
Mobile browser users	1,042
Chrome browser users with extension detection error	6
Non Chrome users with at least one extension detected	261
Brave browser users	31
Users whose browser has an empty user-agent string, screen resolution, fonts, or canvas fingerprint	2,015
Users with more than 4 experiments	66
Final dataset	**16,393**
Chrome browser users in the final dataset	7,643

Table 7.1 – Users filtered out of the final dataset

Since extensions detection were designed for Chrome, we then excluded mobile browsers. Moreover, mobile users tend to prefer native apps rather than their web versions [2]. In fact, the popular logins in our dataset, such as Gmail, Facebook, Youtube, all have a native mobile version.
We have also removed 2,015 users that have deliberately tampered with their browsers: for example, users with empty user-agent string, empty screen resolution or canvas fingerprint. We think that it is reasonable not to trust information received from those users, as they may have tampered with it. We also needed this information to compare our study with previous works on browser fingerprinting. Finally, we have excluded users who have tampered with extension detection. This includes Chrome users for whom extension detection did not successfully complete, and users of other browsers with at least 1 extension detected.
For users who visited our website and performed up to 4 experiments, we kept only one experiment, the one with the biggest number of extensions and logins. We then removed 66 users with more than 4 experiments. We suspect that the goal of such users with numerous experiments was just to use our website in order to test the uniqueness of their browsers with different browser settings.

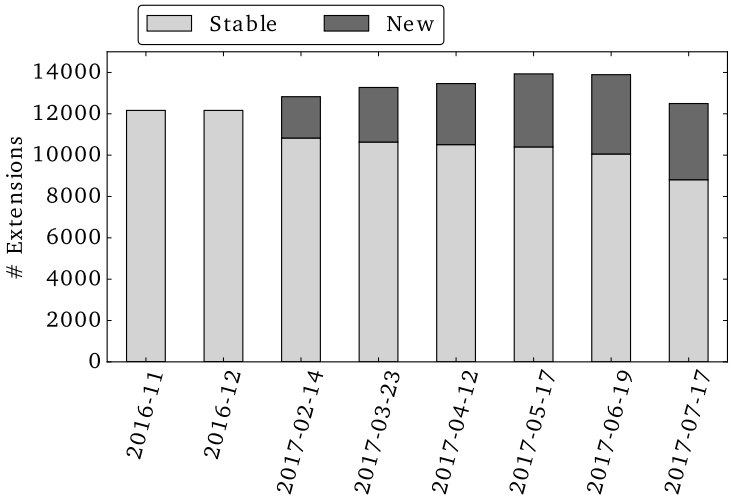


Figure 7.3 – Evolution of detected extensions in Chrome

2. https://jmango360.com/wiki/mobile-app-vs-mobile-website-statistics/

Evolution of browser extensions. From November 2016 to July 2017, we crawled on a monthly basis the free extensions on the Chrome Web Store in order to keep an up-to-date set of extensions for our experiment. Figure 7.3 presents the evolution of extensions throughout the period of our experiment. Since some extensions got removed from the Chrome Web Store, the number of stable extensions decreased.
Out of 12,164 extensions that were detectable in November 2016, 8,810 extensions (72.4%) remained stable throughout the 9-months-long experiment. In total, 16,743 extensions were detected at some point during these 9 months. Since every month the number of detectable extensions was different, on average we have tested around 13k extensions during each month.

3.2 Data statistics

Our study is the first to analyze uniqueness of users based on their browser extensions and logins at large scale. Only uniqueness based on browser extensions was previously measured, but on very small datasets of 204 [245] and 854 [260] participants. We measure uniqueness of 16,393 users for all attributes, and of 7,643 Chrome browser users for browser extensions.
Comparison to previous studies. To compare our findings with the previous works on browser extensions, we randomly pick subsets of 204 (as in [245]) and 854 (as in [260]) Chrome users 100 times (we found that picking 100 times provided a stable result). Table 7.2 shows uniqueness results from previous works and an estimated uniqueness using our dataset.

Table 7.2 – Previous studies on measuring uniqueness based on browser extensions and our estimation of uniqueness.

Study	Fingerprints collected in a study	Extensions targeted in a study	Unique fingerprints in a study	Unique fingerprints in our dataset
Timing leaks [245]	204	2,000	56.86%	55.64%
XHOUND [260]	854	1,656	14.10%	49.60%
Ours	7,643	13k	39.29%	39.29%

The last column in Table 7.2 shows our evaluation of uniqueness for a given subset of users. Our estimation for 204 random users is 55.64%, which is close to the 56.86% from the original study [245]. For 854 random users, we estimate that 49.60% of them are unique, while in the original XHOUND study [260] the percentage of unique users is only 14.1%. We think that such small percentage of unique users in [260] is due to (1) a smaller number of extensions detected (only 174 extensions were detected for 854 users); (2) a different user base: while our experiments and [245] targeted colleagues, students and other likely privacy-aware experts, XHOUND [260] used Amazon Mechanical Turk, where users probably have different habits to installing extensions. Out of 7,643 users of the Chrome browser, where we detected extensions, 39.29% of users were unique. This number shows a more realistic estimation of users' uniqueness based on browser extensions than previous works because of a significantly larger dataset.
To the best of our knowledge, our study is the first to `analyze uniqueness of users based on their web logins`, and on combination of extensions and logins.

Normalized Shannon's entropy. We compare our dataset with the previous studies on browser fingerprinting: AmIUnique [218, Table B.3] (contains 390,410 fingerprints, collected between November 2014 and June 2017) and Hiding in the Crowd [191] (contains 1,816,776 users collected in 2017). Entropy measures the amount of identifying information in a fingerprint – the higher the entropy is, the more unique and identifiable a fingerprint will be. To compare with previous datasets, which are of different sizes, we compute normalized Shannon's entropy:

$$H_N(X) = \frac{H(X)}{\log_2 N} = -\frac{1}{\log_2 N} \cdot \sum_i P(x_i) \log_2 P(x_i) \tag{7.1}$$

where X is a discrete random variable with possible values $\{x_1, ..., x_n\}$, $P(X)$ is a probability mass function and N is the size of the dataset.
Table 7.3 compares the entropy values of well-known attributes for standard fingerprinting and for logins for all 16,393 users in our dataset and for browser extensions for 7,643 Chrome users. All the attributes in standard fingerprinting are similar to previous works, except for fonts and plugins. Unsurprisingly, plugins entropy is very small because of decreasing support of plugins in Firefox [237] and Chrome [246]. Differently from previous studies that detected fonts with Flash, we used JavaScript based font detection, relying on a list of 500 fonts shipped along with the FingerprintJS2 library. As those fonts are selected for fingerprinting, this could explain why our list of fonts provides a very high entropy.

Table 7.3 – Normalized entropy of extensions and logins compared to previous studies.

Standard fingerprinting studies			
Attribute	Ours	AmIUnique [218]	Hiding [191] Desktop
User Agent	0.474	0.601	0.304
List of Plugins	0.343	0.523	0.494
Timezone	0.168	0.187	0.005
Screen Resolution	0.271	0.276	0.213
List of Fonts	0.652	0.370	0.335
Canvas	0.611	0.503	0.387
Studies on extensions and logins			
Attribute	Ours	Timing leaks [245]	XHOUND [260]
Extensions	0.641	0.869	0.437
Logins	0.441	N/A	N/A

In our dataset, as well as in previous studies, browser extensions are one of the most discriminating attributes of a user's browser. The computed entropy of 0.641, computed for the 7,643 Chrome users, lays between the findings of Timing leaks [245] and XHOUND [260]. One possible explanation is the size of the user base. For instance, users in XHOUND had few and probably often the same extensions detected (out of 1,656 targeted extensions, only 174 were detected for 854 users), making only 14.1% of them unique. This explains why the entropy in XHOUND is smaller. Sánchez-Rola et al. [245] computed a very high entropy, but on a very small dataset of 204 users: 116 of them had a unique set of installed extensions, and thus the computed entropy was very high.

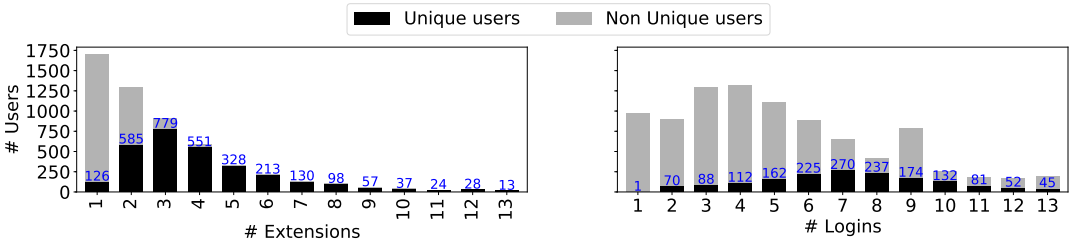


Figure 7.4 – Usage of browser extensions and logins by all users.

3.3 Usage of extensions and logins

Figure 7.4 shows the distribution of users in our dataset according to the number of detected extensions and logins (users having between 1 and 13 logins or extensions detected), and the number of unique users as they are grouped by number of detected extensions and logins. The maximum number of extensions we detected for a single user was 33. The number of users decreases with the number of extensions. The largest group of users have only 1 extension detected, followed by users with 2 detected extensions, etc. We notice that the more extensions a user has, the more unique she is. We analyze this phenomenon further in Section 4.2. Among users with exactly 1 extension detected, 7.39% are unique. This percentage rises to 45.35% and 85.89% for groups of users with exactly 2 and 3 detected extensions respectively.
Figure 7.4 also shows the distribution of users per number of detected logins. We found that most users have between 1 and 10 logins, with a maximum number of 40 logins detected for one user. On our website, we were able to detect the presence of 60 logins, which is rather small with respect to the large number of extensions we tested (around 13k per user). This explains why fewer users are unique based on their logins: for example, among users with exactly 1 login detected, 0.10% are unique, and 7.82% are unique among users with exactly 2 logins detected.

Table 7.4 – Top seven most popular extensions in our dataset and their popularity on Chrome Web Store

Extension	**Dataset**	**Chrome**
AdBlock	1,557	10,000,000+
LastPass: Free Password Manager	1,081	7,297,730
Ghostery	735	2,665,427
Privacy Badger	594	771,804
Adobe Acrobat	585	10,000,000+
Cisco WebEx Extension	482	10,000,000+
Save to Pocket	428	2,752,642

What extensions are the most popular among our users? Table 7.4 presents the seven most detected extensions in our dataset of 16,393 users. The three most popular extensions are AdBlock [6], password manager LastPass [86] and tracker blocker Ghostery [61]. These extensions are also very popular according to their downloads statistics on Chrome Web Store.
What websites users are logging into the most? Table 7.5 shows the seven most detected websites in our experiment. These websites are also highly rated according to

Table 7.5 – Top seven most popular logins in our dataset and their ranking according to Alexa

Website	Dataset	Alexa Rank
Gmail (subdomain of Google)	6,828	1
Youtube	6,780	2
Facebook	5,493	3
LinkedIn	3,913	13
Blogger	3,393	53
Twitter	3,274	8
eBay.com	2,220	33

Alexa[3]. For instance, Google [65], Facebook [53] and Youtube [157] are regularly ranked as the top 3 most popular websites by Alexa[4]. Being able to detect such popular websites further strengthen our study as they represent websites that are widely used by users in the wild.

4 Uniqueness analysis

In this section we present the results for user's uniqueness based on all 16,743 extensions and 60 logins. We first show uniqueness for the full dataset of 16,393 users, and then present more specific results for various subsets of our dataset.

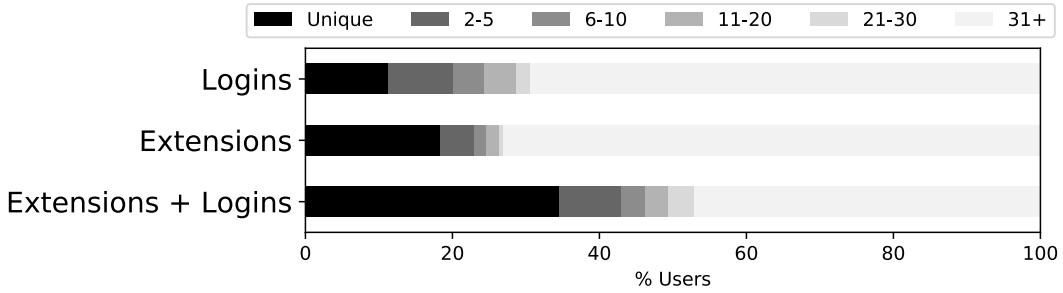


Figure 7.5 – Distribution of anonymity set sizes for 16,393 users based on detected extensions and logins.

Uniqueness results for the full dataset. Figure 7.5 shows the uniqueness of users according to their extensions and logins, and a combination of both attributes. Out of the 16,393 users, 11.30% are unique based on their logins. For 42.1% of users in our dataset, we did not detect any logins. These users either did not log into any of the 60 websites we could detect or blocked third party cookies, that prevented our login detection from working properly.
Considering only detected extensions, 18.38% of users in our dataset are unique. This result is also influenced by the 66.61% of users who did not have any extension detected: these are either Chrome users with no extensions detected, or users of other browsers.
An attacker willing to fingerprint users can also use their detected logins and extensions combined. Interestingly, by combining extensions and logins, we found that 34.51% of users

3. Alexa ranking extracted on the the 28th of June 2018
4. Note that Gmail is a subdomain of Google, that is why it is ranked 1 in Table 7.5.

are uniquely identifiable. It is worth mentioning that 32.61% of users have no extensions and no logins detected. This impacts significantly the computed uniqueness.

4.1 Four final datasets

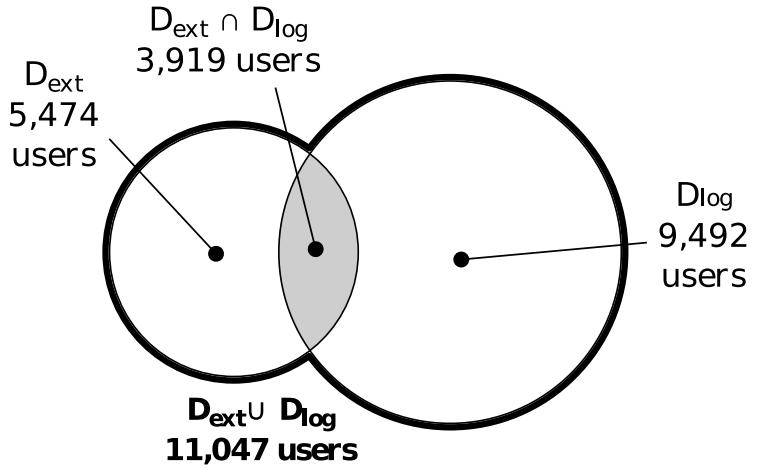


Figure 7.6 – Four final datasets. D_{Ext} contains users, who have installed at least one detected extension and D_{Log} contains users, who have at least one login detected.

In our full dataset of 16,393 users, we have observed 7,643 users of Chrome browser, for whom testing of browser extensions worked properly. In this subsection we consider various subsets of our full dataset that demonstrate uniqueness results for users who have at least one extension or one login detected. Figure 7.6 shows four final datasets that we further analyze in this section:

— D_{Ext} contains 5,474 Chrome users, who have installed at least one extension that we can detect.

— D_{Log} contains 9,492 users, who have logged into at least one website that we detect.

— $D_{Ext} \cap D_{Log}$ contains 3,919 Chrome users who have at least one extension and one login detected.

— $D_{Ext} \cup D_{Log}$ contains 11,047 users who have either at least one extension or at least one login detected.

4.2 Uniqueness results for final datasets

Figure 7.7 presents results for the four datasets. D_{Ext} dataset shows that 54.86% of users are uniquely identifiable among Chrome users, who have at least one detectable extension. This demonstrates that browser extensions detection is a serious privacy threat as a fingerprinting technique.
Among 9,492 users with at least one login detected (D_{Log} dataset) only 19.53% are uniquely identifiable. This result can be explained by a very small diversity of attributes (only 60 websites).
When we analyzed Chrome users who have at least one extension and one login detected ($D_{Ext} \cap D_{Log}$ dataset), we found out that 89.23% of them are uniquely identifiable. This means that without any other fingerprinting attributes, the mere installation of at least one extension, in addition to being logged into at least one website imply that the majority

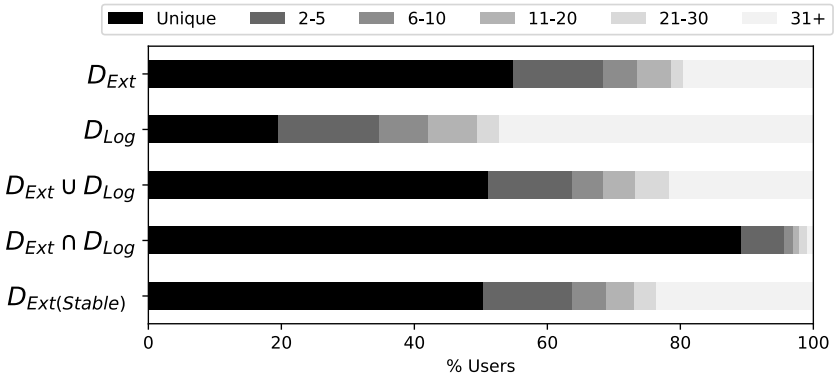


Figure 7.7 – Anonymity sets for different datasets

of users in this dataset can be tracked by their fingerprint based solely on extensions and logins!

Furthermore, for dataset $D_{Ext} \cup D_{Log}$ that contains users with at least one extension or at least one login, we compute that 51.15% of users can be uniquely identified. This result becomes particularly interesting when we compare the size of the $D_{Ext} \cup D_{Log}$ dataset, which contains 11,047 users, with the size of the D_{Ext} dataset, that has 5,474 users. The size of $D_{Ext} \cup D_{Log}$ is almost twice as large as D_{Ext}. Nevertheless, the percentage of unique users and the distribution of anonymity set sizes in these datasets are very similar: 54.86% of unique users in D_{Ext} and 51.15% of unique users in $D_{Ext} \cup D_{Log}$. We believe this is due to the fact that extensions and logins are orthogonal properties. We checked the cosine similarity between these attributes as binary vectors, and found that all attribute pairs had a very low similarity score, all below 0.34, with 11 exceptions below 0.2.

The last row $D_{Ext(Stable)}$ shows uniqueness of users in the D_{Ext} dataset, but considering only stable extensions (see more details in Section 3). Interestingly, 50.35% of users are uniquely identifiable with their stable extensions only and the distribution of anonymity set sizes is very similar too. This result shows that browser extensions that were added or removed throughout the 9-months-long experiment do not influence the result of users' uniqueness.

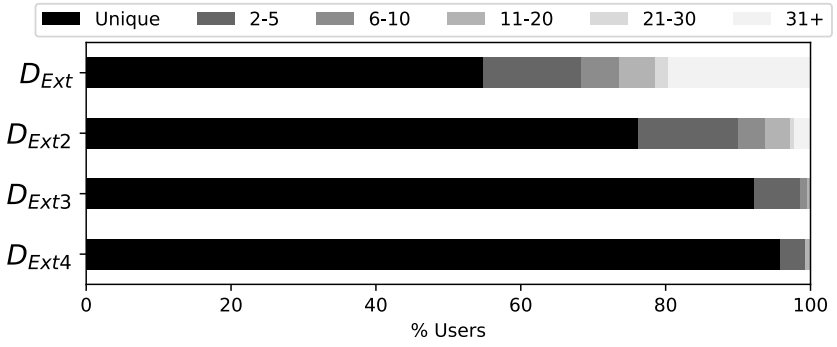


Figure 7.8 – Anonymity sets for users with respect to the number of detected extensions

The more extensions you install, the more unique you are. In the beginning of this section, we have shown that 54.86% of users are unique among those who have at least one extension detected (D_{Ext} dataset). Figure 7.8 shows how uniquely identifiable users are when they have more extensions detected. Among users with at least two extensions

detected, 76.25% are uniquely identifiable. This percentage rises quickly to 92.22% and 95.85% when we consider users with at least three and four extensions detected respectively. We made a similar analysis for logins: likewise, the percentage of unique users grows if we consider users with a higher number of detected logins. 31.58% users with at least 5 logins are uniquely identifiable with their logins only. This percentage rises to 38.98% when we detected at least 8 logins. Intuitively, the more extensions or logins a user has, the more unique he becomes. It is worth mentioning that the subsets of users considered decreases as we increase the number of extensions or logins detected, as shown in Figure 7.4.

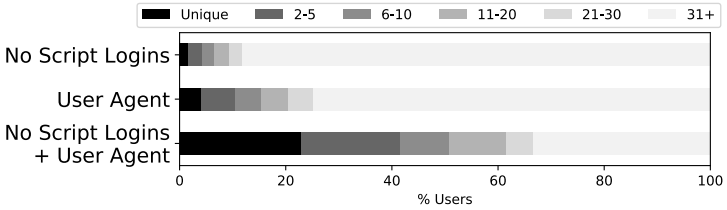


Figure 7.9 – Anonymity sets when JavaScript is disabled

Uniqueness if JavaScript is disabled. Users might decide to protect themselves from fingerprinting by disabling JavaScript in their browsers. However even when JavaScript is disabled, detection of logins via a CSP violation attack still works. Among 60 websites in our experiment, we discovered that such an attack works for 18 websites. Figure 7.9 shows anonymity sets for 9,492 users of D_{Log} dataset assuming users have disabled JavaScript. By considering only logins detectable with CSP, 1.63% of users are uniquely identifiable, and 4.10% are unique based on a user agent string that is sent with every request by the browser. However, when we combine the user agent string with the list of logins detectable with CSP, 22.98% of users become uniquely identifiable.

5 Fingerprinting attacks

According to the uniqueness analysis from Section 4, 54.86% of users that have installed at least one detectable extension are unique; 19.53% of users are unique among those who have logged into one or more detectable websites; and 89.23% are unique among users with at least one extension and one login. Therefore, extensions and logins can be used to track users across websites. In this section we present the threat model, discuss and evaluate two algorithms that optimize fingerprinting based on extensions and logins.

5.1 Threat model

The primary attacker is an entity that wishes to uniquely identify a user's browser across websites. An attacker is recognizing the user by his `browser fingerprint`, a unique set of detected browser extensions and Web logins (we call them `attributes`), without relying on cookies or other stateful information. A single JavaScript library that is embedded on a visited webpage can check what extensions and Web logins are present in the user's browser. By doing so, an attacker is able to uniquely identify the user and track her activities across all websites where the attacker's code is present. We assume that an attacker has a dataset of users' fingerprints, either previously collected by the attacker or bought from data brokers.

5.2 How to choose optimal attributes?

The most straightforward way to track a user via browser fingerprinting is to check all the attributes (browser extensions and logins) of her browser. However, testing all 13k extensions takes around 30 seconds[5] and thus may be unfeasible in practice. Therefore, the *number of tested attributes* is one of the most important property of fingerprinting attacks – the attack is faster when fewer attributes are checked. But testing fewer attributes may lead to worse uniqueness results, because more users will share the same fingerprint.
While it was shown that finding the optimal fingerprint is an NP-hard problem [197], finding approximate solutions is neither a trivial task. For example, choosing the most popular attributes worked in the case of tracking based on Web history, but this strategy is not necessarily the globally optimal case.
Following the theoretical results of Gulyás et al. [197], we consider these two strategies: (1) to target a specific user, and thus to select attributes that makes her unique with high probability – called `targeted fingerprinting` algorithm, and (2) to uniquely identify a majority of users in a dataset, and thus select the same set of attributes for all users – we call it `general fingerprinting` algorithm. Targeted fingerprinting mainly uses popular attributes if they are not detectable (e.g., popular extensions are not installed) or unpopular ones if they are detectable. General fingerprinting instead, considers attributes that are detectable roughly at half of the population (this allows to chose more independent attributes which makes a fingerprint based on these attributes more unique).
Using the algorithms developed in [197], we performed experiments with general and targeted fingerprinting. Our goal is to achieve results close to those in Section 4, but by testing a smaller number of attributes[6].

5.3 Targeted fingerprinting

Attack outline. The attacker aims to identify a specific user with high probability. In order to do this, the attacker needs to have information about the targeted user in her dataset of fingerprints. The attacker generates a `fingerprint pattern` that consists of a list of attributes with a known value, such as f_j = [AdBlock=yes, LastPass=No, ...]. Notice that a fingerprint pattern contains not only extensions that the user installed, but also extensions that are not installed. This information also helps to uniquely identify the user.
Let us denote the user database as D of n users and m attributes, each row i corresponding to user U_i and each column j corresponding to attribute A_j. Let the algorithm target user U_i. First, we need to find her most distinguishing attribute A_j, shared among the smallest number of other users. Let us denote these users as $S_{i,j}$. Then we need to find a second most distinguishing property A_k which separates U_i from $S_{i,j}$. Then the algorithm continues searching for the most distinguishing attributes, until the given pattern makes U_i unique (or there are no more acceptable choices left).
Evaluation. We applied targeted fingerprinting algorithm [197] on our datasets D_{Ext}, D_{Log}, $D_{Ext} \cap D_{Log}$ and $D_{Ext} \cup D_{Log}$, and computed a fingerprint pattern for each user. By using these patterns, we have computed the anonymity sets for all datasets, that are identical to those shown in Figure 7.7. We therefore omit repeating these results in a new figure.
For each unique user, the fingerprint pattern contains a smaller number of attributes than the number of attributes detected for the user. For example, it is enough to test only 2 extensions for a user who has installed 4 detectable extensions. Figure 7.10 shows the

5. We evaluate performance in Section 6.
6. We reused the implementation of Gulyás et al., who shared their code [196].

distribution of fingerprint pattern sizes of unique users (marked with "targeted"), and compares them to the number of attributes detected for each user. The figure clearly shows that fingerprint patterns are typically smaller than the number of detected attributes users have.

For non-unique users, the size of the fingerprint pattern is often bigger than the number of detected attributes the user has. Let us discuss this on our largest dataset $D_{Ext} \cup D_{Log}$, but note that other datasets exhibit the same phenomena. For unique users, on average we have 7.94 attributes detected, while the average size of fingerprint pattern is 3.94 attributes only. For non-unique users, the average number of detected attributes is 5.41, while the average size of fingerprint pattern grew up to 30.17. This result is not surprising: with less information it is more difficult to distinguish users, and the fingerprint pattern may also include negative attributes (i.e., LastPass=No means an extension should not be detected), which can extend the length greatly.

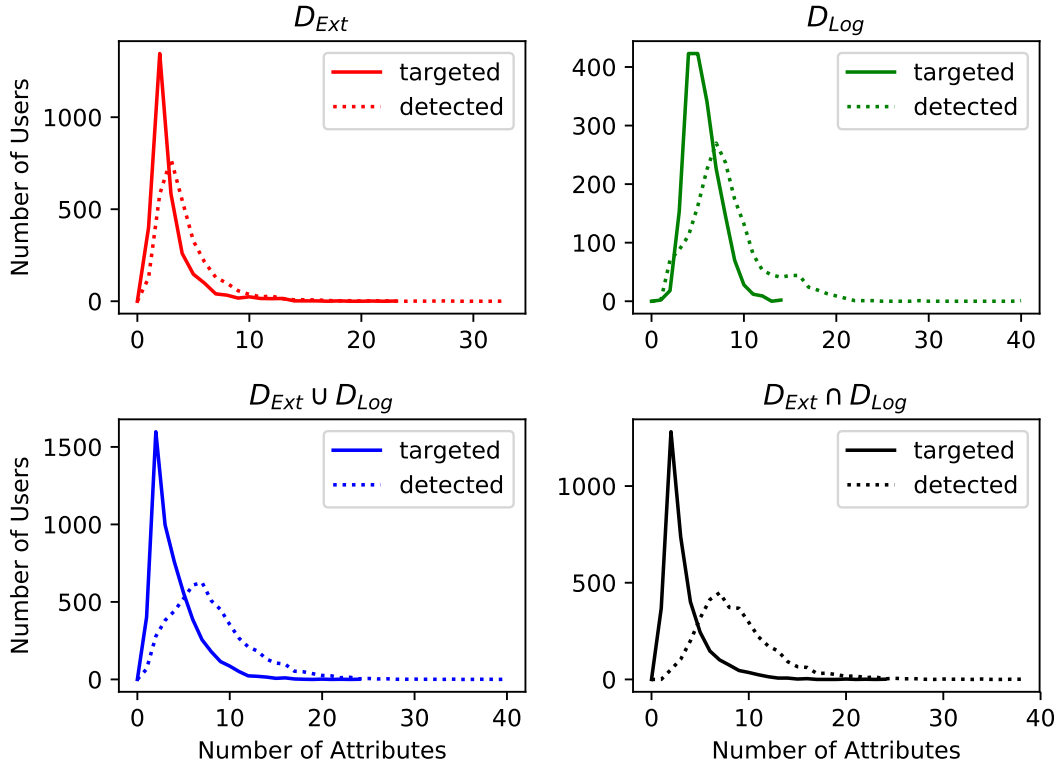


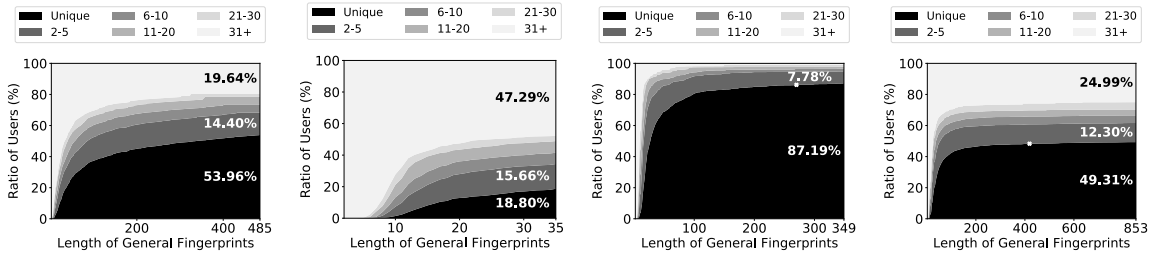
Figure 7.10 – Comparison of fingerprint pattern size (targeted) and the total number of detected attributes (detected) for unique users.

The targeted fingerprint is efficient, as it provides almost maximal uniqueness while reducing the number of attributes. However, it cannot be used for new users, because the attacker does not have any background knowledge about them. To reach a wider usability with a trade-off in the fingerprint pattern size, we also consider general fingerprinting [197].

5.4 General fingerprinting

Attack outline. The purpose of this algorithm is to provide a short list of attributes, called `fingerprint template`. If the attributes in a fingerprint template are tested for a certain user, she will be uniquely identified with high probability. Similarly to the example

of targeted fingerprinting, we consider the fingerprint template F = [AdBlock, LastPass, ...], that would yield the fingerprint f_j^F=[yes, no, ...] for the user U_j.
The algorithm first groups all users into a set S. Then it looks for an attribute A_i that will separate S into roughly equally sized subsets S_1 and S_2 if we group users based on their attribute A_i. In the next round, it looks for another $A_j \neq A_i$ that splits S_1, S_2 further into roughly equally sized sets. This step is repeated until we run out of applicable attributes or the remaining sets could not be sliced further.



(a) D_{Ext} - 5,474 users (b) D_{Log} - 9,492 users (c) $D_{Ext} \cap D_{Log}$ - 3,919 users (d) $D_{Ext} \cup D_{Log}$ - 11,047 users

Figure 7.11 – Anonymity sets for different numbers of attributes tested by general fingerprinting algorithm.

Evaluation. To apply general fingerprinting, we first measure uniqueness by using all attributes, which will be our target level A. Then, we run the algorithm until either it stops by itself (e.g., fingerprint cannot be extended further), or we terminate it early when the actual level of uniqueness B is less than 1% from level A.
Figure 7.11 shows the anonymity sets for different fingerprint lengths for our datasets D_{Ext}, D_{Log}, $D_{Ext} \cap D_{Log}$ and $D_{Ext} \cup D_{Log}$, generated by the general fingerprinting algorithm. For D_{Ext} and D_{Log}, the algorithm provided fingerprint templates of 485 extensions and 35 logins. In these cases the algorithm stopped since no more attributes could be used for achieving better uniqueness – hence the final anonymity sets are very close to those in Figure 7.7. In the cases of $D_{Ext} \cap D_{Log}$ and $D_{Ext} \cup D_{Log}$, we observed slow convergence in uniqueness, thus we could stop the algorithm earlier (shown as white dots in Figure 7.11). As a result, for $D_{Ext} \cap D_{Log}$, we can obtain 86.19% of unique users by testing 270 extensions and logins. For $D_{Ext} \cup D_{Log}$, we can obtain 48.31% of unique users by testing 419 extensions and logins.
We conclude that the general fingerprint can achieve a significant decrease in the fingerprint length while maintaining the level of uniqueness almost at maximum. In the next section we discuss the performance of these results.
For D_{Ext} dataset, general fingerprinting algorithm provides 485 extensions, but we found out that 20 of these extensions were not stable (see Figure 7.3) and were not present in the last month of our experiment. Using all extensions, including unstable ones, can be useful to maintain fingerprint comparability with older data or with users having older versions of extensions. However, if we constrain general fingerprinting to stable extensions only, we get a fingerprint template of 465 extensions, leading to 50.33% uniqueness – still very close to the results of baseline uniqueness results, which was 50.35% with stable extensions only.

6 Implementation and performance

In this section we discuss the design choices we made for our experimental website and analyze whether browser extensions and Web logins fingerprinting is efficient enough to be used by tracking companies.

To collect extensions installed in the user's browser, we first needed to collect the extensions' signatures from the Chrome Web Store. We collected 12,497 extensions in August 2017, using the code shared by Sjösten et al. [249]. To detect whether an extension was installed, we tested only one WAR per extension (see more details on WARs in Section 2). Because the extensions' signatures size was 40Mb and could take a lot of time to load on the client side, we reorganized and compressed them to 600kb. However, testing all the 12,497 extensions at once took 11.3–12.5 seconds and was freezing the UI of a Chrome browser. To avoid freezing, we split all the extensions in batches of 200 extensions, and testing all the 12,497 extensions ran in approximately 30s.

Since testing all the extensions takes too long, trackers may not be using this technique in practice. Therefore, we measured how much time it takes to apply the optimized fingerprinting algorithms from Section 5. Targeted fingerprinting addresses each user separately, hence the number of tested extensions differs a lot from user to user. General fingerprinting instead provides a generic optimization for all users. Based on our results from Section 5, an attacker can test 485 extensions and obtain the same uniqueness results as with testing all 12,497 extensions. Such testing can be run in 625 milliseconds with the signature file size below 25Kb, which make real-life tracking feasible. For websites with limited traffic volumes, extension detection alone could be used for tracking, or for websites with a higher traffic load, it could contribute supplementary information for fingerprinting. Regarding targeted fingerprinting the attacker can do even better, as such short patterns can be detected in less than 10 milliseconds.

Compared to extension detection, Web login detection methods depend on more external factors (such as network speed and how fast websites respond), thus they should be used with caution. For redirection URL hijacking detection, we observed that the majority of Web logins can be detected in 0.9–2.0 seconds, however the timing was much harder to measure for the method based on CSP violation report. We observed that if the network was overloaded and requests were delayed, then the results of login detection were not reliable; however, it is likely that unreliable results can be easily discarded by checking timings of results (e.g., large delays appearing only in few cases).

Moreover, we found a bug in the CSP reporting implementation in the Chrome browser that makes this kind of detection even more difficult. In fact, without a system reboot for more than a couple of days (we observed that this varies between one day to multiple weeks), the browser stopped sending CSP reports. We reported the issue to Chrome developers, as this bug not only makes CSP-based detection unreliable, but more importantly CSP itself.

7 The dilemma of privacy extensions

Various extensions exist that block advertisement content, such as AdBlock [6], or block content that tracks users, such as Disconnect [47]. Such extensions undoubtedly protect users' privacy, but if they are easily detectable on an arbitrary webpage, then they can contribute to users' fingerprint and can be used to track the user across websites. In our experiment based on detecting extensions via WARs, we could detect four privacy extensions: AdBlock [6], Disconnect [47], Ghostery [61] and Privacy Badger [117]. The goal of this section is to analyze the tradeoff between the privacy loss (how fingerprintable

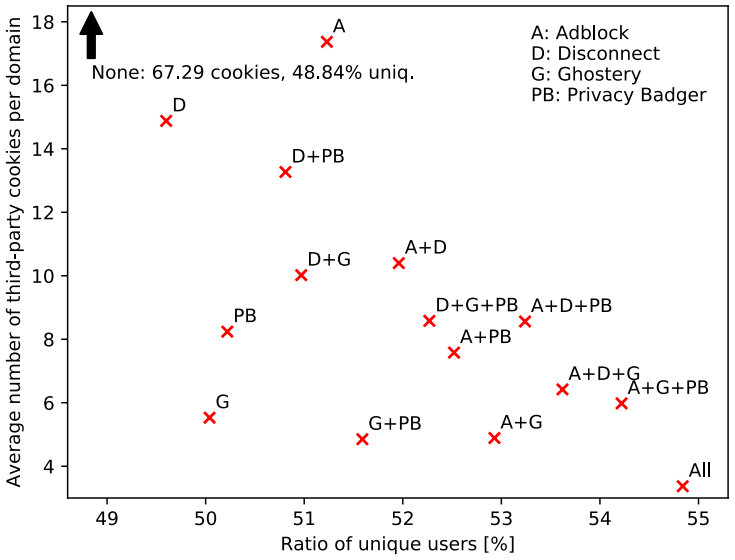


Figure 7.12 – Uniqueness of users vs. number of unblocked third-party cookies

users with such extensions are) and the level of protection provided by these extensions. To understand this tradeoff, we computed (i) how unique are the users who install privacy extensions; (ii) how many third-party cookies are stored in the user's browser when a privacy extension is activated (the smaller the number of third-party cookies, the better the privacy protection). We analyzed four privacy extensions detectable by WARs and 16 combinations of these extensions: AdBlock [6], Disconnect [47], Ghostery [61] and Privacy Badger [117].

First, we measured how a combination of privacy extensions contributes to fingerprinting. To measure uniqueness of users for a combination of extensions (i.e., AdBlock+Ghostery), we removed other privacy extensions from the D_{Ext} dataset [7] (i.e., Disconnect and Privacy Badger), and then evaluated the percentage of unique users for each combination.

Second, we measured how many third-party cookies were set in the browser, even if privacy extensions were enabled. We performed an experiment, where for each combination of extensions, we crawled the top 1,000 Alexa domains, visiting homepage and 4 additional pages in each domain [8]. We kept the browsing profile while visiting pages in the same domain, and used a fresh profile when we visited a new domain. We explicitly activated Ghostery, which is deactivated by default, and trained Privacy Badger on homepages of 1,000 domains before performing our experiment. We collected all the third-party cookies that remained in the user's browser for each setting and divided it by the number of domains crawled.

Figure 7.12 reports on the average number of cookies that remained in the browser for each combination of extensions, and the corresponding percentage of unique users.

Similarly to the results of Merzdovnik et al. [226], Ghostery blocks most of the third-party cookies, and the least blocking extension is AdBlock. Surprisingly, some combinations such as Disconnect + Ghostery resulted in more third-party cookies being set than for Ghostery

7. The total number of users does not change since we simply remove certain extensions from the user's record in our dataset.

8. We have extracted the first 4 links on the page that refers to the same domain.

alone – even after double checking the settings, and re-running the measurements, we do not have an explanation for this phenomena. However, as this can have a serious counter-intuitive effect on user privacy, it would be important to investigate this in future work.
More privacy extensions indeed increase user's unicity. All of these privacy extensions are also part of the general fingerprint we calculated in Section 5.4. However, this has little importance in practice. If we ban the general fingerprint algorithm from using privacy extensions, it will generate a fingerprint template of 531 (instead of 485) extensions, leading to a uniqueness level of 51.27%. While 46 is a significant increase in the number of extensions for fingerprinting, as we have seen it already, this would only contribute very little to the overall timing of the attack.
On the other hand, as this experiment revealed, these extensions are also very useful to block trackers. We could therefore conclude that using Ghostery is a good trade-off between blocking trackers and avoiding extension-based tracking. However, in order to efficiently solve the trade-off dilemma, we believe that such functionality should be included by default in all browsers.

8 Countermeasures

We provide recommendations for users who want to be protected from extensions- and logins-based fingerprinting. We also provide to developers recommendations to improve browser and extensions architecture in order to reduce the privacy risk for their users.
Countermeasures for extension detection. Extension detection method based on Web Accessible Resources detects 28% of Google Chrome extensions, while for Firefox the number is much smaller: 6.73% of extensions are detectable by WARs [249]. Firefox gives a good example of browser architecture that makes extensions detection difficult. The upcoming Firefox extensions API, WebExtensions, which is compatible with Chrome extensions API [25], is designed to prevent extensions fingerprinting based on WARs: each extension is assigned a new random identifier for each user who installs the extension [152]. To protect the users, developers of Chrome extensions could avoid Web Accessible Resources by hosting them on an external server, however this could lead to potential privacy and security problems [249]. Developers of the Chrome browser could nonetheless improve the privacy of their users by adopting the random identifiers for extensions as in WebExtensions API.
Most of the browsers are vulnerable to extension detection, and websites could also detect extensions by their behavior [260]. Therefore today users cannot protect themselves completely, but they still can minimize the risk by using browsers such as Firefox, where a smaller fraction of extensions are detectable.
Countermeasures for login. Users may opt for tracker-blocking and adblocking extensions, such as Ghostery [61], Disconnect [47] or AdBlockPlus [7]. But these extensions block requests to well-known trackers, while Web logins detection sends requests to completely legitimate websites, where the user has logged into anyway. Another option is to install extensions that block cookies arriving from unknown or undesirable domains. These extensions do not protect users for the same reason: cookies that belong to websites that the user visits (and treated as first-party cookies) are the same cookies used for login detection (with the only difference that the same cookies are treated as third-party cookies). For example social websites, such as Facebook or Twitter, use first-party cookies. Their social button widgets with third-party cookies may still be allowed by the browser extensions in the context of other websites. Therefore, users can protect themselves from Web logins detection, only by `disabling third-party cookies` in their browsers.

Website owners could also react to such potential privacy risk for their users. In our case, this would simply mean filtering login URL redirection, and sanity checking other redirection mechanisms against the CSP-based attack. Unfortunately, this issue has been known for a while, but website owners do not patch it because they do not consider this as a serious privacy risk [223].
Browser vendors could help avoid login detection by blocking third-party cookies by default. The new intelligent tracking protection of the Safari browser takes a step in the right direction, as it blocks access to third-party cookies and deletes them after a while.

9 Discussion and future work

Realistic datasets. To compare our study with previous works on fingerprinting by browser extensions, we analyzed different random subsets of 7,643 users, who run Chrome web browser (where browser extension detection is possible in our experiment). Figure 7.13 shows how user uniqueness based on extensions changes with respect to the various subsets of our dataset. It clearly demonstrates an intuition that the smaller the user set is, the smaller is the diversity of users, and the easier it is to uniquely identify them.

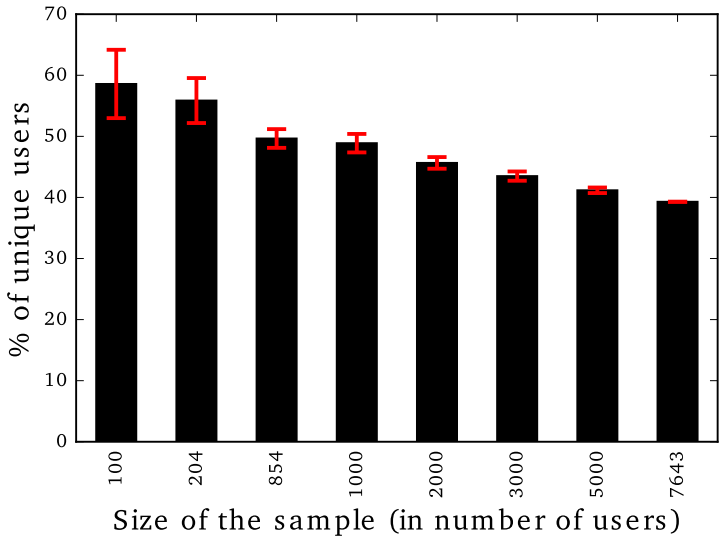


Figure 7.13 – Uniqueness of Chrome users based on their extensions only vs. number of users - 204 is the number of users used in [245] and 854 the number of users considered in [260]

Figure 7.13 compares our results to previous studies on browser extensions fingerprinting: we have 7,643 Chrome users, while previous studies had 204 [245] and 854 [260] users, and therefore draw different conclusions about uniqueness of users based on browser extensions. We reported on the number of unique users in subsets of 204 and 854 users in Section 3.2 (see Table 7.2). By exploring this comparison, we raise a fundamental question: `What is the ‘‘right’’ size for the dataset?`
Taking a look at research on standard fingerprinting, in 2010 Eckersley showed that 95% of browsers were unique based on their properties [185], which was backed by several papers since then [173,221]. However, a recent study states that by looking at 2 million fingerprints in 2018, the authors only found 33.6% of those fingerprints to be unique [191].
It is extremely difficult for computer scientists to get access to such large datasets – in our

experience, we advertised our experiment website through all possible channels, including Twitter, Reddit, and press coverage. We experienced that having larger, high quality datasets is a highly nontrivial research task. It is important to re-evaluate our results over time while also aiming to obtain larger dataset sizes.

Stability of fingerprints While studying uniqueness based on various behavioral features, it is very important to know how stable these features are, as the ability to use some of this information as part of a fingerprint does not solely depend on its anonymity set of overall entropy, but also on the information stability (i.e., how frequently it changes over time). Vastel et al. [266] recently analyzed the evolution of fingerprints of 1,905 browsers over two years. They concluded that fingerprints' evolution strongly depends on the type of the device (laptop vs mobile) and how it is used. Overall, they observed that 50% of browsers changed their fingerprints in less than 5 days.

In our study we did not have enough data to make any claims about the stability of the browser extensions and web logins because only few users repeated an experiment on our website (to be precise, only 66 users out of 16,393 users have made more than 4 tests on our website). We would expect that browser extensions are more stable than logins since users do not seem to change extensions very often, while they may log in and log out of various websites during the day. However, studying the stability of extensions and logins would require all our users to install a tool (probably a browser extension) in their browsers that would monitor the extensions they install and logins they perform. This kind of experiment would be even harder to perform at large scale since users do not easily trust to install new browser extensions. In AmIUnique experiment, Laperdrix [218] was trying to measure stability of browser fingerprints – he collected data from 3,528 devices over a twenty-month-long experiment. We managed to have 16,393 users testing our website in 9 months. This shows that users have more trust in testing their browser on a website than installing new extensions.

We therefore keep the study of fingerprints stability for future work and raise an important question in privacy measurement community: `How can we ensure a large scale coverage of users for our privacy measurement experiments?`

10 Conclusion

This chapter reports on a large-scale study of a new form of browser fingerprinting technique based on browser extensions and website logins. The results show that 18.38% of users are unique because of the extensions they install (54.86% of users that have installed at least one detectable extension are unique); 11.30% of users are unique because of the websites they are logged into (19.53% are unique among those who have logged into one or more detectable websites); and 34.51% of users are unique when combining their detected extensions and logins (89.23% are unique among users with at least one extension and one login). It also shows that the fingerprinting techniques can be optimized and performed in 625 ms.

This work illustrates, one more time, that user anonymity is very challenging on the Web. Users are unique in many different ways in the real life and on the Web. For example, it has been shown that users are unique in the way they browse the Web, the way they move their mouse or by the applications they install on their device [201]. This work shows that users are also unique in the way they configure and augment their browser, and by the sites they connect to. Unfortunately, although uniqueness is valuable in society because it increases diversity, it can be misused by malicious websites to fingerprint users and can therefore hurt privacy.

Another important contribution of this work is the definition and the study of the trade-off that exists when a user decides to install a "privacy" extension, for example, an extension that blocks trackers. This work shows that some of these extensions increase user's unicity and can therefore contribute to fingerprinting, which is counter-productive. We argue that these "privacy" extensions are very useful, but they should be included by default in all browsers. "Privacy by default", as advocated by the new EU privacy regulation, should be enforced to improve privacy of all Web users.

Part III

Browser Extensions

Introduction

The concept of browser extensions or addons, regardless of the underlying architecture or browser, always exhibits very common characteristics: they are third party codes that execute in browsers with elevated privileges, giving them access to features and user data that traditional web applications for instance cannot directly access. In the past, different browsers have supported different systems for extensions development. Egele et al. [186] applied a dynamic analysis system to detect spyware in the Internet Explorer Browser Helper Objects (BHOs). Authors had also shown the dangers of misusing the powerful APIs provided to Firefox XPCOM extensions and proposed tools for discovering vulnerabilities and securing extensions [170, 224, 232, 233]. Barth et al. [172] analyzed the Firefox XPCOM architecture and proposed a new extensions architecture that has since been adopted by Google Chrome and evolved into the Chrome Extensions API compatible with the cross-browser WebExtensions API. Among other things, the permissions system in extensions was meant to reduce extensions capabilities, and hence reduce the harms that attackers can cause if they compromise an extension. However a good number of studies have shown that many extensions still request too many permissions [195, 202, 213].

The WebExtensions API [100] has been introduced by Firefox as a cross-browser platform for developing extensions that run on many browsers. To a large extent, it is compatible with the Chrome extension API [25], Opera Extension API [110] and Microsoft Edge Extension API [2]. Carlini et al. [181] reviewed 100 Chrome extensions and found many vulnerabilities due to the injection of insecure scripts (loaded over insecure HTTP channels), inline scripts, and the use of eval-like functions, used for turning strings into code. Their proposal of banning these insecure practices is now part of the browsers extensions APIs [2, 25, 100, 110].

Guha et al. [195] proposed IBEX, a platform for writing cross-browser extensions in high-level type safe languages such as .NET and a secure subset of JavaScript. Finding the permission system of Chrome extension API too coarse-grained, they propose to specify more fine-grained access control and data flow policies for extensions. Then they provide tools for verifying the compliance of the extension with the security policies.

Kapravelos et al. [213] introduced Hulk, for discovering malicious extensions with a dynamic analysis system in which they monitor the execution and network traffic of extensions. To trigger extensions behavior, they present honeypages and are able to generate on-the-fly elements that extensions require access to. Then, they use a fuzzer to trigger network events listeners in the extension, to which they present mock network objects. Applying Hulk to Chrome extensions allowed them to discover malicious extensions performing user credentials theft, social network abuse, etc.

Following the idea of honey pages, Weissbacher et al. [270] introduced Ex-Ray for discovering history-leaking Chrome extensions.

Starov and Nickiforakis [259] also performed a dynamic analysis of Chrome extensions and found many extensions leaking sensitive user information such as browsing history, search

queries, form data and extensions list to third parties.
Calzavara et al. [176] modeled the Chrome browser extension system, and formalized the privileges that an opponent can escalate thanks to the message passing API between web applications and extensions content scripts. They then proposed a prototype implementation of their system, named CHEN, which can be used by extension developers to evaluate the robustness of an extension against privilege escalation and help them refactor their extensions.

Communications with web applications

In Chapter 8, we present the first large-scale study on the security and privacy implications of the communications between browser extensions and web applications, allowing the latter to benefit from extensions privileged capabilities. We built a static analysis for analyzing extensions and identified a good number of them, demonstrating how these extensions can be exploited by web applications to benefit from extensions privileged capabilities and thereby access sensitive user information: bypass the SOP and read user data on any web application, access user cookies, browsing history, bookmarks, list of installed extensions, store and retrieve data from extension storage for tracking purposes, or even trigger the download of malicious files on the user's device.
Our work has some similarities with the work of Calzavara et al. [176], since we are also interested in the message passing interfaces. However, technically, we believe that our tool is more engineered than their implementation prototype and the goal of their study was not to systematically study the security and privacy implications of the message passing interfaces at large-scale. For instance, they did not model many of Chrome sensitive APIs, and the related threats, as we have done in this work. The way message handlers are extracted and analyzed is also very different. While they considered only the first function registered as listener, our tool is able to track sensitive APIs calls in the first handler and all of its dependencies (functions that it further invokes). Moreover, their tool does not consider the complexities of JavaScript function invocations, object properties accesses, which influences the precision of the tool to detect APIs calls and listeners registration. Content scripts in the Chrome extension API [25] at the time of their study were not privileged. In other words, they always needed to forward messages to the background pages in order to get access to the privileged APIs. Content scripts in WebExtensions however are privileged: they are not subject to the Same Origin Policy, and have access to the extension storage. They also did not consider direct communications between extensions background pages and webpages, while we found this practice widespread among extensions. The long-term communications (ports) between content scripts and background pages were also not considered. While they did not perform any large-scale analysis with their system, we analyzed Chrome, Firefox and Opera extensions and discovered many of them with different security and privacy threats.

CORS headers manipulations

In Chapter 9, we study the implications of CORS headers manipulations by browser extensions. We first developed `CORSER`, a cross-browser extension that tampers with CORS headers so as to authorize unauthorized cross-origin requests. If the development of such an extension requires little effort, it rather requires a good understanding of the CORS mechanism. Worryingly, we found that such an extension is considered benign from a browser vendors perspective. In fact, it successfully passed extensions review processes on

Chrome, Firefox and Opera where we published it. Furthermore, we performed an empirical study on extensions permissions and found that around 10% of Chrome, Firefox and Opera extensions have the capability to disable the SOP in browsers by tampering with CORS headers. We further statically analyzed extensions source codes and found that a few of them effectively tamper with CORS headers to allow cross-origin requests. But more surprisingly, we found that many extension developers misunderstand the CORS mechanism, as they manipulate HTTP headers in a way that fails legitimate CORS requests, thereby breaking web applications running in the user browser. Finally we discuss countermeasures and different proposals as to improve the security and privacy of users, and the security of web browsers and applications.

To the best of our knowledge, only two works have studied the manipulation of security headers by browser extensions. By analyzing the network traffic generated by Chrome extensions, Kapravelos et al. [213] flagged 24 of them as malicious because they were tampering with `Content-Security-Policy` and `X-Frame-Options` security headers. Hauskenetch et al. [200] found many extensions tampering with the `Content-Security-Policy` and proposed an endorsement mechanism, which can be implemented by browsers and web servers, to authorize or not CSP headers modifications. Tampering with these two headers leaves a web application unprotected against the attacks they mitigate, namely Cross-Site Scripting [41] and clickjacking [244] attacks.

Our work is the very first that addresses the ability of extensions to directly disable the Same Origin Policy in browsers, by appropriately tampering with CORS headers. Contrary to other security headers which when remove introduce security threats in specific web pages, and further require that the page has a vulnerability that an attacker can exploit, tampering with CORS headers however removes the Same Origin Policy protection in browsers, giving attackers unrestricted access right away to all user data in any web application. As we have also shown, harshly tampering with CORS headers can even break the normal functionality of web applications, preventing a user from using her favorite applications in the browser.

Chapter 8

Implications of the communications between browser extensions and web applications

Preamble

This chapter reports the results of an analysis of the communications between browser extensions and web applications. We found many extensions that can be exploited by web applications to access sensitive user information.
This chapter is under submission

1 Introduction

In this work, we focus on the WebExtensions API, the cross-browser extensions system compatible with major browsers including including Chrome, Firefox, Opera and Microsoft Edge [2,25,100,110]. Extensions can make HTTP requests to get data from any web application server, including those where users are logged into, such as their mailing, banking, social network applications, etc. As a comparison, web applications are bound by the Same Origin Policy (SOP) [125] and cannot access other web applications data, unless they both implement mechanisms such as Cross-Origin Resource Sharing (CORS) [40]. Web applications can store information in the user browser (cookies, HTML5 localStorage, cache, etc.). However, since such storage mechanisms can be abused for tracking purposes [225, 242], modern browsers provide users with the ability to prevent, control or remove information that web applications can store. Extensions on the contrary, have access to a persistent storage, in where they can store and retrieve data as long as they are installed in a browser. Even when users clear their browsing data, extensions storages are not affected. Other privileged APIs that browser extensions can use, are APIs to read and write user browsing history, bookmarks, cookies, manage the list of extensions the user has installed, or even trigger the download of arbitrary files and save them in the user's device.
For security reasons, extensions and web applications execute in different and isolated contexts. Extensions can inject content directly in the execution context of web applications, but the contrary is not possible. Nonetheless, there are many mechanisms that can be used by extensions and web applications to exchange data. First of all, extensions have access to web applications DOM and localStorage, which they can read and write while executing in their separate contexts. These modifications are visible to both sides (extension and web application), and can thus serve as means for sharing data. Moreover, extensions and web apps can set up communication channels to exchange data with one another using the `postMessage` API [39] for instance.

In this work, we focus on the communications channels that extensions can establish with web applications to exchange data. The messages the extensions expect from web applications, and more importantly how they handle them, is entirely up to the developers of the extensions. For instance, an extension can allow an application to send it the URL of a resource (data) hosted by another web application. It then makes a request to fetch the data (since it can do so with any web application as it is not subject to the SOP) and returns the response to the web application that previously sent the message. Another extension may allow a web application to send information that it will store in its persistent storage. Later on, the same application can send a message to the extension, which retrieves the previously stored data, and returns it back to the application. Yet again in another scenario, upon receiving a message from an application, an extension can retrieve the list of extensions the user has installed, or their browsing history, bookmarks, cookies and send them back to the application. Hence, these communications channels are a way for an extension to indirectly give a web application access to browser features and APIs that the web application is not directly allowed to access.

We built a static analyzer and applied it to the message passing interfaces exposed by Google Chrome, Firefox and Opera extensions to web applications. When the tool found that a privileged extension capability could potentially be exploited by web applications, the extension was flagged suspicious. By manually reviewing the code of suspicious extensions, we found that 197 of them (mostly on Chrome) can be exploited by web applications (attackers) to access elevated browser features and APIs and sensitive user information. Our results let us analyze the security and privacy implications of the communications between extensions and web applications. Extensions can be exploited by web applications to bypass the Same Origin Policy and access user data on any application including those applications where the user is logged into. Persisting data in the extension storage can be exploited by a web application to uniquely identify the user and track her even though she used privacy features provided by modern browsers such as blocking cookies, cleaning applications storages, etc. By reading the user's credentials (cookies), an attacker can perform session hijacking attacks [129], access user data and take arbitrary actions on her behalf. In addition, accessing the user's browsing history or bookmarks violates her privacy, and represents valuable information, which in the hands of an attacker, can be used to serve targeted advertisement, or even uniquely identify the user for tracking purposes. Discovering the list of extensions a user has installed reveals information about the user's interest and can serve to fingerprint her browser [198, 245, 249, 260]. Finally, being able to trigger downloads can be exploited by an attacker to add malicious software to the user's device. Inadvertently executing such software could let an attacker take control of the user's device and perform malicious actions (exfiltrating or damaging her data).

In summary, this work shows the security and privacy threats associated with the interactions between browser extensions and web applications and makes the following contributions:

— We built a static analysis tool and analyzed extensions message passing interfaces at large-scale: 66,401, 9,391 and 2,523 extensions on Chrome, Firefox and Opera respectively. About 4.97%, 5.14% and 8.48% of Chrome, Firefox and Opera extensions respectively were flagged as suspicious.

— We identified 197 extensions that pose various security and privacy threats to browsers, web applications, and users. They can be exploited by web applications to bypass the SOP, read user cookies, browsing history, bookmarks, list of installed extensions, store and retrieve data from the extension storage, or download malicious files and store them on the user device.

Our findings raise many questions about the security and privacy design of extensions. Despite the threats of the aforementioned interactions between extensions and web applications, we found no browser vendor that warned users about the possible threats posed by the extension they install. We argue that browser vendors need to review extensions more rigorously, in particular take into consideration the security and privacy threats that we have identified. The static analysis tool we have developed could be applied to any extension on major browsers in order to identify and fix the threats described in this work, before the extension is made public for users to install and use.

2 Context

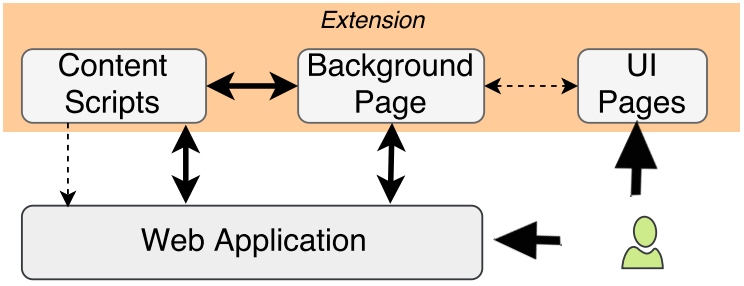


Figure 8.1 – Browser extensions architecture - Communications with web applications

Extensions can be divided in three main parts, as shown in Figure 8.1. The background page is the main part of the extension. It has full access to all the capabilities of the extension. Users interact with the extension through UI pages (i.e. UI elements, options and setting pages), in order to enable or disable it, or customize its behavior. UI pages also have access to the full capabilities of the extension. Content scripts are injected by extensions to run along web applications. Even though they are not granted access to all the extension capabilities, they can directly use the `host` and `storage` permissions to access user data on any web application or to store and retrieve data from the extension storage. Content scripts can also manipulate the DOM of webpages [48] and inject content in them. On Chrome and Opera, each extension is assigned a permanent unique identifier, which is the same for all users of the extension. Firefox however generates a random identifier for the extension, per user browser [52].

2.1 Interactions

Background and UI pages have direct access to each other's execution contexts [19], but content scripts execute in a separate context. Web applications run in yet other separate execution contexts. Nonetheless, content scripts have direct access to web applications localStorage, DOM, and execution context, where they can inject and execute arbitrary scripts.

Even though content scripts, background pages and web applications run in separate execution contexts, they can establish communication channels to exchange messages with one another [93, 107] as shown in Figure 8.1. We describe below the APIs for sending and receiving (listening for) messages between the content scripts, background pages and web applications.

Content scripts and background pages There are two types of communication channels: one-time and long-lived channels. One-time channels are opened to send a message

and are closed after the response is received. Long-lived channels, connections or ports, are maintained open to exchange multiple messages. A port can have a name in order to distinguish it from other long-lived channels.
For one-time messages, content scripts use the `runtime.sendMessage` API to send messages to background pages. Similarly, background pages employ the `tabs.sendMessage` API to send messages to content scripts. For receiving messages, both components can invoke the `runtime.onMessage.addListener` API.
Similarly, `runtime.onConnect.addListener` and `runtime.connect` are used to establish long-term communications between background pages and content scripts.
Web applications and content scripts Exchanges between web applications and content scripts are achieved with the Cross-Origin Communications API [39]: `postMessage` is used for sending messages, and `onmessage` or `addEventListener` to receive messages. Below is a listing which shows how messages are sent and received between web applications and content scripts.

```
  // Send and receive
postMessage("Hello Extension", "*");
addEventListener("message", function(event){
  Received_response = event.data;
});
  // Receive and Reply
addEventListener("message", function(event){
  Received_message = event.data;
  postMessage("Hello Web Application", "*")
});
```

In this example, the web application sends the message `Hello Extension` to the content script, which receives and writes it in the variable `Received_message`. Then it replies with `Hello Web application`, which the web application receives and saves in the variable `Received_response`.
Web applications and background pages On Chrome and Opera, web applications can also directly communicate with extensions background pages. To do so, extensions have to declare in their `manifest.json` file, using the `externally_connectable` key, the list of web applications, where communication with the background page is allowed. For security reasons, one cannot use wildcard (for instance *) to allow communications between the background pages and all web applications. Additionally, communications can only be initiated by web applications.
The `runtime.sendMessage` and `runtime.connect` APIs are exposed to web applications in Chrome and Opera, and can be used to send one-time messages or establish long-term connections with background pages. The APIs `runtime.onMessageExternal.addListener` and `runtime.onConnectExternal.addListener` are used in the background page, to receive and reply to messages sent by web applications. Below is an example of how to send a message from a web application to the background page of an extension which unique identifier is `ExtensionID`.

```
  // Send and Receive
chrome.runtime.sendMessage(ExtensionID, "Hello Extension",
    function(response){
  Received_response = response;
});
  // Recieve and Reply
chrome.runtime.onMessageExternal.addListener(function(message,
    sender, sendResponse){
```

```
  Received_message = message;
  sendResponse("Hello Web application");
})
```

The application sends `Hello Extension` to the background page which replies with `Hello Web application`.

2.2 Threat models

An attacker is a script that is present in a web application currently running in the user browser. The script either belongs to the web application or to a third party. The goal of the attacker is to interact with installed extensions, in order to access user sensitive information. He relies on extensions whose privileged capabilities can be exploited via an exchange of messages with scripts in the web application. We consider the following security and privacy threats posed by extensions.

1. **Execute code**: these are extensions that can be exploited by the attacker to execute arbitrary codes in the extension context. Executing code in the background page gives the attacker access to all the capabilities of the extension. In content scripts, the attacker can bypass SOP by making cross-origin AJAX requests, and use the extension permanent storage for tracking purposes.
2. **Bypass SOP**: in this case, an attacker can exploit the capability of the extension to make cross-origin requests without being restricted by the Same Origin Policy.
3. **Read cookies**: the attacker can read the user cookies and use them to mount session hijacking attacks, access user data and take actions on her behalf.
4. **Trigger downloads**: the attacker exploits extensions to trigger the download of arbitrary malicious files (software) and saves them on the user's device without requiring any action from the user. If the user inadvertently runs such software, the attacker takes control of her device and performs malicious actions.
5. **Read browsing history, bookmarks and list of installed extensions**: these information reveal the user interests and habits and can be used by the attacker for tracking purposes, or to serve targeted and personalized advertisement.
6. **Store data**: the attacker can store and retrieve information in the extension storage. This can be used for tracking purposes, even though users clear web applications storages.

For the sake of simplicity, we often refer to the attacker as the web application in which it runs. One can view at https://swexts.000webhostapp.com/extensions/ a set of videos demonstrating how we exploited these threats on some Chrome extensions.

3 Methodology

We built a static analyzer that detects suspicious communications enabled by extensions with web applications. To identify extensions that are potentially concerned with the security and privacy threats identified in the previous section, we focus on 78,315 extensions from Chrome, Firefox and Opera browsers. Then we manually reviewed the code of the extensions to precisely validate the results of the static analyzer, and more importantly to construct the signatures of the messages that have to be exchanged with extensions to successfully exploit their capabilities. Figure 8.2 shows the analysis process.

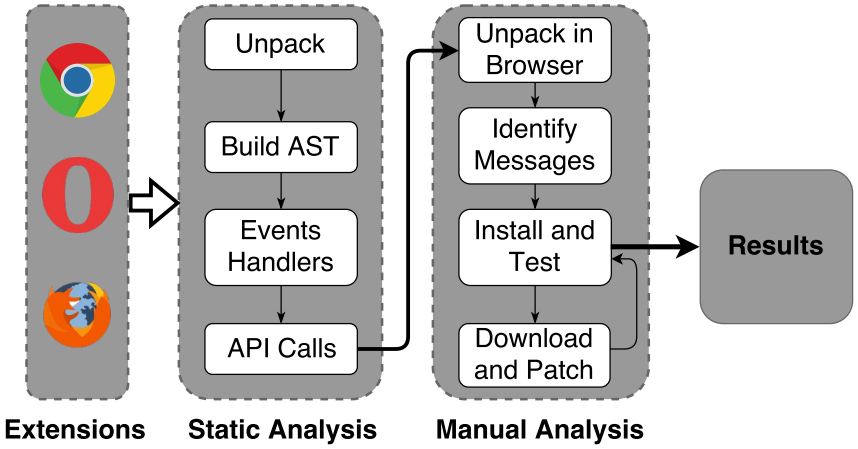


Figure 8.2 – Methodology - static and manual analysis

3.1 Static analysis

The goal of the static analyzer is to report only extensions that potentially pose a security and privacy threat, in order to reduce false positives as much as possible, and reduce the burden of the manual analysis. It has been fully written in JavaScript, using various Node.js packages. We used Esprima [204] and Recast [228], for parsing and manipulating JavaScript abstract syntax trees (AST), and Jsdom [69] for parsing HTML.

Unpack extensions and gather scripts We crawled extensions using SlimerJS Browser Automation tool [130]. In the extension `manifest.json` file, background pages are either declared by a set of scripts files, or an HTML file, which further includes the scripts of the background page. UI pages are built as HTML pages, and also indicated in `manifest.json` file. The Jsdom HTML parser served here to extract scripts embedded in background as well as UI pages. Static content scripts are directly declared in the `manifest.json` file. Background and UI pages can further dynamically inject content scripts in web applications, by calling the `tabs.executeScript` API. Those were also extracted by analyzing the AST of background and UI pages scripts, and analyzed as other content scripts.

Parse scripts and build AST Scripts were parsed with Esprima, resulting in an AST [4], which contains all JavaScript constructs used in content scripts, background and UI pages scripts. Almost everything in JavaScript is an object [50]. To ease manipulation of the AST, the following additional actions were taken to build three indexed tables of assignments to variables and object properties (`assignments`), function definitions/expressions and object methods (`functions`), and finally functions and object methods invocations (`calls`). Basically, those are key/value pairs, in which the keys in the tables corresponded to the names of variables, object properties and functions. Each entry was then associated with a list of all possible values it could resolve to. For assignments, the values were all expressions assigned to a variable or object. For function definitions and object methods, the values were the parameters and body of the function. Finally, for function calls, the values associated to their names in the indexed table were their invocation arguments. The static analyzer successfully handled functions defined using the `bind` method, and functions invoked using the `call` or `apply` methods.

Event handlers of page messages APIs For each message listener (See Section 2) in content scripts, background and UI pages, we first looked up the indexed table of function invocations (`calls`) to search whether the extension registered listeners for messages from the web applications (a call to `addEventListener` API for instance in con-

tent scripts). In browser contexts, all JavaScript objects are properties of a global object named `window`. Different aliases, `this, self, global`, are sometimes used to refer to the `window` object [83, 154]. JavaScript object properties can be accessed using the dot and the array or bracket notations [81]. For the sake of simplicity, we considered the dot notation and the bracket notation when the property name was a literal (a string). Considering the global object names (`window, top, self, this`), and JavaScript dot and bracket property accesses, we generated the different ways an API can be invoked. For instance, `addEventListener` can be called in 9 different ways `addEventListener, window.addEventListener, window["addEventListener"]` and others. In general, we consider that an object could be accessed in 9 different ways, its properties in 18 different ways, the properties of its properties in 36 ways and so forth. When we found an invocation to communications APIs in content scripts, background and UI pages, we extracted their arguments and resolved them as follows.

For `addEventListener`, the first argument should be the literal `message`, and the second argument a function. Otherwise, we use the indexed table of `assignments` and `functions` to resolve them to the literal `message` and a function respectively. Resolving an argument simply consist in checking whether the indexed table has an entry which key matches the argument name, and further checking whether any of its associated values resolve to the type and value we expect the argument to have. For `addEventListener`, we expect the first argument to be a `Literal` and have the value `message`. Its second argument is expected to be a function. We follow the same process to extract all message handlers (listeners) in content scripts, background and UI pages.

Sensitive APIs Calls The handlers (functions) of web applications messages in extensions are parsed to extract all their constructs. If the handlers further call other functions, those functions are looked up using the indexed table, and their bodies parsed to also extract their constructs. Finally, the constructs are analyzed to decide whether the extension potentially poses any of the security and privacy threats considered in this work.

- An extension is flagged as potentially executing arbitrary code sent from web applications if it invokes functions like `eval` (in any part of the extension) or `tabs.executeScript` (in background and UI pages).
- An extension is flagged as potentially allowing web applications to bypass SOP, if its constructs include APIs that can be used to make AJAX calls. This includes the creation of new `XMLHttpRequest` objects, calls to `fetch` API, or any AJAX specific API provided by popular third party libraries such jQuery and AngularJS (`$.get, $.ajax, $.post, $http.get, $http.post`).
- If the constructs include invocations to `storage` APIs such as `storage.local.set`, `storage.local.get`, `storage.sync.set`, `storage.sync.get`, then the extension is flagged as potentially storing/retrieving data for web applications.
- An extension is considered as potentially leaking user cookies, history, bookmarks, and list of extensions to web applications if either of the following invocations were found in their message handlers constructs: `cookies.getAll, history.search, history.getVisits, bookmarks.getTree, management.getAll`, and related APIs.
- Finally, an extension is considered as probably allowing web applications to download and save files in the user computer (device) if their messages event handlers constructs include invocation to `downloads.download`.

It is worth mentioning the case of content scripts forwarding messages to background pages. When this is the case, the constructs of content scripts messages handlers in the background pages are also analyzed, looking for calls to any API which potentially poses

security and privacy threats. In fact, content scripts only have access to the `host` and `storage` capabilities. When they need access to more capabilities, they can send messages to the background pages which may then give them access to the related capability. Content scripts can forward messages they receive from web applications, to the background page. The latter handles the message and responds to the content scripts which in turn respond to the application. This is particularly true in Firefox which does not allow direct communications between web applications and background pages. Nonetheless, we have observed many content scripts forwarding messages to background pages, even to access APIs they can directly use from their own context.

3.2 Manual Analysis

The goal of the manual analysis was to confirm the suspicion of the static analyzer and build the precise signatures of messages that had to be sent by web applications to exploit extensions capabilities. Extensions reported for manual analysis were unpacked in the browser using the CRX Extension Viewer [277]. We inspected their message handlers. If the suspicion was confirmed, we built the signature of the messages that the extension accepted to handle. We also identified the web applications from which the messages have to be sent, and the targets of those messages in the case of SOP bypass.
Then we installed the extension and interacted with it, navigating to the appropriate web applications, and interacting with the extension by sending messages from the browser console [17] and validating that the extensions successfully replied with the requested information. For some extensions, we patched their code with hooks in the message handlers, installed them again (in developer mode) and interacted with them to validate the results.

3.3 Limitations

Our static analysis tool suffers from many limitations. The first one is the fact that we did not consider scopes [83], which lead to unnecessary functions being analyzed. However, this is not a problem ultimately because all the results were further manually reviewed to remove false positives. It also suffers from some false negatives, mainly because of the flexibility of JavaScript that make it challenging to exhaustively address all the ways message listeners can be invoked in extensions. Finally, for a few extensions, despite all our efforts at the static and manual analysis levels, we could not draw any conclusion about the potential threats they may pose.
Note also that we considered only scripts that are part of the extension packages. For instance, background and UI pages may reference external scripts. Those scripts were not considered in our analysis. Nonetheless, we think that extensions bundles are more likely to contain most of the APIs that we consider in this work, as extensions developers are recommended to avoid referencing remote scripts in extensions codes.

4 Empirical Study

We downloaded Chrome [24], Opera [108], and Firefox [58] extensions by the end of November 2017. The extensions were statically analyzed in the beginning of February 2018 — on a cluster of 200 nodes mainly because of storage limitations on our own devices. This was preceded by a long period of tests during which we improved the static analyzer, and fixed the list of security and privacy threats. In the middle of May 2018, we did another crawl and analysis. The results presented here are for this second dataset.

In this section, we first give an overview of the results, then we discuss in more details each threat and the report extensions where it was found.

4.1 Overview

Table 8.1 presents the number of extensions we collected and analyzed. Chrome provides the largest share of extensions, followed by Firefox and Opera. Recall that for Firefox, we are considering only extensions built using the new WebExtensions API [156], and not those using the XPCOM/XUL API [100].
The static analysis tool reported 3,996 suspicious extensions that we manually vetted. The results of the manual analysis are also shown in Table 8.1. As with the share of extensions, Chrome had the largest share of extensions with threats. In a total of 197 extensions, only 16 were found on Firefox, 10 on Opera, and the 171 others are Chrome extensions. Note that some single extensions pose more than one threat at a time. All the 197 extensions reported here effectively posed at least one or more of the security and privacy threats described in Section 2. During the manual analysis, we also identified the messages to be sent in order to exploit their capabilities. The full list of the extensions and the threats that they pose are given in Table A.7 in the Appendix, to ease readability.

Table 8.1 – Data overview

	Chrome	Firefox	Opera	Total
Extensions analyzed	66,401	9,391	2,523	78,315
Suspicious extensions	3,303	483	210	3,996
Execute Code	15	2	2	19
Bypass SOP	48	9	6	63
Read Cookies	8	-	-	8
Read History	40	-	-	40
Read Bookmarks	37	1	-	38
Get Extensions Installed	33	-	-	33
Store/Retrieve Data	85	2	3	90
Trigger Downloads	29	5	2	36
Total of unique extensions	**171**	**16**	**10**	**197**

Extensions installs and categories Figure 8.3 presents the distribution of users impacted, or the number of installs per extension at the time of writing this thesis. Around 55% of the extensions have less than 1000 users, while the remainder 45% have thousands of installs, showing that those threats are present in rather popular extensions, hence affecting many users. About 27% of extensions have less than 100 users and another 27% have between 100 and 1000 users. We see this as an opportunity for a tool such as ours to help improve extensions security, as it can serve to detect potentially malicious extensions while they are not yet very popular among users, thereby limiting their impact on users.
Table 8.2 further presents the category of these extensions. Note that the categorization of extensions is not done the same way by Chrome, Firefox and Opera browsers. Some categories exist only on specific browser, and not on others. Moreover, we found similar (or the exact same) extensions being differently classified depending on the browser. We tried to merge the different categories whenever possible.
As one can observe, `Productivity` is the most popular category among the reported extensions. It is also the most popular category among all Chrome and Opera extensions we have downloaded, and also the most popular category in various datasets in recent

studies [245, 259, 260]. This category does not exist on Firefox.
We were surprised by the results that only 15 extensions (7.61%) are classified as `Developer Tools`. Considering the severity of the threats, we were expecting that most of them would be extensions provided for developers to perform some controlled experiments. Since our results represent only a lower bound of the number of extensions potentially posing these risks, it would not be surprising that even more extensions also exhibit similar threats.

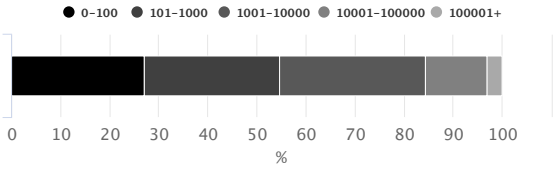


Figure 8.3 – Distribution of the number of users per extension

Table 8.2 – Category of extensions

Category	# Extensions
Productivity	81
Social & Communication	48
Fun	19
Accessibility	17
Developer Tools	15
Search Tools	6
Shopping	4
Blogging	2
Privacy & Security	2
Other	2
Appearance	1
Total	**197**

Extensions privilege only some web applications About 55 extensions (45, 7 and 3 on Chrome, Firefox and Opera respectively) communicate with any web applications to give them access to extensions privileged APIs. Interestingly, on Chrome, 7 of them allow to execute arbitrary code in the extension context, 15 of them are concerned with SOP bypass, 26 for storing data, 2 can be exploited by any web application to read all user cookies and 5 to read the cookies of the current web application.
The vast remainder of extensions (72.08%) can be exploited only by specific web apps to benefit from their privileged capabilities. For instance, reading user browsing history, bookmarks, and list of installed extensions, is enabled by extensions only to specific applications such as `fliptab.io`, `atavi.com`, `mail.google.com`. In particular, downloads are allowed by many extensions (on Chrome and Opera) mostly from `vk.com`.
The fact that most extensions allow communications with only some specific apps can also be explained by the fact that most of those we found allow interactions between web apps and the background pages directly. Let us recall that it is only possible to allow communications between background pages and specific web apps (and not all web apps).
Extensions allow to connect to arbitrary web applications If many extensions tend to privilege specific web applications as shown previously, the exact opposite is observed regarding the hosts extensions allow web applications to connect to, in order to access user data. For example, 37 out of the 48 extensions that can be used to bypass SOP on

Chrome, give access to the user data on any other application. On Firefox, it is 6 out of the 9 extensions which allow access to any web application data.
These two observations (extensions mostly give access to their privileged APIs only to some web applications, and allow them to access any other web application data in the case of SOP bypass) suggest that the access they give to their capabilities is rather deliberate. Moreover, for the majority of extensions, the messages to send to exploit the different APIs in extensions are so trivial that they could have only been deliberate (See Section 6).
Most privileged web applications As already mentioned, most extensions allow specific apps to benefit from their privileged APIs. This is the case for instance of `fliptab.io` where scripts can communicate with 31 very similar HD wallpaper extensions on Chrome, that has hundreds to thousands of users. The domain `vk.com` can interact with 19 extensions (17 on Chrome and 2 on Opera), mostly to download files on the user device. The domain `atavi.com` can get access to user's history, most visited websites (topsites) and bookmarks thanks to 6 extensions.
Extensions which pose more than one threat All the extensions reported here pose at least 1 of the security and privacy threats considered in this work. Nonetheless, some extensions pose several threats.
The `eRail.in` [51] extension on Chrome gives access to all user cookies and allows full SOP bypass from any web application. Moreover, it has more than 400k users. Interestingly, a version of the extension exists on Firefox, but it leaks cookies and data of a limited set of web applications (all related to the extension owner's domain) to the the extension's provider own domains. Five extensions provided by Fabasoft (See Table A.4 in the Appendix) leak the current tab cookies. As such, they allow attackers to even access HTTPOnly cookies, and use them to mount session hijacking attacks.
`Ringostat dialer` [123] is the only extension that executes arbitrary code sent from `app.ringostat.com` directly in its background page. All other extensions execute the arbitrary attacker code in the context of the content scripts. Recall that the background page has access to all the capabilities an extension declares. Interestingly, it has the `host`, `storage`, `cookies`, and `tabs` permissions, meaning that any script present on `app.ringostat.com` can access user data on any other domain, access the extension storage, cookies, open new tabs, inject code directly in any tab, etc.
`StartHQ` [132] also allows to bypass SOP from `starthq.com`, and leaks user browsing history. Similarly, `SalesforceIQ CRM` [124] allows to bypass SOP and leaks installed extensions to `mail.google.com` and `salesforceiq.com`.
Finally, user browsing history, bookmarks and installed extensions can be read by an attacker in `atavi.com` and `*.fliptab.io` thanks to 6 and 31 extensions respectively (See the full list in the Appendix). The latter also let `fliptab.io` stores and retrieves data in the extension storage.
Cross-browser extensions It is worth mentioning that most of the extensions we found on Opera and Firefox were also present on Chrome. While the compatibility of extensions APIs on major browsers [2, 25, 100, 110] let developers reach more users, attackers also widen their attack surface because they can impact more users thanks a single cross-browser extension. For instance, we have noticed that `megatest2016`, an extension provider, had 2 extensions on Chrome, and a very similar one on Opera. At the time of writing this thesis, Chrome removed the 2 extensions (they were allowing `ok.ru` and other applications to bypass SOP, but we do not know if their removal were due to the SOP bypass) while on Opera, it is still available as `MegaTest` [92]. The 2 `Photo Zoom for Facebook` and `Facebook Photo Zoom` Firefox add-ons have similar versions on Chrome, but these do not allow SOP bypass. Similarly, the `ModernDeck` extension is present both on Opera [98]

and Chrome [97]). On Opera, it allows to store/retrieve data, while on Chrome it does not. This represent yet another problem of cross-browser extensions. While users of the same extension suffer from security and privacy threats on one browser, on the other browser where the extension is removed or fixed, users do not. Browser vendors, and more importantly users would gain from security and privacy perspectives, if browser vendors share their reviews of extensions with one another, in order to help take similar actions like removing extensions, or updating them to remove threats they pose.

4.2 Execute code

Extensions execute in browsers with elevated privileges. From an attacker's perspective, being able to execute arbitrary code in an extension context also gives the attacker access to the extension capabilities. We found 15 extensions on Chrome, 2 on Firefox and 2 on Opera that can be exploited by web apps to execute code in their privileged context. Only one extension on Chrome `Ringostat dialer` [123] executes in its background page, code that it receives from `app.ringostat.com`. Then it gives access to user data on any application, user cookies, allows code injection in in any tab the user opens, the use of the extension storage, etc. All other extensions execute the attacker's code in the contexts of the content scripts. Even though content scripts have limited access to extensions capabilities, they are not subject to SOP, can store/retrieve data, and more importantly, they have access to the full DOM on the web applications pages in which they are injected.
The extension `iwassa`, present on Opera [79] and Chrome [78] allows any app to open any URL in a new tab, and execute any code (content script) in it. If the code in the context of the content script can already access any application data, one can further inject specific content in the DOM of the new tabs opened, to exfiltrate for instance any token/secret present in the application DOM. In fact, in addition to cookies, many sensitive applications use tokens to further perform additional checks about the origins of requests before letting users perform sensitive actions on their data.
Another interesting example is that of the `LinkClicker` extension also present on Opera [88] and Chrome [87]. It allows any application to send a code which will be further injected in any new tab the user opens during the current browsing session. One can use it to track the user while she is browsing, gather any credentials that she is providing to log into any application, and exfiltrate those to the attacker.
In many of these cases, the problem is due to the fact that the extension does not correctly sanitize the codes received from web applications, allowing attackers to execute arbitrary codes. A good example is that of the `GureTV: To watch television` extension on Firefox [67]. It did well to sanitize content sent from web applications, but not content sent from iframes embedded in the applications. Hence, one can create an iframe, and send an arbitrary code which will be executed in the context of the content scripts.
Many of the other extensions work similarly, and allow (at least) to access arbitrary user data on any application, and/or store and retrieve data (when they have the appropriate permissions).

4.3 Bypass SOP

Extensions are not subject to the SOP, and therefore have access to user data on any web application for which they have declared the `host` permission. Through message exchanges with extensions, 48, 9 and 6 of extensions on Chrome, Firefox and Opera respectively, allow web applications to bypass SOP by accessing user data on any other web application. As for other threats, the trend is rather to allow only some web applications to bypass SOP, even

though 15 of such Chrome extensions allow any application to access any other application data. Hence, the majority of arbitrary SOP bypass can be exploited by specific web applications, including: ok.ru, mail.google.com, logincat.com, etc. Interestingly, when SOP bypass is possible, in most of the cases the data of all domains can be accessed. On Chrome for instance, it is 37 out of the 48 extensions that allow access to any application data. Even when the SOP bypass is partial, it is enabled to rather sensitive domains. For instance, 5 extensions out of 11 allow SOP bypass to users' Google accounts: salesmate.io, appspot.com and aliexpress.com can access users Gmail account. One extension [89] allows access to the linkedin.com data of more than 400k users from Gmail, and blog.renren.com can access github.com [120].

4.4 Cookies

We found 8 Chrome extensions that can be exploited by web applications to read user cookies: 2 of them allow any web application to read all user cookies [51, 133], 1 only allow app.ringostat.com [123] to read all user cookies, and the other 5 of them allow an attacker script to read the cookies of the tab in which it executes. The number of users affected is very important (more than 415k for eRail.in [51], 9.6k for Telerik Test Studio Chrome Playback 2014.1 [133] and 78 for Ringostat dialer [123]. Cookies can be used to hijack users browsing sessions, access their data and take actions on their behalf. It is worth mentioning that the three extensions that can be exploited to read all user cookies, have probably been poorly programmed. It is more likely that the ability to read cookies was meant to be used from specific web applications, but unfortunately the extensions were poorly programmed, allowing other web applications to also get access to user cookies. In particular, the Ringostat dialer [123] extension did not expose any means to get user cookies. But it allows to execute any code sent from app.ringostat.com in the extension background page context (using `eval` function), giving the application access to all the capabilities of the extension. Among those, the cookies, storage and arbitrary host permissions, and the ability to open tabs, inject and execute arbitrary code in them, etc.
We found that the web application https://erail.in/ is effectively reading all user cookies when the eRail.in [51] Chrome extension is installed. This means that the extension is intentionally given access to user cookies to https://erail.in. However, it is not clear whether the cookies of interest were only those of https://erail.in or any cookie or if only cookies of https://erail.in/ were meant to be leaked. Unfortunately, any web app can access all user cookies stored by any web application, and use them to hijack user sessions. Interestingly, the extension has a version on Firefox, where the cookies which are leaked are only those of domains related to erail.in and are leaked only to erail.in and eair.in.
The case of the extension Telerik Test Studio Chrome Playback 2014.1 [133] is particularly interesting, as one has to setup complex interactions, involving the extension content scripts and background page, as well as the application and its server. In particular, the interactions are triggered from the web application, but the cookies are sent to the server of the application instead of being returned directly to the application. Following the same mechanism, one can clear cookies, delete user browsing history, etc. A similar extension is also available on Firefox progress-test-studio-extension. Unfortunately, we could not analyze it as it was not downloading.
Finally, 5 Fabasoft extensions (See more details in Table A.4 in the Appendix) allow the attacker to read the current tab cookies of any web application. Even though a web application protects its cookies with the HTTPOnly flag [74], an attacker script running

in the web application bypasses this protection by obtaining the cookies via the extension. It can further use them to mount session hijacking attacks against the user.

4.5 Downloads

Exploiting extensions to trigger the download of arbitrary files is enabled mainly from specific applications including vk.com (See Table A.2 in Appendix) and ok.ru. Only 2 extensions on Chrome and 3 on Firefox allow downloads from arbitrary web apps. The main purpose of the related extensions were to allow the download of music and videos. Sometimes, they would even suffix the downloaded file name by .mp3 or .mp4. Nonetheless, we have been able to exploit these extensions in order to trigger the download of arbitrary files and save them in the user's device. An attacker can also do so to download malicious software, which when inadvertently executed by the user, may allow the attacker to take control of their computer and perform malicious actions.
It is worth mentioning that none of these extensions required user action to trigger the downloads. One of them, multiDownloader [101] even overwrites a file if it is already present on the user's device.
It is also worth mentioning the case of the Chrome repl.it download extension [121]. It is a helper extension for the https://repl.it application used for creating and running programs in different languages online. The extension allows to save the code being created. Even though the extension prompts the user to confirm the file name (default is program.), the content of the file can be fully arbitrary. As such, an attacker can trick the user in saving the code being edited, while a completely different content is saved.

4.6 History, bookmarks, and list of installed extensions

Two providers distinguish themselves with regards to extensions that can be exploited to get access to user browsing history, bookmarks and list of extensions. On Chrome, fliptab.io [68] provides 31 very similar HD wallpapers extensions (See the full list in Appendix), and allows fliptab.io to get all browsing history, bookmarks and the list of user installed extensions. Each of these extensions has between a hundred and 25k users. Furthermore, six extensions provided by atavi.com also provide the same privileges to pages at atavi.com and atavi.test. One of them, Atavi - bookmark manager [13] has more than 96k users.
Additionally, Browser History [18] leaks user browsing history to www.americaninternetmatrix.com/history. Finally, StartHQ [132] leaks browsing history to https://starthq.com. Other extensions that give access to the list of extensions include Boomerang for Gmail [15] (with more than 1.5 million users) to mail.google.com and SalesforceIQ CRM [124], to mail.google.com and salesforceiq.com.

4.7 Store/retrieve data

About 85 extensions can be exploited by various web applications to store and retrieve data. On Chrome, 26 of these extensions give any application access to their storage. Others give specific apps access to their storage. For instance, fliptab.io can store data in the user's browser thanks to its 31 extensions. The domain netflix.com is also able to store data thanks to 3 extensions, and mail.google.com to do so thanks to 2 extensions. The extensions ISOGG Y-Tree AddOn [77] and PhyloTreeMT AddOn [114] are from the same provider, even though the web applications they allow to persist data are respectively isogg.org and phylotree.org.

Recall that extensions storage is persistent and not affected by the clearing of browsing data (web application cookies, storages, ...). As such, they represent a resilient storage which can be used to bypass user privacy preferences and uniquely identify them even though they have cleared their cookies. Interestingly, some extensions propose to sync data they store on all the devices the user is logged into. For instance, if a user logs into multiple devices with the same extension installed, then syncing storages lets an application tracks her accross all her devices.

4.8 Other threats

For SOP bypass, we have reported here the cases where web applications can access arbitrary data on other web applications. Nonetheless, we found many extensions allowing to access some predefined data of other web applications. This also represents a SOP bypass (since web applications cannot access such data with their normal privileges). Finally, we found some Opera and Chrome extensions (like the 31 HD wallpaper extensions by `fliptab.io`), and some not reported here) which allow web applications to clear user browsing data including cookies (or even set/get cookies of some specific domains), history, bookmarks, cache, stored passwords, or enable/disable/uninstall extensions. We do not include such cases in this thesis.

5 Tool for analyzing message passing APIs

We provide online at `https://swexts.000webhostapp.com/extsanalyzer/`, a tool for analyzing the message passing APIs of extensions. The only difference with the version used in this work is that it does not handle dynamically injected content scripts. This was done for simplicity reasons. That notwithstanding, in order to analyze dynamic content scripts, one can simply declare them in the extension manifest as static content scripts.
Listing 8.1 shows the result produced by the tool when applied to the `eRail.in` Chrome extension [51].

```
{
    "com_via_cs": {
        "to_back": {
            "back": {
                "ajax": {
                    "$.get": "",
                    "$.post": "",
                    "$.ajax": "",
                    "XMLHttpRequest": ""
                },
                "cookies": {
                    "chrome.cookies.getAll": "",
                    "chrome.cookies.remove": "",
                    "cookies": ""
                }
            }
        }
    }
}
```

Listing 8.1 – Result of analyzing the `erail.in` extension

— `com_via_cs` implies that webpages can communicate with the extension via the content scripts, by using the `postMessage` API. This extension has only 1 content script. When there are multiple content scripts, the tool analyzes each of them independently and produces results corresponding to each of them.

— `to_back` indicates that the messages sent by webpages to the content script are forwarded to the extension background page.

— The tool found that two sensitive APIs are reached in the background page: AJAX requests with calls to the jQuery AJAX APIs (`$.get, $.post, $.ajax`) and access to cookies with invocation to the `chrome.cookies.getAll` and `chrome.cookies.remove` APIs.

The main goal of the tool is to raise awareness about the fact that an attacker may potentially get access to the extension's privileged APIs. One can then further review the code to validate or refute the results of the tool. For instance, after manually vetting the code of the `eRail.in` extension, we effectively confirm that any webpage can access all user cookies and make AJAX request to any domain. See Section 6 for more details about examples of messages to be sent to extensions to benefit from their privileged capabilities. There is room for further improving the tool. Lessons can be learnt from the state-of-the-art on JavaScript static analysis tools in order to improve the extraction of messages passing listeners and tracking the escalation of extensions sensitive APIs. The set of threats considered in this work can also be extended further with state-of-the-art extensions threats in the literature. Finally, our ultimate goal is to make the tool usable by everybody. By providing the name of a Chrome, Firefox or Opera extension, the tool would automatically download and analyze it for different threats and output a score as well as a non-technical explanation about the potential threats that an extension may pose. For extensions that have been manually vetted (as the ones found in our empirical study), the tool can therefore provide a precise report about the threats they pose and warn the user about whether the extension is malicious or not and thus if it is safe to install it.

6 Case study

In this section, we show how an attacker can exploit the capabilities of an extension by sending the appropriate message. In order to gain access to privileged browser features via an extension, an attacker first needs to ensure that the extension is installed and enabled. Many recent studies discussed extensions discovery, using for instance their unique identifiers and web accessible resources [245, 249] or DOM specific changes they introduce in web pages [260]. This is not really needed here. Knowing the structure of messages extensions respond to, is sufficient. If the extension is present, it will surely reply. To benefit from extensions capabilities, it is sufficient that the attacker is present in a web application with which the extension can interact. We recorded videos demonstrating some extensions and the threats discussed in this study. They are accessible at `https://swexts.000webhostapp.com/extensions/`.

6.1 Example of messages to send to extensions

We refer to Section 2 which presents the message passing APIs between webpages and the different components of an extension. We illustrate at least each threat by an extension.

Execute code in content scripts context Listing 8.2 present the structure of messages that can be sent from any webpage to the `jianlibao` [84] Chrome extension to execute

arbitrary code in the context of its content scripts. Replace `CODE` with real JavaScript code, then serialize the message using `JSON.stringify` before sending it. The extension has the `storage` and `host` permissions meaning that any page can bypass SOP and get access to user data on any domain, store data in the extension storage and later retrieve it for tracking purposes. Moreover, the code is injected in the active tab the user is interacting with. As the user may switch tabs at any time, one can send the code regularly (say every second) in order to ensure that it is injected in all the web applications the user is interacting with. Since content scripts have access to the DOM of webpages, the injected code also has full access to the active tab DOM, giving it the ability to undertake any action: recording user name and password, credit card numbers, emails, etc.

```
{
  type: "getResumeInfo",
  downloadObj: {
        resumeWhereabouts: 5
    },
    context: {
        contentScript: CODE,
        jsMethod: "console.log"
    }
}
```

Listing 8.2 – Executing arbitrary code in the context of the content scripts of the current tab the user navigates to, thanks to the `jianlibao` Chrome extension.

Extensions such as `iwassa` [78,79] or `LinkClicker` [87,88], present on Chrome and Opera, even allow to send a URL and a code. They will open the URL in a new tab, and execute the code in the context of the content scripts injected by the extension in the new tab. Listing 8.3 presents the case of the `iwassa` extension. Replace `URL` with the URL of the page to open in a new tab, and `CODE` with the real code to be executed in the context of the new tab content scripts.

```
{
  from: "logininfo",
  val: [URL, CODE, "LoginAPI"]
}
```

Listing 8.3 – Executing code in the context of a choosen tab thanks to the `iwassa` extension present on Chrome and Opera. `URL` is the URL of the page to open in a new tab, and `CODE` the code to be executed.

The extension also has the `host` permission, allowing to make AJAX requests to any domain.

Execute code in background page context Background pages are the most privileged contexts, as they have access to all the capabilities of an extension. Listing 8.4 shows the message to send to the `Ringostat dialer` [123] Chrome extension to execute arbitrary code in the context of its background page. Interestingly, this extension has the `host`, `storage`, `cookies` and `tabs` permission, giving an attacker the ability to bypass SOP, store data in the extension storage, manage user cookies and tabs (open new tabs, close some, etc.). Messages are to be sent from webpages which URLs match `*://app.ringostat.com/*`.

```
{
  message: "execCommand",
  data : {
```

```
    command: "eval",
    params: CODE
  }
}
```

Listing 8.4 – Message to send to `Ringostat dialer` background page to execute arbitrary code. Replace `CODE` with the real code to be executed.

Bypass SOP Here we take the example of the `Buxenger` extension, available both on Chrome and Firefox. Listing 8.5 shows the structure of messages to be sent to the extension in order to make AJAX requests to any domain (SOP bypass). The case shown here, is for making HTTP `GET` requests. But the extension also allows to make AJAX requests using HTTP `POST, DELETE, PATCH` methods.

```
{
  message: "ajax-get",
  url: URL,
  callbackId: ID
}
```

Listing 8.5 – Make arbitrary AJAX requests thanks to the `Buxenger` extension present on Chrome and Firefox. Replace `URL` with the URL of the data to access, and `ID` with any value.

Retrieve cookies Listing 8.6 shows the case of the `eRail.in` Chrome extension which allows any webpage to retrieve the list of user cookies.

```
{ Action: "GETCOOKIE" }
```

Listing 8.6 – Message to send to `erail.in` extension in order retrieve all user cookies

This includes any cookies, such as the user authentication cookies set after she has logged into web applications. One can further use the cookies to mount session hijacking attacks. The extension also allows to make arbitrary AJAX requests, by sending messages as shown in Listing 8.7

```
{
  Action: "GET_BLOB",
  URL: URL
}
```

Listing 8.7 – Making AJAX requests thanks to the `eRail.in` Chrome extension

Downloads files Listing 8.8 shows the signature of messages to send from any webpage, to the `HTTP Commander` [72] Chrome extension in order to trigger the download of any file. Replace `FILE_URL` with the URL of the file to download, and `FILE_NAME` with the name under which the file will be saved on the user device. Multiple files can be sent in the message. They will all be downloaded one after the other.

```
{
  type: "HTCOMNET_DOWNLOAD",
  files: [{
    url: FILE_URL,
    path: FILE_NAME
  }]
```

```
}
```

Listing 8.8 – Download files on the user device, thanks to the HTTP Commander extension.

Store data in extension storage Listing 8.9 shows messages to send in order to store and retrieve data in the VisualSP Training for Office 365 [151] Chrome extension storage. Replace DATA_TO_STORE with the data to be stored in the extension storage. Later on, send the second message to retrieve data. The data will be sent to iframes in the page. To collect the data previously stored in the extension storage, before sending the message, one can simply add an iframe to the webpage, then send the message, collect the previously stored data from the iframe, and send it back to the parent page.

```
// Store data
{
  owner: "VisualSP",
  command: "SetUserId",
  data: DATA_TO_STORE
}
// Retrieve data.
{
  owner: "VisualSP",
  command: "GetUserId"
}
```

Listing 8.9 – Store and retrieve data in VisualSP Training for Office 365 Chrome extension storage

History, bookmarks, extensions list We show here the case of the Space Galaxy HD Wallpapers [131]. It is one of the 31 HD Wallpapers from `fliptab.io` (See Table A.1 in the appendix) that lets pages matching `*.fliptab.io`, to manage user history, bookmarks, extensions list and storage. Listing 8.10 shows the different messages that has to be sent to get the related information.

```
// Message for retrieving user browsing history
{
  type: "history",
  act: "get_all"
}

// Message for retrieving bookmarks
{
  type: "bookmarks",
  act: "get_all"
}

// Message for retrieving the list of extensions
{
  type: "extensions",
  act: "get_all"
}
```

Listing 8.10 – The Space Galaxy HD Wallpapers Chrome extension allows to get user browsing history, bookmarks and extension list

6.2 Forcing the attack

In order for an attacker to gain access to an extension's APIs, he must have a script loaded in a web application that is allowed to interact with the extension. Moreover, in most cases, the application has to be running in the user browser in order for communications to be possible. Figure 8.4 shows a simple scenario in which `A.com` is an application currently running in a user browser. This application provides content `A.com/content` (a script) for another application `B.com` which can communicate with an extension to get access to some privileged APIs. However, `B.com` is not currently running in the user browser. `A.com` can force the attack to happen, by opening `B.com` (upon a user interaction with the `A.com`). Once `B.com` runs, the script that it embeds from `A.com` gets executed and can communicate with the extension to get access to its privileged APIs — for instance to access user data on any other application — and exfiltrate this to `A.com`. With the prevalence of some third party scripts providers among web applications [229], this scenario can be easily implemented by attackers to gain from extensions capabilities.

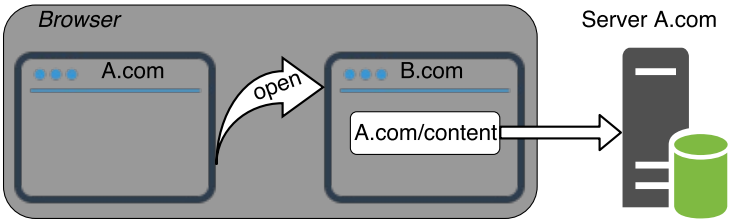


Figure 8.4 – `A.com` forces an attack by opening `B.com` thereby allowing `A.com/content` to load, execute and interact with extensions in order to exfiltrate user data to `A.com`.

Combining multiple extensions Another scenario where access to any extension capabilities can be indirectly gained is when some extensions make it possible to open new tabs and inject and execute arbitrary codes in them. We have recorded a video showing the use of the `LinkClicker` extension [87] which allows to open a new tab and execute code in it, and the `Space Galaxy HD Wallpapers` extension [131] which allows only `fliptab.io` to get/delete user browsing history, bookmarks and extensions list. From any application (the `localhost` in our example), we opened `www.flipatab.io`, and injected a code in its context. The code retrieved the list of extensions, bookmarks and user history. This information could be further sent to a server chosen by the attacker. One can even use also the `LinkClicker` extension to send the retrieved information back to the attacker by opening a new tab of the attacker application (`localhost` in our case). The video is also accessible at `https://swexts.000webhostapp.com/extensions/`

7 Discussion

Here we discuss countermeasures to mitigate these security and privacy threats introduced by browser extensions. We are not disclosing this list of extensions until vendors take a definitive decision regarding our findings. We did not either publicly share the link to the videos demonstrating how we exploited extensions. Vendors can deactivate vulnerable extensions until they are fixed and remove them if their developers are not willing to update their codes.

7.1 Browser vendors

We are planning to disclose to vendors the extensions posing any of the security and privacy threat we have reported in this paper. We also disclose and demonstrate our analysis, methodology and tool. There is definitely room for further improving the tool. That notwithstanding, as we have shown in this paper, the tool in its current state has been able to flag various extensions posing security and privacy threats. We argue that extensions review processes should also consider the different security and privacy threats we have identified, in order to help extension developers fix their codes before users install their extensions. Moreover, since many browsers now support the same cross-browser WebExtensions API, using a common tool to analyze extensions can help identify similar or identical potentially malicious extensions, and suggest appropriate actions to fix them. Furthermore, we think that browser vendors will gain by sharing information with one another on their extensions review processes. For instance, if an extension is flagged as malicious and removed by a vendor, this information may be shared with others so that the same extension may also be removed. Finally, we suggest that browser vendors should take appropriate means to require that extension developers provide explanations about the usage of permissions in extensions. This explanation can be given in the form of a privacy policy, which is clearly indicated in the `manifest.json` file. This policy can be shown to users when they install the extension.

7.2 Web applications developers

To some extent, web application developers can detect SOP bypass, especially when the requests are made by content scripts injected in a third party web application. By checking the presence and the value of the `Referer` header for instance when requests are received server-side, one can detect whether an AJAX request is originating from a trusted domain, and therefore authorize or prevent it. We found Twitter and Gmail preventing (responding with 403 HTTP status) requests from content scripts of third party web applications, especially when the user was logged in. However, we found no way one can prevent SOP bypass, when extensions allow the attacker to inject code directly in the web application he needs to access. This was the case for almost all extensions we identified as allowing web applications to execute arbitrary code in their context.

7.3 Extensions developers

Most of the issues we have found in extensions are imputable to extensions developers. The privileged APIs they have access to must be used with care, as they can put at serious risks, the security and privacy of users. Most of code execution can be avoided by properly sanitizing messages received from web applications. To avoid leaking user information such as browsing history, extensions can manage them in extension UI pages instead of using webpages and message passing to manage them, the reason being that an attacker script may be present on the webpage. It also seems that some of the SOP bypass are the result of poor programming practices where extensions allow SOP bypass via message passing for pages from their own domains in order to avoid supporting CORS. Unfortunately, an attacker script may also be present on these pages, or when the extension is poorly programmed, the SOP bypass could be inadvertently enabled for all web applications.

7.4 Extensions users

Finally, to users, we suggest to always log out from web applications. This may limit leaking cookies and data of applications where the user is logged in. By default, extensions are not allowed in incognito and other private browsing modes. Browsing in such modes, without any extensions enabled, surely protects users from many of the security and privacy threats shown in this work.

8 Conclusion

Web applications and browser extensions can interact with one another by exchanging messages. In this work, we built a static analyzer and applied it to Chrome, Firefox and Opera extensions. We identified a good number of extensions that enable web applications to benefit from their privileged capabilities. In particular, some extensions allow web applications to access any other application data, thereby bypassing the Same Origin Policy security mechanism. Extensions also leaked user credentials (cookies), browsing history, bookmarks, list of installed extensions, to web applications or allowed them to download any file on the user device, or store data in the extension storage for tracking purposes. We showed how trivially, attackers can exploit those threats, and argued that browser vendors should take this into consideration while reviewing extensions. The static analysis tool we have used in this work, could be used to help detect such extensions, and fix or remove them from browsers.

Chapter 9

Breaking the Same Origin Policy ! On CORS headers manipulations by browser extensions

Preamble

This chapter analyzes CORS headers manipulations by browser extensions. It is under submission.

1 Introduction

Extensions can intercept and modify HTTP communications, in particular the requests and responses headers between web applications running in the user browser, and web servers [28, 59]. The `User-Agent Switcher for Chrome` Chrome extension [147] for instance, uses this capability to simulate different browsers, by modifying the `User-Agent` request header. When an extension has the capability to manipulate HTTP communications, it can do so with almost any HTTP request and response header. It can remove headers, change their values, or even add new ones. Nonetheless, as different HTTP headers are used for different purposes, tampering with HTTP headers can have various implications from a security perspective. The `X-Frame-Options` header for instance, is set by web servers in HTTP responses to fight against clickjacking attacks [244]. Therefore, removing this header enables clickjacking attacks. Similarly, the `Content-Security-Policy` is used by web applications to deploy Content Security Policies (CSPs) [272, 275] to mitigate content injection attacks such as XSS (Cross-Site Scripting) [41]. Hence, tampering with `Content-Security-Policy` may enable XSS attacks, as it has been demonstrated by Kapravelos et al. [213] and Hausknecht et al. [200].

In this work, we consider the implications of manipulating Cross-Origin Resource Sharing (CORS) HTTP headers. With their ability to modify HTTP headers, extensions can also modify CORS headers. Intuitively, an extension can change or add the appropriate headers in HTTP requests and responses, in order to always make unauthorized CORS requests successful. Doing so allows any web application to directly access data, including user sensitive data on any other web application server, thereby breaking the Same Origin Policy (SOP). Moreover, if extensions do not correctly handle CORS headers, they can potentially break legitimate cross-origin requests, even though web servers allow them, thereby breaking the functionality of web applications that the user is interacting with.

We analyzed the WebExtensions API, the cross-browser extensions system compatible with major browsers including Chrome, Opera, Firefox and Microsoft Edge [2, 25, 100, 110]. We first of all wanted to assess whether tampering with CORS headers in a extension,

is considered a security threat from the perspective of browser vendors. To do so, we developed `CORSER`, a cross-browser WebExtension. If crafting such an extension takes little effort, it however requires a correct understanding of the CORS mechanism. We then decided to publish `CORSER` on different browsers in order to assess whether browser vendors consider tampering with CORS headers a security threat. From the perspective of browser vendors, `CORSER` is a benign extension as it successfully passed the different extension review processes. On Firefox, the extension was made public only a few seconds after we submitted it for review [37]. On Opera, it was published a few minutes after the review process started [38]. Finally, on Chrome, the extension was made public on the same day [36].

We also performed an empirical analysis of extensions in the wild. Among the capabilities or permissions usually requested by extensions, is the ability to intercept and manipulate HTTP headers, thus CORS headers, for any web application. We therefore built a static analyzer that flags suspicious extensions potentially tampering with CORS headers. Then, we manually vetted such extensions and found dozens of them effectively manipulating CORS headers. This was done mostly to authorize unauthorized CORS requests. But more surprisingly, we also found that the CORS mechanism is widely misunderstood among extensions developers, as most of the extensions that manipulate CORS headers do it in a way that break legitimate CORS requests made by web applications running in the user browser.

In summary, we make the following contributions

— With a in-depth understanding of the CORS mechanism, we developed the `CORSER` extension that tampers with CORS headers to disable the Same Origin Policy in browsers by authorizing unauthorized cross-origin requests.

— We submitted `CORSER` for review on different browsers, and it successfully passed their extensions review processes and was published on Chrome, Firefox and Opera.

— We found that the ability to tamper with CORS, then breaking the SOP is widespread among extensions. On Firefox, Chrome, and Opera, around 10% of all extensions have the appropriate permissions for doing so.

— Finally, we statically and manually analyzed extensions source codes. A few of them manipulate HTTP headers to break the SOP in browsers. Moreover, we also found that CORS is widely misunderstood among extensions developers. In fact, many modifications are not done correctly, causing legitimate CORS requests to fail, thereby breaking the web applications running in the user's browser.

With these findings, we discuss various countermeasures and the implications of HTTP headers manipulations by browser extensions. First, from a browser vendor perspective, we argue that the extensions review process [23, 56, 118] must take into consideration HTTP headers manipulations, in particular security critical ones such as CORS headers, `Content-Security-Policy` and `X-Frame-Options`. We propose that extensions must explicitly require dedicated permissions to be able to tamper with security critical headers. This would enable browser vendors to warn users of the underlying security and privacy threats that installing such extensions can introduce. In fact, as tampering with HTTP headers is considered benign from the browser vendors perspective, installing an extension that disables SOP would not raise any particular warning, despite the fact that tampering with HTTP headers represent a serious security threat, that users must be aware of. Our recommendation for users is to use their sensitive web applications in a browser environment where they do not have any extensions installed. From a developer's perspective, we show how to safely manipulate CORS headers. In fact, such extensions can be useful for

advanced users to perform web applications testing for instance. In these settings, extensions developers can give users more control over the extension and allow them to define when the extension will be activated, the requests that the extension can manipulate, and disable the extension when it is not in use. Finally, we discuss how web servers can fight against CORS headers manipulations done by browser extensions. A web server implementing CORS can detect CORS headers modifications as done by most of the extensions we analyzed. Nonetheless, this would imply that all servers implement CORS. Unfortunately, this breaks the backwards-compatibility of the mechanism. In fact, normally, when a web server does not support CORS, it would not return any CORS headers, in which case the browser would fallback to the default SOP, by blocking the cross-origin request.

2 Background

The capability of extensions that is of interest in this work is their ability to intercept and manipulate HTTP communications between webpages and web servers, in particular, the HTTP requests and responses headers.
We specifically consider CORS HTTP headers, whose modifications constitute a serious rollback in what makes the foundations of modern browsers security model, namely the Same Origin Policy (SOP) [125]. In its basic form, the SOP prevents cross-origin AJAX requests. CORS is a refinement of SOP, in which control is completely given to web servers, which can decide on accepting or rejecting cross-origin AJAX requests. With the ability given to extensions to manipulate HTTP headers, extensions can simply hijack web servers control over CORS requests, and always make cross-origin successful, thereby removing even the basic SOP protection from browsers, allowing any web application to make cross-origin requests to any other web server, to access user sensitive data. As shown on Fig. 9.1, HTTP requests go through browser extensions before reaching web servers, and HTTP responses also go through browser extensions before the responses are handled by the browser.

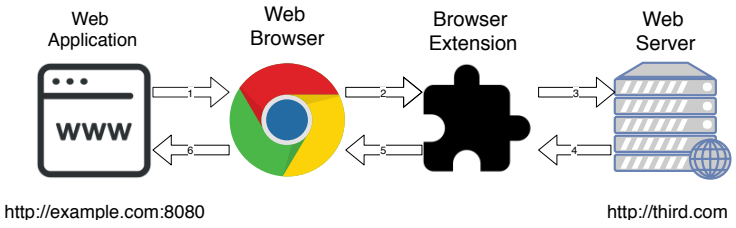


Figure 9.1 – CORS requests workflow in presence of an extension with the capability to intercept and manipulate HTTP headers

2.1 Threat model

The attacker here is any entity willing to make cross-origin requests in order to access user sensitive data, and exfiltrate them to a server under the control of the attacker. To do so, the attacker has a script in a web application running in the user browser. The user has installed an extension that manipulates HTTP headers to allow cross-origin requests, even when they are not authorized by web servers. The extension can either be poorly programmed or intentionally malicious. For instance, an extension that modifies CORS headers could be provided to developers for testing purposes. But if not well programmed, any script in the user browser could benefit from its presence to also make unauthorized

CORS requests to access user data. The attacker can also be a malicious entity that compromised a benign extension, or an extension developer willing to evade the extensions review process for user data exfiltration. In fact, even though extensions are not subject to SOP, and can make cross-origin requests to access user data, extensions would not pass the review process, if they explicitly make use of AJAX requests to exfiltrate user data directly from the extension context. As a matter of fact, Opera [118] forbids that an extension makes explicit XMLHttpRequests [155] to third parties. However, the extension developer could disable SOP with CORS headers modifications, then inject a malicious script in a web application running in the user browser, and make any cross-origin request from that application.

3 CORSER extension

In this section, we first present CORSER, a cross-browser extension that intercepts and manipulates CORS headers in order to authorize unauthorized cross-origin requests. While developing such an extension requires little effort, it nonetheless requires a good understanding of the CORS mechanism [34] and APIs available to extensions for manipulating HTTP headers [28, 59]. We then demonstrate the security implications of such an extension, as it breaks the Same Origin Policy by allowing an attacker to harvest user data on any web application, especially those where the user has logged into. Finally we show that an extension such as CORSER is considered completely benign by browser vendors, as we published it on Chrome [36], Firefox [37] and Opera [38].

3.1 Permissions to manipulate HTTP headers

To intercept and tamper with all HTTP communications, an extension has to declare the `webRequest` and `webRequestBlocking` permissions, in addition to the `host` permission for all HTTP hosts, by specifying the `<all_urls>` permission for instance.

```
{
  "manifest_version": 2,
  "name": "CORSER",
  "version": "1.0",
  "background": {
    "scripts": [
      "background.js"
    ]
  },
  "permissions": [
    "<all_urls>",
    "webRequest",
    "webRequestBlocking"
  ]
}
```

Listing 9.1 – Content of the CORSER `manifest.json` file, with the permissions to manipulate all HTTP requests

Listing 9.1 shows the full content of the `manifest.json` file of the CORSER extension. Among other things are the name of the extension, its version, permissions and the background page scripts. The background page is the main component of extensions. It executes in the background, and can make use of all the capabilities declared by the extension.

3.2 Background page

To manipulate HTTP headers, one has to register handlers (listeners) for events triggered by HTTP communications between web pages and web servers. Each HTTP request that is intercepted in extensions go through a set of stages with dedicated events that can be listened for, in order to take different actions on the HTTP headers [28, 59]. The request is assigned a unique identifier, that remains the same at the different stages of the request. The response to the request is also assigned the same identifier. This helps to link a request to its response. Even in the case of preflighted requests, the two sequential requests and their responses are considered a single request and thus assigned the same identifier.
The code to intercept and manipulate HTTP headers is defined in the background page, declared by the `background.js` file (Line 7 of Listing 9.1). The whole code of the background page script is shown in Listing 9.2

```
1
2   // Cross-browser extension API
3 chrome = chrome != null ? chrome : browser;
4
5   // HTTP requests headers to record
6 var corsRequestHeaders = {
7   "origin": "",
8   "access-control-request-method": "",
9   "access-control-request-headers": ""
10 };
11
12  // HTTP responses headers to modify
13 var corsResponseHeaders = {
14   "access-control-allow-origin": "",
15   "access-control-allow-method": "",
16   "access-control-allow-headers": "",
17   "access-control-allow-credentials": "true"
18 }
19
20  // Global object to keep track of HTTP requests
21 var savedRequestsHeaders = {};
22
23
24  // Intercepting HTTP requests headers
25 chrome.webRequest.onBeforeSendHeaders.addListener(
       requestListener, {
26     urls: ["<all_urls>"],
27     types: ["xmlhttprequest"]
28 }, ["blocking", "requestHeaders"]);
29
30
31  // Manipulating HTTP request headers
32 var requestListener = function(details){
33   var lcorsHeaders = {}
34
35   for (let header of details.requestHeaders) {
36       if (header.name.toLowerCase() in corsRequestHeaders) {
37         lcorsHeaders[header.name.toLowerCase()] = header.value;
38       }
39   }
40   if("origin" in lcorsHeaders){
```

```
41      savedRequestsHeaders[details.requestId] = lcorsHeaders
42    }
43    return {
44          requestHeaders: details.requestHeaders
45      };
46  }
47
48
49    // Intercepting HTTP response headers
50  chrome.webRequest.onHeadersReceived.addListener(responseListener
        , {
51      urls: ["<all_urls>"],
52      types: ["xmlhttprequest"]
53  }, ["blocking", "responseHeaders"]);
54
55
56
57    // Manipulating HTTP response headers
58  var responseListener = function(details){
59    if(details.requestId in savedRequestsHeaders){
60      let newResponseHeaders = []
61      for(let header of details.responseHeaders){
62        if(!(header.name.toLowerCase() in corsResponseHeaders)){
63          newResponseHeaders.push(header)
64        }
65      }
66      for(let header in savedRequestsHeaders[details.requestId]){
67        switch(header){
68          case "origin":
69            newResponseHeaders.push({
70              name: "Access-Control-Allow-Origin",
71              value: savedRequestsHeaders[details.requestId][
                    header]
72            });
73            break;
74          case "access-control-request-method":
75            newResponseHeaders.push({
76              name: "Access-Control-Allow-Methods",
77              value: savedRequestsHeaders[details.requestId][
                    header]
78            });
79            break;
80          case "access-control-request-headers":
81            newResponseHeaders.push({
82              name: "Access-Control-Allow-Headers",
83              value: savedRequestsHeaders[details.requestId][
                    header]
84            });
85            break;
86          default:
87            break;
88        }
89      }
90      newResponseHeaders.push({
91        name: "Access-Control-Allow-Credentials",
92        value: "true"
```

```
93      });
94      details.responseHeaders = newResponseHeaders
95      delete savedRequestsHeaders[details.requestId]
96    }
97    return {
98      responseHeaders: details.responseHeaders
99    }
100 }
```

Listing 9.2 – Content of the `background.js` file

The following are the main features of the `CORSER` extension. It starts with the definition of different objects, in particular the list of CORS requests that are recorded (Lines 6-10), and response headers that will be changed or added to make any CORS request successful (Lines 13-18). Then it defines a global object `savedRequestHeaders` in which intercepted CORS request headers will be saved (Line 21).

Intercepting HTTP requests The handler of the `onBeforeSendHeaders` event is the right place to intercept HTTP requests. Lines 25-28 of Listing 9.2 shows how to intercept HTTP requests by registering a listener or handler for the `onBeforeSendHeaders` event. The extension intercepts all (`<all_urls>`) headers (`requestHeaders`) of AJAX requests (type `xmlhttprequest`). The `requestListener` argument is a callback function or handler that will be invoked to manipulate the requests headers. Its definition is shown from Lines 32 to 46. In this function, we simply record the values of any CORS request header (See Table 2.2) found in the cross-origin request (Lines 36-38). The recorded request headers are then associated to the request unique identifier and saved in the global object `savedRequestHeaders` (Line 41).

Intercepting HTTP responses To safely manipulate HTTP response headers, one can do so by registering a listener for the `onHeadersReceived` event, as shown from Lines 50-53. Response headers of AJAX requests will be provided to the `responseListener` callback function, as defined in Lines 58-100. If the request was a cross-origin request, it means that we had previously saved its CORS requests headers. So, we use the request identifier in the response object to link to the CORS request headers previously saved in a global variable (Line 59). Then, to make the CORS request successful, we start by removing any CORS response headers returned by the web server (Lines 61-65), only those that will be modified or added. Then, for each recorded request header, we add its dual response header by setting the appropriate values (See Table 2.2): the `Access-Control-Allow-Origin` header is added and assigned the value of the recorded `Origin` header (Lines 68-73); the `Access-Control-Allow-Headers` header is added and assigned the value of the recorded `Access-Control-Request-Header` header (Lines 74-79); the `Access-Control-Allow-Methods` is added and assigned the value of the recorded `Access-Control-Request-Method` request; finally in order to authorize CORS requests with credentials, we add the `Access-Control-Allow-Credentials` response header, assigning it the value `true` (Lines 90-93).
After manipulating the response headers, we remove the recorded request headers from the global object `savedRequestHeaders` (Line 95) and return the new response headers (Lines 97-99) [1]. These modifications successfully authorize unauthorized CORS requests.

1. To be more precise, we could have removed the recorded request headers in the `onCompleted` event [28, 59]

3.3 Deploying and testing CORSER

The code of the CORSER extension is available online at https://github.com/mesolido/corser. First we locally deployed (in developer mode) CORSER extension on Chrome, Firefox, and Opera. We tested it to ensure that it worked as expected. In our demonstration scenario, we took http://jquery.com as a webpage where an attacker has injected some code to make AJAX requests to https://www.google.com.
After installing the extension, we navigated to the homepage of http://jquery.com. We took this page as a webpage for our experiments, mostly because it includes jQuery libraries [85] with convenient APIs for making AJAX requests (i.e. `$.get, $.ajax`). To make cross-origin AJAX requests, we used the browser console [17]. The request we made was `$.get("https://www.google.com", console.log)`. The response to the request were be displayed in the browser console [17].
We deactivated the extension and made a simple CORS request. It got blocked because https://www.google.com does not authorize cross-origin requests from http://jquery.com. Fig. 9.2 shows the error message displayed in the browser console. One of the reasons

Figure 9.2 – CORS request blocked with CORSER deactivated

why the request got blocked is because no CORS header was returned. In particular the Access-Control-Allow-Origin header was missing.
We then activated the extension and made the simple CORS request again. The request became successful as CORSER added the appropriate CORS headers. Fig. 9.3 shows only

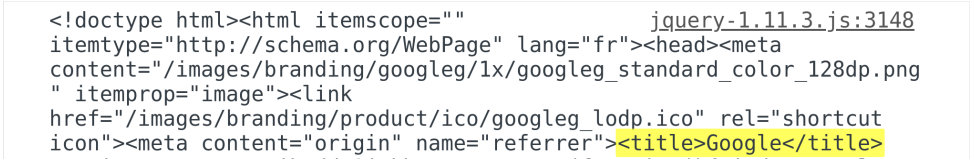


Figure 9.3 – CORSER allows CORS requests

the head of the response to the request. It is the HTML response corresponding to Google homepage (https://www.google.com).
Finally, we made a cross-origin request with credentials as shown in the following listing.

```
1 $.ajax({
2     url: "https://www.google.com",
3     xhrFields: {
4         withCredentials: true
5     },
6     success: console.log
7 });
```

When a user is logged into his Google account, the response from https://www.google.com contains the username and her email address. Fig. 9.4 highlights the user's email address in the response. We have chosen this example for the sake of simplicity. Nonetheless we used CORSER to make successful cross-origin requests in order to access more sensitive

```
"https://accounts.google.com/RemoveLocalAccount?
source=ogb\u0026Email=$email","REMOVE","SIGN
IN",0,0,1,0,1,0,0,"000770F203CEE7033E2DD070710731EA45F
D0A8779DDEDC794::1534516047527"],
[1,0,0,null,"0","emos.ereilod@gmail.com","","AD5MFfDWr
T4NzD-
```

Figure 9.4 – `CORSER` allows CORS requests with credentials

information such as reading emails from Gmail, Yahoo Mail, reading information from the user Twitter account, etc.

3.4 Publishing `CORSER`

On the 16th of August, we submitted `CORSER` for review on Chrome, Firefox and Opera, following their publishing guidelines [23, 56, 118]. In particular, all browsers required that extensions do not collect and exfiltrate user data, be self-contained, and performed as expected. `CORSER` fully complies with all these requirements.
Despite the clear threats that such an extension poses, none of the browser vendors complained about it. Firefox was the first to accept the extension. Right after it was submitted, the extension was made public [37]. We were sent an email a few days later to fix a typo (removing an extra 's') in the description on the extension. On Chrome, we had to pay a 5$ fee before publishing the extension. Afterwards, the extension was made public [36] the same day. Finally, on Opera, the review process of the extension started the day following its submission. A reviewer complained because we did not specify icons in the manifest of the extension. After updating the manifest with the icons, the extension was published right away [38]. The extension already have a few users (202 downloads on Opera, 35 on Firefox and 7 on Chrome, as of the 13th of October 2018).

4 Empirical study on CORS headers manipulations

In this section, we assess CORS headers manipulations in the wild, in particular whether extensions tamper with CORS requests, which modifications they bring to CORS requests, and the threats that this implies.

4.1 Data collection and static analyzer

To do so, we collected Chrome, Firefox and Opera extensions [24, 58, 108]. The collection of extensions was automated using the SlimerJS browser automation tool [130]. We considered only extensions with the permissions to manipulate HTTP headers, including CORS. These are extensions that declare the `webRequest` and `webRequestBlocking` permissions in their `manifest.json` file. We then applied a static analyzer to scripts of the extensions background pages. The static analyzer was written in Node.js [105], using various modules. In particular, the Jsdom HTML parser [69] was used to parse and extract scripts from extensions background pages which are declared as HTML files in the `manifest.json`. The Esprima [204] parser was used for parsing the background pages scripts. It produced an Abstract Syntax Tree (AST) of each script, from which we collected the `Literal` constructs (i.e. strings, numbers). When we found at least one CORS header among the literals of a script, the extension was considered suspicious and flagged for further manual analysis.

Limitations We considered only background scripts that are bundled in extensions packages. In fact, browser vendors recommend that extensions be self-contained. To do so, extensions are applied a default restrictive Content Security Policy (CSP) which bans external libraries by default [31, 32, 118]. However, an extension developer may still relax the default CSP of extensions, in order to dynamically load external scripts in background pages. We therefore miss extensions whose code for manipulating CORS headers might have been loaded from an external library. Our results therefore represent a lower bound of extensions that may be tampering with CORS headers.

4.2 Manual analysis

The manual analysis always consisted in first reviewing the code of suspicious extensions to assess whether they manipulated CORS headers, which headers were manipulated and how they did so, the web pages whose CORS requests were intercepted, the URLs that were intercepted, and if a user action was required to activate the ability of the extension to manipulate CORS headers. For the code review, we installed CRX Viewer [277], a convenient extension for navigating extensions bundles directly in the browser. Then, we installed the extension in the browser, interacted with it in order to confirm that it effectively tampered with CORS headers. The interactions consisted in making cross-origin requests to different hosts declared in the extension `manifest.json`. For more control over the HTTP communications, we setup our own local web server implementing the CORS mechanism. It enabled us to trigger different behaviors of suspicious extensions regarding headers manipulations. For extensions we could not confirm just by installing and interacting with them, we would download their package, patch the background scripts with hooks, before reinstalling and debugging the extensions. The hooks were basically additional lines of codes added to the extension code that helped us understand how the extension works, and which requests were to be made to trigger CORS headers manipulations.

Limitations Despite much efforts, for a few extensions, we could not successfully confirm our suspicion of whether they tampered with CORS headers or not. There are two reasons for that.

— The extension code was obfuscated, and we did not find a way to deobfuscate and analyze it. For instance, we found extensions written mostly in hexadecimal. The static analyzer successfully found CORS headers among their literals. However, by interacting with them, the CORS headers was not manipulated, but we cannot confirm this, as we could not manually review their codes.

— Or the extension had listeners that clearly tampered with CORS headers, but they were not triggered when we interacted with the extension. This is the case for instance for the `ZenMate VPN`, a very popular extension on Chrome [158], Firefox [160] and Opera [159]. It has a response listener, that if triggered, would have allowed any cross-origin requests.

4.3 Results overview

We crawled the extensions in the middle of May 2018. Table 9.1 shows an overview of the data we collected and analyzed, and Table 9.2 presents the top 11 most requested permissions. Chrome provides the largest share of extensions (66,401) followed by Firefox (9,391), then Opera, which totalizes 2,523 extensions. It is worth noting that a good number of extensions do not request any permission. That is 17.52%, 17.32% and 8.08% of Chrome, Firefox and Opera extensions respectively.

	Chrome	Firefox	Opera
Extensions	66,401	9,391	2,523
Declare no permissions	11,632	1,627	204
Have webRequest(Blocking) permissions	6,316	1,152	320
Have all hosts permission	5,031	893	265
Extensions analyzed	101	30	5
Tamper with CORS headers	51	11	1
Unconfirmed	5	1	1

Table 9.1 – Data collection and analysis results overview

Permission	**Chrome**	**Firefox**	**Opera**
tabs	49.29	44.19	54.93
storage	37.52	49.91	43.08
activeTab	18.85	25.08	10.90
http://*/*	15.41	8.66	17.24
https://*/*	14.49	8.76	16.53
contextMenus	14.30	17.40	21.56
notifications	14.12	13.36	16.37
`webRequest`	**12.10**	**17.18**	**16.17**
<all_urls>	12.07	18.36	18.15
cookies	9.66	7.55	8.40
`webRequestBlocking`	**9.58**	**12.29**	**12.76**

Table 9.2 – Most requested permissions among Chrome, Firefox and Opera extensions

Extensions with HTTP headers manipulation capabilities

As one can see from Table 9.2, all hosts (`<all_urls>`, `http://*/*` or `https://*/*`), `webRequest`, and `webRequestBlocking` are among the most requested permissions. Extensions that declare both the `webRequest` and `webRequestBlocking` permissions, giving them the ability to tamper with HTTP headers represent 9.51%, 9.51% and 12.63% of Chrome, Firefox and Opera extensions respectively (See Table 9.1). This represents all extensions which can potentially tamper with CORS headers in order to disable the Same Origin Policy in browsers and authorize cross-origin requests. It is simply a coincidence (and not an error) that Chrome and Firefox have the same share of extensions that can manipulate HTTP requests. It is worth mentioning that among these extensions with headers manipulation capabilities, 79.65%, 77.52% and 82.81% of them on Chrome, Firefox and Opera respectively, can do so for any HTTP request, as they also declare all hosts (`<all_urls>`, `http://*/*` or `https://*/*`) permissions, along with the `webRequest` and `webRequestBlocking` permissions.

Chrome	Productivity (33.35%), Photos (11.85%), Developer Tools (11.55%)
Firefox	Privacy & Security (20.33%), Social & Communication (14.79%), Web Development (13.79%)
Opera	Productivity (26.33%), Privacy & Security (21.32%), Social (13.79%)

Table 9.3 – Top 3 most popular categories among extensions with the ability to manipulate HTTP headers

Categories of extensions Table 9.3 presents for each browser the top 3 most popular categories among extensions with capabilities to tamper with HTTP headers. Productivity is the most popular category among Chrome and Opera extensions. This category however does not exist on Firefox, where Privacy & Security is the most popular category. Privacy & Security category is also the second most popular on Opera. It includes adblockers and privacy extensions such as Adblock, Ghostery and uBlock Origin, which are popular on Chrome, Firefox and Opera. There is no Privacy & Security category on Chrome. Nonetheless, most of the Privacy & Security extensions are classified in the Productivity category, which is the most popular category on Chrome. The fact that privacy and security extensions tamper with HTTP headers is something that can be easily understood because most of them intercept HTTP communications in order to block advertisements and other trackers. Social and Developer categories are also popular among extensions with the ability to manipulate HTTP headers. In particular, Developer category is the top third category on Chrome and Firefox. Among those are extensions used by developers for testing purposes such as user agent switchers. The extensions `User-Agent Switcher for Chrome` [147], `User-Agent Switcher` [146] on Opera and `User-Agent Switcher` [146] on Firefox tamper with the `User-Agent` request header to simulate different browsers. It is surprising however that Social and Photos categories appeared as one of the most popular category of extensions with the ability to tamper with HTTP headers. In the Social category are different helper extensions for social networks and applications such as Facebook, Twitter, Whatsapp. This can be explained by the fact that these extensions often require such capability to tamper with (remove) headers like `X-Frame-Options` header, in

order to frame the social network sites in the extension UI pages, allowing users to log into their social networks from these extensions.

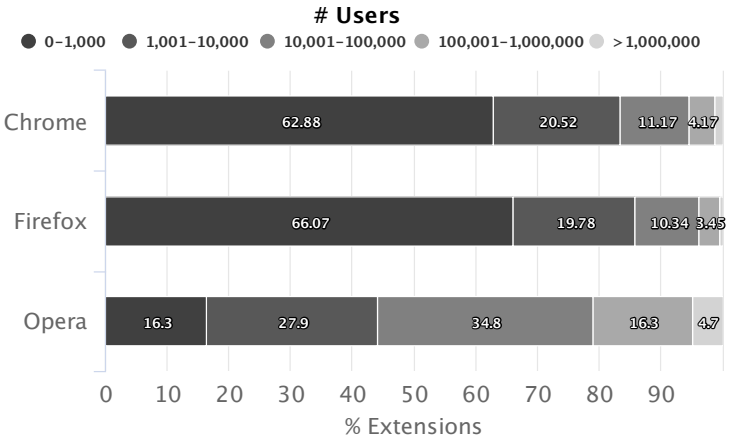


Figure 9.5 – Distribution of users of extensions with the capability to tamper with CORS headers

Distribution of users Figure 9.5 shows the distribution of the users of extensions with the ability to manipulate HTTP headers. On Chrome and Firefox, the majority of the extensions have less than a thousand users, while on Opera the distribution of users is more even. It is worth mentioning that a few extensions have millions of users. It the case for 1.26%, 0.36% and 4.7% of Chrome, Firefox and Opera extensions respectively. Compared to Chrome and Opera, the few Firefox extensions with millions of users can be explained by the fact that the WebExtensions API [100] is relatively new on Firefox where the XPCOM [156] API was long used for extensions development. The most popular extensions include adblockers, tracker blockers such as uBlock Origin, Ghostery, Adblock.

Extensions that effectively manipulate CORS headers

The static analyzer flagged 136 extensions for manual analysis (See Table 9.1): 101 on Chrome, 30 on Firefox and 5 on Opera. After manual vetting, we confirm that 63 of them (51 on Chrome, 11 on Firefox and 1 on Opera) were effectively modifying CORS headers. This represents almost half of the total extensions flagged for manual analysis. All extensions reported here have been heavily tested, and were effectively tampering with CORS headers. On Chrome, we found 4 extensions written in hexadecimal, for which we could not draw any conclusion regarding CORS headers manipulations. The other unconfirmed extension was the `ZenMate VPN` extension, present on Chrome, Firefox, and Opera. It had a response listener that adds many CORS response headers, but by interacting with it, we could not trigger this listener. The extension had quite a large code base, which further made it hard to review.

Origins and targets of the CORS requests Table 9.4 shows the webpages and web servers whose communications are intercepted and modified in order to authorize CORS requests. The majority of extensions (52) allow any webpage to make unauthorized cross-origin requests to any web application server (29 extensions) or to one or more servers

	To any server	To one or more servers
From any webpage	29	23
From one or more webpages	4	7

Table 9.4 – Sources (origins of webpages) and targets (web applications servers) of CORS requests allowed by extensions. The table reads from row to column

	Chrome	Firefox	Opera	Total
User Action (click)	10	1	1	12
User defined settings	3	2	0	5
Total	13	3	1	17

Table 9.5 – User action to enable extensions

(23). The remaining 11 extensions allow one or more webpages to connect to any server (4 extensions) or to one ore more web application servers (7 extensions). As one can see, the majority of extensions allow CORS requests for any webpage, which is rather worrisome.

User action required to activate extension We further analyzed the extensions to see whether they required user action to be activated before they started manipulating HTTP requests. Table 9.5 presents the results. Only 17 extensions require a user action to be activated. All other extensions, once installed, start tampering with CORS headers. In most of the cases (12 extensions), the user action is a click to activate the extension. Nonetheless, 5 extensions give the user more control over the extension settings by allowing him to define the webpages and web servers whose HTTP communications will be manipulated. We think that this is the right way to go regarding CORS headers manipulation. As these headers are sensitive, it is preferable not to enable the functionality by default in extensions, but rather give control to the user to decide when such feature must be enabled. We discuss, in Sections 5 and 6, different guidelines on how to safely manipulate CORS headers to testing purposes.

Users of extensions Figure 9.6 shows the distribution of users concerned with these threats. Most of the extensions have less than a thousand users. Some are relatively

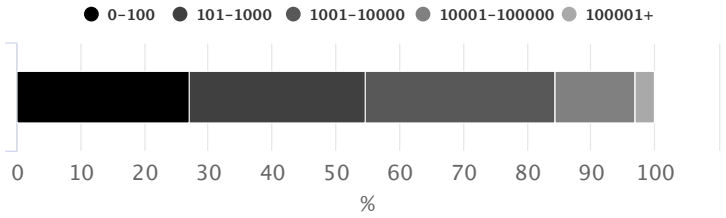


Figure 9.6 – Distribution of users of extensions manipulating CORS headers

popular, with thousands of users, with 3 extensions having more than 100k users. The most popular extension is the `Allow-Control-Allow-Origin: *` Chrome extension [5], which totalizes 455,875 users. Interestingly, this extension does not correctly tamper with CORS headers. It breaks legitimate CORS requests with credentials. For instance, after installing the extension, we could no longer play Youtube videos. This is also the case for

the only Opera extension, CORS Toggle [35] we found tampering with CORS headers. See Section 4.5 for more details.

Categories of Extensions Figure 9.7 presents the categories of extensions we found effectively tampering with CORS headers. Interestingly, the `Developer Tools` category is

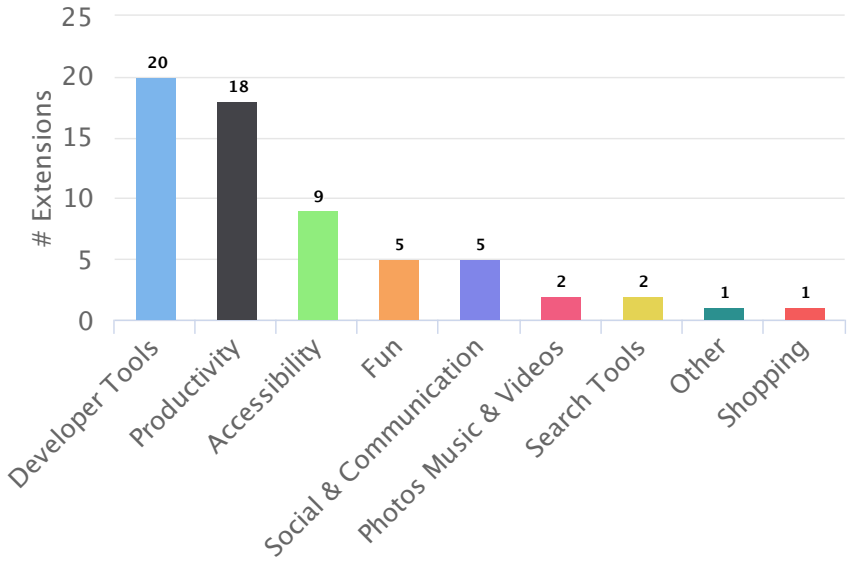


Figure 9.7 – Categories of extensions manipulating CORS headers

the most popular, followed by `Productivity` and `Accessibility` categories. However, it is less than 1/3 of extensions which are in this `Developer Tools` category, suggesting that CORS headers modifications could be more diffused, and not only limited to extensions provided for advanced users (developers) for testing purposes for instance.

Implications of HTTP headers manipulations Manipulating CORS headers has 2 main implications. The first implication is that if the CORS mechanism is well understood and the manipulations are well done, as in the case of the `CORSER` extension presented in Section 3, then the extension authorizes unauthorized cross-origin requests, thereby allowing web applications to bypass the Same Origin Policy by accessing data of cross-origin web applications servers, even though such servers do not authorize cross-origin accesses. The second implication is that, without a good understanding of the CORS mechanism, an extension can break legitimate requests made from web applications that the user is currently interacting with. As an example, if a webpage makes a cross-origin request to a server that authorizes the request (the server responds with an `Access-Control-Allow-Origin` header setting its value to the origin of the page which makes the request, and the `Access-Control-Allow-Credentials` response header whose value is set to `true`), and an extension modifies the `Access-Control-Allow-Origin` header to `*`, then it breaks the legitimate request (See Section 2 for more details about the CORS mechanism).

4.4 Breaking the Same Origin Policy

Table 9.6 shows how the extensions we have analyzed tamper with CORS HTTP headers, and which types of CORS requests (simple, preflighted) they authorize.

	Chrome	Firefox	Opera	Total
Simple	51	11	1	63
Preflighted	29	6	1	36
With credentials	7	3	0	10
Total	51	11	1	63

Table 9.6 – Extensions breaking the Same Origin Policy - Types of authorized CORS requests

Simple requests All extensions modify requests as to authorize at least simple CORS requests without credentials. This is achieved by adding the `Access-Control-Allow-Origin` header in responses headers, setting its value to `*`, or to the origin of the webpage which makes the cross-origin request. It is worth mentioning the case of the `Disable CORS` Chrome extension [46]. In fact, it changes the value of `Access-Control-Allow-Origin` response headers to the origin of the webpage from which a cross-origin request is made. Hence, it requires that the server first responds with an `Access-Control-Allow-Origin` header in order to change its value to the origin of the page. It still bypasses SOP, because a server can respond with a value of `Access-Control-Allow-Origin` set to an origin different from that of the webpage making the request. Without this extension, such CORS request would have failed. By changing the value of the header to the origin of the request, the request becomes authorized.

Preflighted requests Half of the extensions (35) also modify CORS headers in order to allow preflighted requests. In most of the cases, in order to allow preflighted requests, extensions would add the `Access-Control-Allow-Methods` header in responses, by setting its value to the most commonly used HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`). For HTTP headers, most extensions also add the `Access-Control-Allow-Headers` header with a predefined set of known HTTP headers. A very few of them reflect the value of the `Access-Control-Request-Header` in the response, in order to allow requests with custom headers.

Requests with credentials From a security perspective, simple or preflighted CORS requests without credentials are less sensitive, as they are not made with user credentials (See Fig. 9.3). Even though they represent unauthorized accesses to web servers, and break the Same Origin Policy (because browsers must not allow cross-origin requests unless the web servers respond with the appropriate CORS headers), the responses to these CORS requests could potentially be obtained by other means, for instance by using a proxy server, to simulate simple and preflighted CORS requests.
The most concerning issues, from a user perspective, are extensions that allow CORS requests with user credentials. We found 10 such extensions (7 on Chrome, and 3 on Firefox) which allow CORS requests using user credentials. Fig. 9.4 demonstrates a CORS request with credentials, where any webpage can access the user google account information (in this example, it is the email of the user).

4.5 Breaking legitimate CORS requests

The second concern we found among extensions that modify CORS headers, is the breakage of legitimate web applications. Table 9.7 present extensions which can break some

legitimate CORS requests, because of harsh modifications done on HTTP requests.

	Chrome	Firefox	Opera	Total
Simple	3	1	0	4
Preflighted	30	5	0	35
with credentials	36	8	1	45
Total	46	9	1	56

Table 9.7 – Breaking web applications, because of a harsh modification of CORS headers by extensions.

Figure 9.8 – Breaking legitimate CORS requests by adding multiple values to the `Access-Control-Allow-Origin` header

We found 2 extensions on Chrome and 1 on Firefox that add * as a second value to the `Access-Control-Allow-Origin` response header when a server responds with the `Access-Control-Allow-Origin` header. This causes the requests to fail, because the `Access-Control-Allow-Origin` header cannot have 2 values: either it is *, or an origin. Fig. 9.8 shows a message displayed in browser console in the case of such an error.
Another case is that of the `CloudExtend Gmail for NetSuite` Chrome extension [30]. It fixes the value of the `Access-Control-Allow-Origin` to a specific origin, regardless of the origin of the webpage making the request. Therefore, web pages whose origins are different from the fixed one, can no longer legitimately connect to the web servers whose responses are modified by the extension.
For preflighted requests, the breakages are due to the fact that many extensions fix the list of methods they set as values to the `Access-Control-Allow-Methods` header, or the list of headers set to the `Access-Control-Allow-Headers` header. This prevents web applications from making CORS requests with methods or headers other than the predefined ones. Even though web servers accept the custom headers and methods, they are overwritten in the responses by the extension, causing a mismatch, then a failure of the preflighted request.

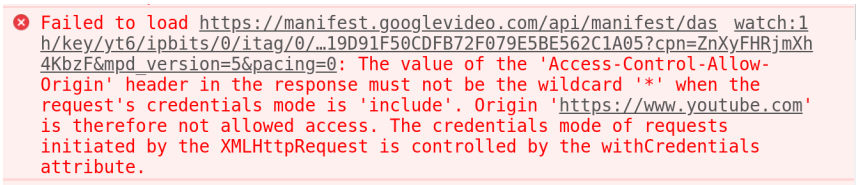


Figure 9.9 – Breaking legitimate CORS requests with credentials by changing `Access-Control-Allow-Origin` to *.

Finally, the vast majority of legitimate CORS breakage are due to a misunderstanding of CORS requests with credentials. Most of the requests fail because extensions change the value of the `Access-Control-Allow-Origin` header to *, even in presence of CORS

requests with credentials. This prevents legitimate extensions from making CORS requests with credentials. In fact, to allow CORS requests with credentials, the value of the `Access-Control-Allow-Origin` header must be the origin of the webpage which issued the request, and not `*`. Fig. 9.9 shows an error displayed in Google Chrome console when CORS requests with credentials fail because of the value of the `Access-Control-Allow-Origin` being set to `*`.
At `https://swexts.000webhostapp.com/cors/youtube.webm`, one can view a video demonstrating how the `Access-Control-Allow-Origin:*` extension on Chrome [5], having more than 400k users, break the Youtube application. Installing this extension makes it impossible to play Youtube videos. This is because it breaks CORS requests with credentials. Fig. 9.9 shows an error message displayed in the console regarding this error. The `CORS Toggle` [35] extension on Opera, also poses the same issue.

5 Discussions

The main question we raise here is, must all HTTP headers be equally treated ? For instance, changing the value of the `Server` header, which indicates the name of the server hosting a resource, does not have the same implications from a security perspective, as tampering with CORS headers, as we have shown throughout this work. Security critical headers manipulations by browser extensions is a threat that users must be aware of when installing an extension that has this capability. It is therefore important to consider the manipulations of security critical headers as a threat, and review extensions accordingly, so as to identify them and warn users at install time. However, we acknowledge that this may be a daunting task from a browser vendors perspective, because extensions may be obfuscated in order to evade such review process. Therefore, we think that the ability to tamper with security critical headers must not be automatically granted to extensions when they have the permission to tamper with HTTP headers with the `webRequest` and `webRequestBlocking` permissions. By security critical headers, we include CORS headers (See Table 2.2). We also include `Content-Security-Policy` [275] which is used to fight against content injection attacks and in particular the notorious XSS [41, 135], `X-Frame-Options` which helps to mitigate clickjacking attacks [244], and any header that could potentially introduce serious security threats in web applications. We propose 2 possible directions, from a browser vendor perspective: (i) either disallow the manipulation of CORS and other security critical headers, as also suggested by Kaparavelos et al. [213], otherwise (ii) browser vendors should require that extensions explicitly request a dedicated permission for the header that they need to tamper with.

5.1 Disallowing security headers manipulations

It is not clear why a benign extension may require to modify headers such as CORS. In fact, extensions are not subject to the Same Origin Policy. That is, cross-origin requests from an extension are allowed, regardless of whether the targeted server responds with CORS headers or not. However, extensions that make cross-origin requests may be considered suspicious by browser vendors during the extension review process. Hence, an extension can first disable the Same Origin Policy via CORS headers manipulation, then inject a script in a webpage from where it can unsuspiciously make cross-origin requests without being considered a malicious extension during the extensions review process. This is exactly what we have achieved with the `CORSER` extension.
Because of the security threats posed by extensions tampering with CORS headers, we ar-

gue that manipulating security headers must be forbidden by default in browser extensions. This has been recommended by Kaparavelos et al. [213] regarding the `Content-Security-Policy` and `X-Frame-Options` headers, even though no browser vendor has so far implemented this. Moreover, browser vendors already do not allow extensions to tamper with a few HTTP request headers, including `Authorization`, `Host`, `Cache-Control`, `If-Modified-Since`, `etc.` [28,60]. To prevent extensions from modifying the restricted headers, the headers are simply not included in the list of headers passed to the different events triggered by HTTP communications, such as the `onBeforeSendHeaders` event listener, where HTTP requests headers are usually modified. Similarly, to prevent extensions from tampering with security headers (CORS headers, `Content-Security-Policy`, `X-Frame-Options`), browser vendors may choose not to expose them to the appropriate request and response event handlers.

5.2 Requesting permissions to manipulate security headers

Our second proposal towards mitigating the threats introduced by extensions tampering with HTTP security headers, is to require that browser extensions explicitly request permissions to be able to tamper with a sensitive header. Hence, in addition to the `webRequest` and `webRequestBlocking` permissions required to intercept HTTP requests, the ability to manipulate sensitive headers would be granted to an extension only if it has explicitly requested that permission, by including the name of the sensitive header in the list of permissions of the extension `manifest.json` file as shown in Listing 9.3 below.

```
"permissions": [
    "<all_urls>",
    "webRequest",
    "webRequestBlocking",
    "access-control-allow-origin"
  ]
```

Listing 9.3 – Permissions to manipulate the `Access-Control-Allow-Origin` CORS response header

The advantage of this proposal is that it gives browser vendors the possibility to warn users of the security implications of installing an extension that can tamper with HTTP security headers. In fact, we argue that web browser vendors must warn users about extensions tampering with security headers such as CORS. Unfortunately, when installing an extension which could potentially break the Same Origin Policy, no browser vendor would warn the user about such threats. This is mainly due to the fact that headers modifications is not taken into consideration as a security threat when extensions are reviewed.
From an implementation perspective, and for backwards compatibility, if an extension does not have the permission to modify a header, the modifications that it does will be simply ignored. If one argues that this breaks the functionality of the extension, in reality, when multiple extensions tamper with HTTP headers, each extension can alter and potentially revert the modifications done by other extensions that manipulated the headers before it. Moreover, there is no guarantee on the order in which extensions will be passed the HTTP headers [28]. So the fact that the browser ignores the modifications done by an extension could be potentially achieved if the user has installed an extension which reverts the modifications done by other extensions.

6 Countermeasures

In this section, we discuss different proposals for web applications and users to fight against the threats introduced with CORS headers manipulation by browser extensions.

6.1 Web applications servers

The main problem with the ability for extensions to tamper with CORS headers is the fact that extensions break the backwards compatibility of the CORS mechanism. In fact, in a normal setting, in order to authorize CORS requests, both browsers and web servers must implement the mechanism. However, if either the browser or the server does not implement CORS, then the SOP applies, in which case, cross-origin requests are forbidden by default. Now with the ability of extensions to tamper with CORS headers, the response returned by the server can be delivered to webpages if an extension adds the appropriate CORS headers. To mitigate this, a server must then implement CORS by default. Then, it must try to ensure that the request is effectively originating from a trusted webpage before responding with data. This can be achieved for instance, by an exchange of tokens prior to authorizing cross-origin requests, as in the case of mitigation of CSRF (Cross-Site Request Forgery) attacks [135]. Web servers can also check the values of the `Origin` and `Referer` to ensure that they are set in the request, they match each other and are from a trusted origin. We found that the majority of extensions that tamper with CORS do not modify these headers, meaning that a server implementing CORS can detect the effective origin of the request. Moreover, a few extensions change the value of `Origin` header to cause a mismatch with the `Referer` header. As a general recommendation, when a cross-origin access is not authorized, a web server should not respond with data.

6.2 Extensions Users

From a user perspective, the main protection against the impact of extensions tampering with CORS, is to log out of web applications, and clear all authentication credentials. Ultimately, one can use browser environments where no extension is installed (browser private or incognito mode) or even a browser with no extension, in order to interact with sensitive web applications.

6.3 Extensions Developers

For extensions developers, there is a need for a better understanding of the CORS workflow, as we have shown in Section 2. Particular attention is to be paid to preflighted requests, and requests with credentials, so as not to break legitimate web applications. Extensions such as `CORSER`, must be used by advanced users only to perform controlled experiments such as testing web applications for instance. Ultimately, full control must be given to the user to define when to activate the extension, on which pages the requests that can be made. One can take as a starting point our `CORSER` extension (https://github.com/mesolido/corser/) and further customize it for the needs of web applications testing.

7 Conclusion

In this chapter, we first showed how trivially, extensions can manipulate CORS headers, and their implications on browser security, the security and functionality of web applications, and the security and privacy of user data. We developed and published the `CORSER`

extension on Chrome, Firefox and Opera. In doing so, we demonstrated that despite the fact that CORS headers manipulation disabled the Same Origin Policy security mechanism in browsers, this practice was not identified as a security threat by browser vendors. We analyzed extensions in the wild, and found a few of them effectively tampering with CORS headers so as to authorize unauthorized cross-origin requests. Furthermore, we found that CORS is often misunderstood as extensions developers harshly manipulate CORS headers, thereby breaking legitimate web applications. To mitigate the aforementioned threats, we suggest that the ability of extensions to manipulate CORS headers be forbidden by default, otherwise extensions should require explicit permissions for the headers they can manipulate. Finally, we discussed countermeasures to fight against these threats from a user and web application perspective.

Chapter 10

Conclusion

The time is now long gone, when web applications were made of content originating only from the server of the application. Nowadays, web developers make extensive use of third party content in order to quickly build full fledged applications. Web applications provides different services to users (mailing, social networking, banking, working, etc.) and browser extensions are third party programs that serve to customize user's browser and improve her browsing experience. Hence, web applications retain and manage user data. Browsers implement the Same Origin Policy (SOP) to ensure that two or more web applications cannot directly access each other data, unless the web applications explicitly authorize the accesses via mechanism such as CORS (Cross-Origin Resource Sharing). Tightly integrated to browsers, extensions however are exempted from compliance with the Same Origin Policy and therefore have access to user data retained by web applications. They can also access many more user information and features in the browser.
In this thesis, we studied the security and privacy threats in the browser posed by web applications the user interact with, and the browser extensions that she installs.
There are many attacks targeting the web of which XSS is the most notorious. Third party tracking is the ability of an attacker to benefit from its presence in many web applications in order to track the user has she browses and build her browsing profile. Two categories of tracking: stateful of which cookie-based tracking is the most known, and device fingerprinting. In the first case, a third party stores information in the user browser and uses that to recognize her. In the second case however, the tracker rather collects information about the user browser and use that to recognize her through different browsing sessions. Malicious or poorly programmed extensions can be exploited by attackers in web applications, in order to benefit from extensions privileged capabilities and access sensitive user information.
Content Security Policy (CSP) is a W3C mechanism proposed for mitigating the impact of content injection attacks in general and in particular XSS. We performed three main studies on CSP. In a first work, we analyzed the interplay of CSP with the SOP. In fact, CSP is a page-specific that applies only to the webpage on which it is deployed. The SOP however allows same-origin webpages to access each other data. We showed that in this case, CSP can be violated by scripts in same-origin webpages without a CSP or with a different CSP. We performed an empirical study and found that indeed many web applications are subject to CSPs violations due to their same-origin webpages. We then discuss countermeasures. In another work, we scrutinized the three CSP versions, and how browser implement them. We found that a CSP that is deployed to protect a webpage, could end up being differently interpreted depending on the browser, the version of CSP it implements, and how compliant the implementation is with respect to the specification. To help developers de-

ploy effective policies that encompass all these differences in CSP versions and browsers implementations, we introduce the deployment of dependency-free policies (`DF-CSP`). This ensures that irrespective of the browser where the application runs, the policy enforced provide the same protection against attacks. Finally, previous works have identified many limitations of CSP. We reviewed the different solutions proposed in the literature and in the specification, and demonstrate that they do not successfully mitigate the identified shortcomings of CSP. Therefore, we propose to extend the CSP specification with four new extensions: a blacklisting mode in CSP, a URL arguments checker mechanism, the introduction of a new directive `disallow-redirects` for mitigating CSP bypasses based on HTTP redirections, and an efficient reporting mechanism for collecting feedback about the runtime enforcement of CSP. We show that these extensions require little modifications in the current implementation of CSP. To demonstrate this, we implemented the proposed extensions.

Turning to third party tracking, we proposed and implemented a tracking preserving architecture, that can be deployed by web developers willing to include third party content in their applications while preventing cookie-based tracking. The architecture consists in a Rewrite Server, that automatically rewrites webpages in order to redirect third party requests to a trusted Middle Party Server, which removes tracking information exchanged between browsers and third party servers.

Regarding browser extensions, we first showed that the extensions that users install and the websites they are logged into, can serve to uniquely identify and track them. To do so, we set up a website where we detect and collect the set of extensions installed in the user browser and the web applications where they are logged into. Our results show that around 55% of users are uniquely identifiable when they have at least 1 extension installed, and around 20% are uniquely identifiable when logged into at least 1 web application. Interestingly, we could uniquely identify around 90% of the users having 1 ore more extension installed and being logged into 1 ore more websites. Then, we studied the interactions between browser extensions and web applications and demonstrate that malicious or poorly programmed extensions can be exploited by web applications to benefit from extensions privileged capabilities. We found around 200 extensions that can be exploited by web applications to bypass the SOP and read other web applications data, read user cookies and mount session hijacking attacks, user browsing history, bookmarks, list of installed extensions and use them for tracking purposes or to serve advertisements, download malicious software to damage or exfiltrate user data, and store information in the extension storage and use this for tracking purposes. Finally, we demonstrate that extensions can disable the Same Origin Policy by intercepting and adding or modifying CORS headers in order to authorize cross-origin requests. We demonstrate that by publishing the `CORSER` extension on Chrome, Firefox and Opera. Even though this extension clearly disables the SOP in browsers, it was considered benign by browser vendors. We furthermore performed an empirical study and found that a large number of extensions have the ability to disable the SOP in browsers. More worryingly, we found that the CORS mechanism is widely misunderstood among extensions developers, because many of them tamper with CORS headers in a way that break legitimate web applications. To mitigate these threats, we propose more fine-grained permission systems and review process for browser extensions that can let browser vendors warn users about the threats posed that extensions they install. Users can fight against these threats by logging out of web applications, or even avoid using sensitive web applications in browsers where they have extensions installed.

Future works and general thoughts

The formalization of dependencies (Chapter 4) can be extended to account for interactions with same-origin pages (Chapter 3). Also, the semantics of a CSP deployed on a page embedded as an iframe, depends on whether the page is sandboxed by its parent or not. Browser extensions present many more security and privacy threats that can be investigated further. Some of the tools we built to conduct different works presented in this thesis can be further engineered in order to make them more usable by the public.

Appendix A

Appendix

Table A.1 – Extensions with the same code base which gives *.fliptab.io access to browsing history (get/delete), bookmarks (get), extensions (get/enable/disable/uninstall) and storage

bddmmehmgpjhhmbbmngdjhlednmkbken
cajmbfbhhfelhgolhldhhodkclpakcfe
cepmfckfppjpbkjgnpokojedlngflnca
clkodoejadlbjaopcjoijihebbgipjff
dekpebffaadijeaogggfhjemdbjgbcao
dkpndikhfepllbpaafgcelembimabofo
eeiedbnahjonkmimigblgchlefcklhok
efdddbobcofamdjmekphjlhgmcnhobbp
ehmhopjniedignnkdeijmpmodhcppgif
eilbnnflfpkhhfmhmlhflhecceajpkcj
fieoemdbopiialnojhifcndkenhjkbmm
fkpmpnljocdllgmplhnmjhjmmilbnofj
gfgchcclfmppnfoakdlhgdhnolbpiedf
glfbbjdfmmlanpikdedpjoeimlijjcjj
hmbedbiicehadpbhbipafffieolpjolh
hocncjdhccalpmblkpagbmjebkfkibbm
iamlligjelallbdddajmbojjjhadkmcf
jcffnpjkbahanenhcnhhdfopkjlpflfm
jokpapkhjeahjbkemfjfhjgcogmbcpoi
kkejopfphkmldfpdmcljfoinfcljijjf
klfeojnepdoehgddffbcjiamcjjahmgj
lbfidebeingoondbmpeapjoeeoloanak
lgphbplfjpemcghfcoajehcmikflcbbd
lmbcpiodajlbgmjbiajgcjdalgbofcbn
loggojfoonblkkhkjpijapeheoogagki
lpkfidfkgflpbakdnhpojiejlpdanknh
mgmodhbknbfmpjmilankiffnjbelcipo
mibaeahdcconphmdndbeipegldkkbcjh
odpiaedkmdpcheddbkilnkelhhocoenn
pfdaccgdljiifplhfnjcacapfedngonb
afddmpnodjaifgjibafjcbfaplnoipei

Table A.2 – Extensions with the same code base for triggering downloads from vk.com, *.vimeo.com, *.coub.com, *.kinopoisk.ru

nfhipbkhabgmkhahoaagkcgppcjikjgl
idenapkfefkbknhbmfgeaclpcpbhcnbe
fnnlocjimhjpmgfjhjamdkjhemfhkhjo
lmlnplkfbiihcpkghkkmfefjdaccmbcc
kbiocjbkoohjjkkeaafiemjeidgalllh
dccmnjciogmmahaogjgkocongokmieog
ekfkljjojhnnhfedepfnbhhfjklagngk
hhfgpbjpilbbaomjmdpnfchbpipehiif
pgajmafmbajahclonccaoaoleghhnpam
ipeeopcjpgcbgnfogjlickeilmkbonen
jfpmehlefcchhhmlmennihbbihaolabk
kcollknpphnodcjdkcmgpjmlbaenabao
backekeabechifnekobfachchocbmjag
mfpbgndgoogfplejodpbhnfmaibnalkf
ojhheobonaamlhlcdngacakdcigpeokl
mienmjdbnnpaigifneeiifdbjkdgelha
amaobfendgcolppeioeageanmillkmkc

Table A.3 – Extensions with the same code base which leaks topsites, history and/or bookmarks to *.atavi.com, *.atavi.test

iglbnbabjdfaobglhonmnlkdbommiebd
knflcnelciofoghldagpknelepafjeif
lamnafpjcnoclihgpefhdbefcmjikhaj
jffjjdoccjiflmckicphblggbppfgklk
ofmacdiceehcibkfednmgpkhgfhpacgi
jpchabeoojaflbaajmjhfcfiknckabpo

Table A.4 – Extensions with the same code base, provided by Fabasoft, which give access to the current tab cookies

ajlbdflhaaflcepndpkdgejimggjcpnm
ngbcdblbfdpjgpmgfagkfofcjbnggfgn
pdhjoolhbkmlgjfedckdhiknnoabbnkk
hiejidhjgjpelfgldfhmnaoahnephhfg
icjlkccflchmagmkfidekficomdnlcig

Table A.5 – Extensions which give access to their storage to any application

eljhpoopiapggnlfcilpbihgbgbpnkgd
akhamklknibionleflabebgeikdookmp
hebabhddakflgmlhgefakkfkciijliie
ilgdjidfijkaengnhpeoneiagigajhco
ohdihpdgfenligmhnmldmiabdhflokkh
abenhehmjmoifipfpjeaejpbeeihnokp
ackpndpapmikcoklmcbigfgkiemohddk
ceogcehidijhepckebfifkpfogkajdkg
cgijoonmpaboophnagdckdcekmpfokel
dhcfokhhmhenbfmeflifppiedabfggkj
dhcmolikocplmafolinkncghmahimooh
eamjolanjdmgochipodfokkfjaeifhon
efhbachoakbcmbcmfffdgphbpcbldjac
fecipnolpdcmoidbjbnakpjgfikbnaik
gnnagpehbmfalanfjadamobejlldgedo
ijdfpccaiklfhpnamolipbjjijilmhli
khjhfgcimhcnaimdbgjbnbhcojkoceoc
niceocbendibobemckcagggppphheomc
okcfiidnmioajibmhhjpiomgejajiafa
pjjceionkajpednnegoanjjdlhbgkkpc
pjojmkmdealampgchopkfbejihpimjia

Table A.6 – Extensions which give access to their storage to specific applications

lpkhcobfjeidpkllbeagkkmmjgbmpfch	mail.google.com
eggdmhdpffgikgakkfojgiledkekfdce	mail.google.com
jmllflbhbembffempimjdbgnaodpoihh	mail.google.com
jmlnhlclbpfcbkaoaegfigepaffoankc	*.google.com,
gaoiiiehelhpkmpkolndijhiogfholcc	netflix.com
ghldlmcbffbcnoofadgcapodmpiimflj	netflix.com
jpgadigdffhcjldfkanacncocacekkie	netflix.com/watch/
peiajekggpiihnhphljoikpjeaahkdcn	beam.pro
bnfboihohdckgijdkplinpflifbbfmhm	plug.dj
aclhfmpoahihmhhacaekgcbjaeojnifa	wordix.io,
hcdfoeppbchkbbpplllggbjkkfokifej	*.vk.com/feed/
hddnlanhlmifafibmlabomkkkobcmchj	thankscoin.org
lhjajgnfmiliphkioedlmbfcdkhdhnkc	*.service-now.com
bmdlalnebjigindhobniianfmhakfelf	robertsspaceindustries.com,
dadggmdmhmfkpglkfpkjdmlendbkehoh	openvideo.droppages.com
pbpfgdgddpnbjcbpofmdanfbbigocklj	tweetdeck-enhancer
ilpkhojfiejdbkgcjbmllngjebdoehim	*.phylotree.org
cfnjeahambijfdljfacldifapdcklhnj	isogg.org
cjkbjhfhpbmnphgbppkbcidpmmbhaifa	*.player.me
ddiaadobgihkgefcaajmkjgmnjakiamn	auth.digitalkeyway.com
dienbdhbgkpddlgaceopelifcjpmkeha	*.gestionderesidencias.es
dnpdkejhfeeipmklhlkdjaoakbkjkkjn	datalane.io
gmjdaaahidcimfaipifeoekglllgdllb	chat.stackexchange.com
kfodnoaejimmmphonklghkimhnhhgbce	overlayBI.com

Table A.7 – Chrome, Firefox and Opera extensions that can be exploited by web applications access privileged APIs and sensitive user information

Extension unique identifier or name	Web applications to send messages from	Target web applications to access	**Permissions (accessible privileged API)**
		Chrome Browser	
fimckmjeammfdcpldmcigeojkkmeeian	*	*	`eval`, `host`, `storage`, `downloads`
fidaihkgnbcbkkdaoebdionfjenegede	*	*	`eval`, `host`, `storage`
hnkmipajjgbclkombnmigfnpekddlhlh	*	*	`eval`, `host`, `storage`
fajjnmbcianlnhmngmabhgkmgdindlha	*	*	`eval`, `host`, `storage`
efajnkcfjjkcodbhkhaigkffdleomnag	*	*	`eval`, `host`
hoobpdoclliidciecjifpikpnopjpmkh	*	*	`eval`, `host`
kjfjdocojijlledbaanbhpcnkoimghal	*	*	`eval`, `host`
pfofjhnkanlacmgfgjohncmgemffkldl	app.ringostat.com	*	`eval`, `host`, `cookies`, `storage`
gooecknlakggnppmhfpopneedjconjjp	lionlock.com,	*	`eval`, `host`, `storage`
bdiogkcdmlehdjfandmfaibbkkaicppk	*.delfa.com.br	*	`eval`, `host`, `storage`
pgbjjemkcflenaakhiehfdmcdnlnlpbl	www.seejay.cloud	*	`eval`, `host`, `storage`
hdanmfijddamndfaabibmcafmnhhmebi	*.hirogete.com,	*	`eval`, `host`
hpmeebiiihmjelpjmmemlihhcacflflc	*.valleyge.com	*	`eval`, `host`
oejnkhmeilmiplpmenkegjaibnjbappo	search.lilo.org,	*	`eval`, `host`
jkoegdibpkleifbkojmplebjhfllkckn	search.uselilo.org,	*	`eval`, `host`
aopfgjfeiimeioiajeknfidlljpoebgc	*	*	`host`, `cookies`
hlagecmhpppmpfdifmigdglnhcpnohib	*	*	`host`
kpgdinlfgnkbfkmffilkgmeahphehegk	*	*	`host`
bjjpnhdlhpfdebcbhdlmecafnokpjpce	*	*	`host`

Table A.7 – Chrome, Firefox and Opera extensions that can be exploited by web applications access privileged APIs and sensitive user information

Extension unique identifier or name	Web applications to send messages from	Target web applications to access	**Permissions (accessible privileged API)**
		Chrome Browser	
bmiedopcajpcehbbfglefijfmmndcaoa	*	*	`host`
jegnjmcegcpodciadcoeneecmkiccfgi	*	*	`host`
jnhibbjmekoijdjaopflcjbjieamifhh	*	*	`host`
jpkfmllgncphdgojhkbcjidgeabaible	*	*	`host`
ilcpdgfepihaomggobhmfiimflngbcoh	starthq.com,	*	`host`, `history`
jpcebpeheognnbogfkpllmmdnimjffdb	mail.google.com,	*	`host`, `management`
cnkgdfnjmgamkcpjdljdncfjcegpgcdg	mail.google.com,	*	`host`, `management`
cfddhmlokgokhcmepddjooekhmgmgfld	*.ok.ru	*	`host`, `downloads`
efhgmgomhamkkmjbgmcpgjnabcfpnaek	*.ok.ru	*	`host`, `downloads`
djhfcchmdelggndcpkgbanfhnpbbijdb	*.ok.ru	*	`host`, `downloads`
fhlkioimlijffnblckmdikkadobdmlgn	*.apistop.com	*	`host`
angncidddapgcmohkdmhidfleomhmfgi	logincat.com,	*	`host`
lndhlcaobijohmgoikmgpgbhepkbhpkl	oneom.tk	*	`host`
olpheomfiimdonpboopcailehdagfhaa	.g3user.com,	*	`host`
idkghekmllmjgnmbohakcddgcclanlca	ln.io	*	`host`
mhdhcccejcjfanablmohbpdbepdkokkj	*.gvt.com.br,	*	`host`
plfffminkgohddbooidppccppgelajfp	mp.weixin.qq.com,	*	`host`
cboekbiaoabkhgjdclenjpipclabkdga	*.apiary.io,	*	`host`
ekeefjfdbaakgbfbagacmckiedkmakem	*.salesmate.io,	mail.google.com,	`host`

Table A.7 – Chrome, Firefox and Opera extensions that can be exploited by web applications access privileged APIs and sensitive user information

Extension unique identifier or name	Web applications to send messages from	Target web applications to access	**Permissions (accessible privileged API)**
		Chrome Browser	
lbjbbkhljiimahdeknpckaoiinopofhl	*.appspot.com	mail.google.com,	`host`
ijmbknjhacbaeeoamjajoolgjgdbpkko	*.aliexpress.com,	*.google.com,	`host`
hihakjfhbmlmjdnnhegiciffjplmdhin	mail.google.com	linkedin.com,	`host`
cfbodcmobhpfbjhbennacnanbmpbcfkd	*.aliexpress.com,	appfreaker.com	`host`
ommfijfafanajffiijecdlfjlbgpmgpl	*.treesnetwork.com,	docs.google.com,	`host`
okgfglgogpkomipfflpajohdkaflndoh	ouramazinghome.com	www.google.com	`host`
iiabjaofopjooifoclbpdmffjlgbplod	blog.renren.com	*.github.com	`host`
mcdjehgaflnlmilhefigdkldfdnembhk	*.spotsetter.com	*.amazonaws.com	`host`
lfekjajdgncmkajdpiadkkhhpblngnlc	sub.watch,	zooqle.com,	`host`
gkfpnohhmkonpkkpdbebccbgnajfgpjp	squares.io/fetch,	www.nytimes.com	`host`
pkkbbimilpjmghfhhppamgigileopnkc	*	*	`cookies`
5 Fabasoft extensions (See Table A.4)	*	current tab	`cookies`
emiplbkkiabideffmpogkbbogkmofgph	*	-	`downloads`
17 extensions (See Table A.2)	vk.com,	-	`downloads`
eadbjnlpeabhbllkljhifinhfelhimha	ok.ru	-	`downloads`
ngegklmoecgejlbkiieccocmpmpmfhim	*.tribecube.com	-	`downloads`
iogibhaacmieogkdgebfbjgoofdlcmgb	*.shutterstock.com	-	`downloads`
ooeealgadmhdnhebkhhbbcmckehpomcj	animevost.org	-	`downloads`
dnohbnpecjinmdpeikpnmheeepnapfci	vtop.vit.ac.in	-	`downloads`

Table A.7 – Chrome, Firefox and Opera extensions that can be exploited by web applications access privileged APIs and sensitive user information

Extension unique identifier or name	Web applications to send messages from	Target web applications to access	**Permissions (accessible privileged API)**
		Chrome Browser	
pgmcojeijjhacgkkjaakdafmloncpema	repl.it	-	`downloads`
hacopcfnbokiahlppemnlneooamldola	hypem.com	-	`downloads`
bpkphnbpiagbpinglgejckickdgaghjo	amer...matrix.com	-	`history`
fheihcbdclkdoeadmjfggiamjgkippli	.my-lucky-star.net	-	`topSites`
llelondjpcjljnjihdflhpclcpbiaiba	*.msn.com	-	`topSites`
6 Atavi Extensions (See Table A.3)	atavi.com,	-	`history`, `bookmarks`, `topSites`
31 HD Wallpapers (See Table A.1)	fliptab.io	-	`history`, `bookmarks`, `management`, `storage`
pnbfclligibfgdknphcodpbcejnkhffp	*	-	`bookmarks`
eihbcgffjehfcgafjljohecmadcefoji	app.launch.menu	-	`bookmarks`
empgohlokhdhhchkenknobacofijiffg	app.launch.menu	-	`bookmarks`
aefmgkhgcmdljpfijlohmbhkhflmbmfi	openoox.com	-	`bookmarks`
dhjhphjhpcelebeaglljbfpipdfkhgi	.azurewebsites.net	-	`bookmarks`
jeabbgpkliknjiacfkfglknajloappkh	yeahap.com,	-	`bookmarks`
22 Extensions (See Table A.5)	*	-	`storage`
24 Extensions (See Table A.6)	mail.google.com, ...	-	`storage`
		Firefox Browser	
guretv-ver-tv	*	*	`eval`, `host`, `storage`

Table A.7 – Chrome, Firefox and Opera extensions that can be exploited by web applications access privileged APIs and sensitive user information

Extension unique identifier or name	Web applications to send messages from	Target web applications to access	**Permissions (accessible privileged API)**
		Chrome Browser	
buxenger	*	*	`eval`, `host`
bitbucket-server	*	*	`host`
logincataddon	logincat.com,	*	`host`
facebook-photo-zoom-easy	www.facebook.com	*	`host`
facebook-photo-zoom	www.facebook.com	*	`host`
markanabak-eklentisi	*.markanabak.com,	*.wipo.int,	`host`
skimdaddy	*	skimdaddy.com	`host`
the-trees-network	*.treesnetwork.com,	docs.google.com,	`host`
assina-me	*	-	`downloads`
liber-capital	*	-	`downloads`
video-downloader-1	*	-	`downloads`
openvost	animevost.org	-	`downloads`
youtube-video-download-convert	*.youtube.com	-	`downloads`
openvideo	droppages.com	-	`storage`
vgis	*.vonage.com	-	`storage`
		Opera Browser	
bmjcngclkmgpfbjcmnbidognkoocpllm	*	*	`eval`, `host`, `storage`
jnmcfakfglphcmgokeeoihifcenjjcgg	*	*	`eval`, `host`
pmpnemphhmmpkcafgpdjanghiaadfbef	*.ok.ru	*	`host`

Table A.7 – Chrome, Firefox and Opera extensions that can be exploited by web applications access privileged APIs and sensitive user information

Extension unique identifier or name	Web applications to send messages from	Target web applications to access	**Permissions (accessible privileged API)**
		Chrome Browser	
mpaghnpkgmnikepcgjddhckcedapomkp	*.ok.ru, *.vk.com,	*	`host`
bcabkcaakkjfdlodkolfagbdejhhkigp	*.lazyrobin.ru	*	`host`
bidjmocompdljmeglljcoecikgogfjbb	sub.watch,	zooqle.com,	`host`
aghgmcnoiflhcnfjkckofmjbeinjkena	vk.com,	-	`downloads`
mhjbdafcpnoapkglmldoofhhbpnogehk	vk.com,	-	`downloads`
hajlecmoacenahambneialopbpleihjn	*	-	`storage`
lkdpdiepahdagdknbbjgnadholcdgfib	tweetdeck-enhancer	-	`storage`

List of Figures

List of Tables

List of tools and websites

[1] A Monitor to Complement Content Security Policy (CSP) Expressiveness. https://swexts.000webhostapp.com/monitor/.

[2] Analyze Message Passing APIs in Browser extensions components. https://swexts.000webhostapp.com/extsanalyzer/.

[3] Building Dependency-Free Content Security Policy (DF-CSP). https://swexts.000webhostapp.com/dependencies/.

[4] CORSER - Cross-browser extension for tampering with HTTP CORS headers. https://github.com/mesolido/corser.

[5] Deploying Server-Side Tracking Protection Architecture. http://www-sop.inria.fr/members/Doliere.Some/essos/deployment.html.

[6] Webstats - Various statistics about top 10,000 Alexa sites. https://webstats.inria.fr/.

Bibliography

[1] Hypertext Transfer Protocol. https://www.ietf.org/rfc/rfc2616.txt.

[2] Microsoft Edge Extensions API. https://docs.microsoft.com/en-us/microsoft-edge/extensions.

[3] A comprehensive tutorial on cross-site scripting. https://excess-xss.com/.

[4] Abstract Syntax Tree. http://esprima.readthedocs.io/en/4.0/syntax-tree-format.html#expressions-and-patterns.

[5] Access-Control-Allow-Origin:* - Chrome Extension. https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfghkpbjhddihlkkiljbi.

[6] AdBlock - Block Ads - Browse Safe. https://getadblock.com/.

[7] Adblockplus official website. https://adblockplus.org/.

[8] AngularJS. https://angularjs.org/.

[9] APACHE HTTP SERVER PROJECT. https://httpd.apache.org/.

[10] Application programming interface. https://en.wikipedia.org/wiki/Application_programming_interface.

[11] ASP.NET Web Framework. https://www.asp.net/.

[12] Asynchronous JavaScript + XML (AJAX). https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX.

[13] Atavi - bookmark manager. https://chrome.google.com/webstore/detail/atavi-bookmark-manager/jpchabeoojaflbaajmjhfcfiknckabpo.

[14] Background Page. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/background.

[15] Boomerang for Gmail - Chrome Extension. https://chrome.google.com/webstore/detail/boomerang-for-gmail/mdanidgdpmkimeiiojknlnekblgmpdll.

[16] Brave browser. https://brave.com/.

[17] Browser Console. https://developer.mozilla.org/en-US/docs/Tools/Browser_Console.

[18] Browser History. https://chrome.google.com/webstore/detail/browser-history/bpkphnbpiagbpinglgejckickdgaghjo.

[19] Browsing Contexts. https://www.w3.org/TR/html51/browsers.html.

[20] Bug 1372288 - webextensions uuid can be used as user fingerprint. https://bugzilla.mozilla.org/show_bug.cgi?id=1372288.

[21] Can I use Content Security Policy 1.0 (Known issues). https://caniuse.com/#search=content%20security%20policy.

[22] Cascading Style Sheets. https://www.w3.org/Style/CSS/.

[23] Chrome - Publish in the Chrome Web Store. https://developer.chrome.com/webstore/publish.

[24] Chrome Extensions. https://chrome.google.com/webstore/category/extensions?hl=en-US.

[25] Chrome Extensions API. https://developer.chrome.com/extensions.

[26] Chrome Extensions API - Content scripts and Content Security Policy. https://developer.chrome.com/extensions/contentSecurityPolicy.

[27] Chrome Platform Status. https://www.chromestatus.com/metrics/feature/popularity#DocumentSetDomain.

[28] Chrome WebRequest API. https://developer.chrome.com/extensions/webRequest.

[29] CLIQZ. https://cliqz.com.

[30] CloudExtend Gmail for NetSuite - Chrome Extension. https://chrome.google.com/webstore/detail/cloudextend-gmail-for-net/fbaloimemjelmonlpfnmiipkeldlnnbl.

[31] Content Security Policy - Firefox Extensions. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_Security_Policy.

[32] Content Security Policy (CSP) - Chrome Extensions. https://developer.chrome.com/extensions/contentSecurityPolicy.

[33] CORS - Mozilla Developer Network. https://developer.mozilla.org/fr/docs/Web/HTTP/CORS.

[34] CORS protocol - Fetch Specification. https://fetch.spec.whatwg.org/#http-cors-protocol.

[35] CORS Toggle - Opera Extension. https://addons.opera.com/en/extensions/details/cors-toggle/.

[36] CORSER extension on Chrome. https://chrome.google.com/webstore/detail/corser/elgclnafddmkhhnhlfgfahgbahkginga.

[37] CORSER extension on Firefox. https://addons.mozilla.org/en-US/firefox/addon/corser-addon/.

[38] CORSER extension on Opera. https://addons.opera.com/en/extensions/details/corser-authorize-cors-requests/.

[39] Cross-Origin Communications. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[40] Cross-origin-resource sharing. https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

[41] Cross-Site-Scripting. https://www.owasp.org/index.php/Cross\protect\discretionary{\char\hyphenchar\font}{}{}site_Scripting_(XSS).

[42] CSP violations online. https://webstats.inria.fr?cspviolations.

[43] CSS Font Loading API. https://developer.mozilla.org/en-US/docs/Web/API/CSS_Font_Loading_API.

[44] CSS Parser for Node.js. https://github.com/reworkcss/css.

[45] Data URI scheme. https://en.wikipedia.org/wiki/Data_URI_scheme.

[46] Disable CORS - Chrome Extension. https://chrome.google.com/webstore/detail/disable-cors/mghlhnfeimllfjdpacagfdmchnhbgfeh.

[47] Disconnect. https://disconnect.me/.

[48] Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[49] Document.cookie. https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie.

[50] ECMAScript® 2017 Internationalization API Specification (ECMA-402, 4th Edition, June 2017). https://www.ecma-international.org/ecma-402/4.0/.

[51] eEail.in Chrome extension. https://chrome.google.com/webstore/detail/erailin/aopfgjfeiimeioiajeknfidlljpoebgc.

[52] Extensions and the add-on ID. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/WebExtensions_and_the_Add-on_ID.

[53] Faceboook website. https://www.facebook.com/.

[54] Fetch Specification. https://fetch.spec.whatwg.org/.

[55] Fetch Specification. https://fetch.spec.whatwg.org/.

[56] Firefox - Submitting an add-on. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/Distribution/Submitting_an_add-on.

[57] Firefox - Web Accessible Resources. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources.

[58] Firefox Add-ons. https://addons.mozilla.org/en-US/firefox/.

[59] Firefox WebRequest API. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest.

[60] Firefox webRequest.onBeforeSendHeaders. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/onBeforeSendHeaders.

[61] Ghostery. https://www.ghostery.com/.

[62] Google Chrome browser. https://www.google.com/chrome/.

[63] Google. Manifest - Web Accessible Resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[64] Google. Manifest File Format. https://developer.chrome.com/extensions/manifest.

[65] Google website. https://www.google.com/.

[66] Google's Gmail. https://gmail.com.

[67] GureTV: To watch television - Firefox Extension. https://addons.mozilla.org/en-US/firefox/addon/guretv-ver-tv/.

[68] HD Wallpapers from fliptab.io. http://www.fliptab.io/.

[69] HTML Parser for Node.js. https://github.com/tmpvar/jsdom.

[70] HTML Standard. https://html.spec.whatwg.org/.

[71] HTML5 Specification - W3C. https://www.w3.org/TR/html5/forms.html.

[72] HTTP Commander - Chrome Extension. https://chrome.google.com/webstore/detail/http-commander/emiplbkkiabideffmpogkbbogkmofgph.

[73] HTTP Cookies. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[74] Http cookies. https://developer.mozilla.org/fr/docs/HTTP/Cookies.

[75] HTTPS. https://en.wikipedia.org/wiki/HTTPS.

[76] Iframe Sandbox Attribute. https://www.w3.org/TR/2011/WD-html5-20110525/the-iframe-element.html#attr-iframe-sandbox.

[77] ISOGG Y-Tree AddOn - Chrome Extension. https://chrome.google.com/webstore/detail/isogg-y-tree-addon/cfnjeahambijfdljfacldifapdcklhnj.

[78] Iwassa - Chrome Extension. https://chrome.google.com/webstore/detail/iwassa/hnkmipajjgbclkombnmigfnpekddlhlh.

[79] IWASSA - Opera Extension. https://addons.opera.com/en/search/?query=bmjcngclkmgpfbjcmnbidognkoocpllm.

[80] Javascript - mozilla developer network. https://developer.mozilla.org/bm/docs/Web/JavaScript.

[81] JavaScript Object Property Access - Dot and Array Notation. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors.

[82] JavaScript Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

[83] JavaScript scope. https://developer.mozilla.org/en-US/docs/Glossary/Scope.

[84] jianlibao - Chrome Extension. https://chrome.google.com/webstore/detail/jianlibao/fimckmjeammfdcpldmcigeojkkmeeian.

[85] jQuery. http://jquery.com/.

[86] Lastpass official website. https://www.lastpass.com/business.

[87] LinkClicker - Chrome Extension. https://chrome.google.com/webstore/detail/linkclicker/hoobpdoclliidciecjifpikpnopjpmkh.

[88] LinkClicker - Opera Extension. https://addons.opera.com/en/search/?query=jnmcfakfglphcmgokeeoihifcenjjcgg.

[89] LinkedIn Sales Navigator - Chrome Extension. https://chrome.google.com/webstore/detail/linkedin-sales-navigator/hihakjfhbmlmjdnnhegiciffjplmdhin.

[90] Linkedin website. https://www.linkedin.com/.

[91] Man-in-the-middle attack. https://en.wikipedia.org/wiki/Man-in-the-middle_attack.

[92] MegaTest - Opera Extension. https://addons.opera.com/en/extensions/details/megatest-uznat-rezultat/.

[93] Message Passing - Google Chrome Extensions. https://developer.chrome.com/extensions/messaging.

[94] Microsoft Edge Extensions. https://www.microsoft.com/en-us/store/collections/edgeextensions/pc.

[95] Microsoft Internet Information Services (IIS). https://www.iis.net/.

[96] MIME types. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types.

[97] ModernDeck - Chrome Extension. https://chrome.google.com/webstore/detail/moderndeck/pbpfgdgddpnbjcbpofmdanfbbigocklj.

[98] ModernDeck - Opera Extension. https://addons.opera.com/en/search/?query=lkdpdiepahdagdknbbjgnadholcdgfib.

[99] MongoDB. https://www.mongodb.com/.

[100] Mozilla WebExtensions API. https://developer.mozilla.org/en-US/Add-ons/WebExtensions.

[101] multiDownloader - Chrome Extension. https://chrome.google.com/webstore/detail/multidownloader/dnohbnpecjinmdpeikpnmheeepnapfci.

[102] MutationObserver API . https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver.

[103] MySQL Database. https://www.mysql.com/.

[104] NGINX. https://www.nginx.com/.

[105] Node.js. https://nodejs.org/en/.

[106] Node.js Proxy. https://newspaint.wordpress.com/2012/11/05/node-js-http-and-https-proxy.

[107] Opera - Passing Messages in Extensions. https://dev.opera.com/extensions/message-passing/.

[108] Opera Add-ons. https://addons.opera.com/en/extensions/.

[109] Opera browser. http://www.opera.com/.

[110] Opera Extensions API. https://dev.opera.com/extensions/.

[111] Oracle Database. https://www.oracle.com/index.html.

[112] Phishing Attack. https://en.wikipedia.org/wiki/Phishing.

[113] PHP: Hypertext Preprocessor. http://php.net/.

[114] PhyloTreeMT AddOn - Chrome Extension. https://chrome.google.com/webstore/detail/phylotreemt-addon/ilpkhojfiejdbkgcjbmllngjebdoehim.

[115] PostgreSQL Database. https://www.postgresql.org/.

[116] PostMessage - Cross-Origin Iframe Secure Communication. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[117] Privacy Badger. https://www.eff.org/fr/privacybadger.

[118] Publishing Guidelines - Opera Extensions. https://dev.opera.com/extensions/publishing-guidelines/.

[119] Python Programming Language. https://www.python.org/.

[120] renren-markdown - Chrome Extension. https://chrome.google.com/webstore/detail/renren-markdown/iiabjaofopjooifoclbpdmffjlgbplod.

[121] repl.it download - Chrome Extension. https://chrome.google.com/webstore/detail/replit-download/pgmcojeijjhacgkkjaakdafmloncpema.

[122] Reverse Proxy. https://en.wikipedia.org/wiki/Reverse_proxy.

[123] Ringostat dialer. https://chrome.google.com/webstore/detail/ringostat-dialer/pfofjhnkanlacmgfgjohncmgemffkldl.

[124] SalesforceIQ CRM. https://chrome.google.com/webstore/detail/salesforceiq-crm/jpcebpeheognnbogfkpllmmdnimjffdb.

[125] Same Origin Policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[126] Secure Hash Algorithms. https://en.wikipedia.org/wiki/Secure_Hash_Algorithms.

[127] Server Side Access Control (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/Server-Side_Access_Control.

[128] Service Worker API. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[129] Session Hijacking Attack. https://www.owasp.org/index.php/Session_hijacking_attack.

[130] SlimerJS - A scriptable browser for Web developers. https://slimerjs.org/.

[131] Space Galaxy HD Wallpapers - Chrome Extension. https://chrome.google.com/webstore/detail/space-galaxy-hd-wallpaper/dkpndikhfepllbpaafgcelembimabofo.

[132] StartHQ. https://chrome.google.com/webstore/detail/starthq/ilcpdgfepihaomggobhmfiimflngbcoh.

[133] Telerik Test Studio Chrome Playback 2014.1. https://chrome.google.com/webstore/detail/telerik-test-studio-chrom/pkkbbimilpjmghfhhppamgigileopnkc.

[134] The Basics of Browser Helper Objects. https://blogs.msdn.microsoft.com/askie/2007/12/07/the-basics-of-browser-helper-objects/.

[135] The OWASP Top Ten Project. https://www.owasp.org/index.php/Top_10_2013-Top_10.

[136] Tor Browser. https://www.torproject.org/projects/torbrowser/design/.

[137] Tracking Compliance and Scope. https://www.w3.org/TR/tracking-compliance/.

[138] Tracking Preference Expression. https://www.w3.org/TR/tracking-dnt/.

[139] uBlock Origin. https://www.ublock.org/.

[140] uBlock Origin - Chrome Extension. https://chrome.google.com/webstore/detail/ublock-origin/cjpalhdlnbpafiamejdnhcphjbkeiagm?hl=en-US.

[141] uBlock Origin - Firefox Extension. https://addons.mozilla.org/en-US/firefox/addon/ublock-origin/?src=search.

[142] uBlock Origin - Opera Extension. https://addons.opera.com/en/search/?query=kccohkcpppjjkkjppopfnflnebibpida.

[143] URI - Uniform Resource Identifier. https://en.wikipedia.org/wiki/Uniform_Resource_Identifier.

[144] URL. https://www.w3.org/TR/url.

[145] URLSearchParams API. https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams.

[146] User-Agent Switcher - Firefox Extensions. https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher-revived/.

[147] User-Agent Switcher for Chrome - Chrome Extension. https://chrome.google.com/webstore/detail/user-agent-switcher-for-c/djflhoibgkdhkhhcedjiklpkjnoahfmg.

[148] Using CORS - HTML5 Rocks. https://www.html5rocks.com/en/tutorials/cors/.

[149] Using Service Workers. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers.

[150] Using Web Workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

[151] VisualSP Training for Office 365 - Chrome Extension. https://chrome.google.com/webstore/detail/visualsp-training-for-off/ohdihpdgfenligmhnmldmiabdhflokkh.

[152] WebExtensions web_accessible_resources. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/web_accessible_resources.

[153] Webstats - Various statistics about top 10,000 Alexa sites. https://webstats.inria.fr/.

[154] Window . https://developer.mozilla.org/en-US/docs/Web/API/Window.

[155] XMLHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[156] XPCOM Interfaces. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Tutorial/XPCOM_Interfaces.

[157] Youtube website. https://www.youtube.com/.

[158] ZenMate VPN - Best Cyber Security & Unblock - Chrome Extension. https://chrome.google.com/webstore/detail/zenmate-vpn-best-cyber-se/fdcgdnkidjaadafnichfpabhfomcebme.

[159] ZenMate VPN - Opera Extension. https://addons.opera.com/en/search/?query=cnhbkkedmelfmalgjpkngiaoifpdfcnl.

[160] ZenMate VPN for Firefox - Firefox Extension. https://addons.mozilla.org/en-US/firefox/addon/zenmate-vpn/.

[161] European Commision Law on Cookies, 2012. http://ec.europa.eu/ipg/basics/legal/cookies/index_en.htm.

[162] Webstats - Use of Content Security Policy and Cookies in top 10,000 Alexa sites, 2016. https://webstats.inria.fr/popsecurity.php.

[163] Erwan Abgrall, Yves Le Traon, Martin Monperrus, Sylvain Gombault, Mario Heiderich, and Alain Ribault. XSS-FP: browser fingerprinting using HTML parser quirks. *CoRR*, 2012.

[164] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proc. of CCS 2014*.

[165] Gunes Acar, Marc Juárez, Nick Nikiforakis, Claudia Díaz, Seda F. Gürses, Frank Piessens, and Bart Preneel. FPDetective: dusting the web for fingerprinters. In *Proc. of CCS 2013*.

[166] Jagdish Prasad Achara, Gergely Ács, and Claude Castelluccia. On the unicity of smartphone applications. *CoRR*, abs/1507.07851, 2015.

[167] Jagdish Prasad Achara, Javier Parra-Arnau, and Claude Castelluccia. Mytracking-choices: Pacifying the ad-block war by enforcing user privacy preferences. *CoRR*, 2016.

[168] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data Exfiltration in the Face of CSP. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 853–864. ACM, 2016.

[169] Tom Anthony. Detect if visitors are logged into twitter, facebook or google+. http://www.tomanthony.co.uk/blog/detect-visitor-social-networks/, 2012.

[170] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. VEX: vetting browser extensions for security vulnerabilities. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 339–354. USENIX Association, 2010.

[171] Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich, editors. *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. ACM, 2017.

[172] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[173] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. User tracking on the web via cross-browser fingerprinting. In *Proc. of the 16th NordSec*, pages 31–46, 2011.

[174] Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors. *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, volume 10379 of *Lecture Notes in Computer Science*. Springer, 2017.

[175] Matthew Bryant. Dirty browser enumeration tricks - using chrome:// and about: to detect firefox and addons. https://thehackerblog.com/dirty-browser-enumeration-tricks-using-chrome-and-about-to-detect-firefox-plugins/index.html, 2014.

[176] Stefano Calzavara, Michele Bugliesi, Silvia Crafa, and Enrico Steffinlongo. Fine-grained detection of privilege escalation attacks on browser extensions. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 510–534. Springer, 2015.

[177] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild. In Weippl et al. [268], pages 1365–1375.

[178] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. CCSP: controlled relaxation of content security policies by runtime policy composition. In Kirda and Ristenpart [215], pages 695–712.

[179] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-based analysis of content security policy deployment. *ACM Trans. Web*, 12(2):10:1–10:36, January 2017.

[180] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via os and hardware level features. In *Proc. of the 24th NDSS*, 2017.

[181] Nicholas Carlini, Adrienne Porter Felt, and David A. Wagner. An evaluation of the google chrome extension security architecture. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 97–111. USENIX Association, 2012.

[182] Giovanni Cattani. The evolution of chrome extensions detection. http://blog.beefproject.com/2013/04/the-evolution-of-chrome-extensions.html, 2013.

[183] Yves-Alexandre de Montjoye, César A. Hidalgo, Michel Verleysen, and Vincent D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific Reports*, 3:1376 EP –, 2013.

[184] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. deDacota: toward preventing server-side XSS via automatic code and data separation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1205–1216. ACM, 2013.

[185] Peter Eckersley. How Unique Is Your Web Browser? In *Proc. of the 2010 PETS*.

[186] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Xiaodong Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*, pages 233–246, 2007.

[187] Ahmed Elsobky. Novel techniques for user deanonymization attacks. https://0xsobky.github.io/novel-deanonymization-techniques/, 2016.

[188] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proc. of the 2016 CCS*, pages 1388–1401, 2016.

[189] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proc. of the 24th WWW*, pages 289–299, 2015.

[190] H. Gamboa, A. L. N. Fred, and A. K. Jain. Webbiometrics: User verification via web interaction. In *2007 Biometrics Symposium*, pages 1–6, 2007.

[191] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 309–318. ACM, 2018.

[192] Willem De Groef. *Client- and Server-Side Security Technologies for JavaScript Web Applications ; Beveiligingstechnologiën voor webapplicaties in JavaScript*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2016.

[193] Jeremiah Grossman. I know what you've got (firefox extensions). http://blog.jeremiahgrossman.com/2006/08/i-know-what-youve-got-firefox.html, 2006.

[194] Jeremiah Grossman. Login detection, whose problem is it? http://blog.jeremiahgrossman.com/2008/03/login-detection-whose-problem-is-it.html, 2008.

[195] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 115–130. IEEE Computer Society, 2011.

[196] Gábor György Gulyás, Gergely Acs, and Claude Castelluccia. Code repository for paper titled 'near-optimal fingerprinting with constraints'. https://github.com/gaborgulyas/constrainted_fingerprinting, 2016.

[197] Gábor György Gulyás, Gergely Acs, and Claude Castelluccia. Near-optimal fingerprinting with constraints. *Proceedings on Privacy Enhancing Technologies*, 2016(4):470–487, 2016.

[198] Gábor György Gulyás, Dolière Francis Somé, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. In *To appear in the Proceedings of the 2018 ACM on Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, Canada, October 15 - 19, 2018*, 2018.

[199] Jonas Haag. Modern and flexible browser fingerprinting library. https://github.com/Valve/fingerprintjs2.

[200] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. May I? - Content Security Policy Endorsement for Browser Extensions. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148 of *Lecture Notes in Computer Science*, pages 261–281. Springer, 2015.

[201] Brian Hayes. Uniquely me! how much information does it take to single out one person among billions? 102:106–109, 2014.

[202] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The most dangerous code in the browser. In George Candea, editor, *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[203] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. HTML5. A vocabulary and associated APIs for HTML and XHTML. W3C Recommendation, 2014. https://www.w3.org/TR/html5/embedded-content-0.html#an-iframe-srcdoc-document.

[204] Ariya Hidayat. ECMAScript Parsing Infrastructure. https://www.npmjs.com/package/esprima.

[205] Ariya Hidayat. PhantomJS Headless Browser, 2010-2016. http://www.phantomjs.org/.

[206] Egor Homakov. Using content-security-policy for evil. http://homakov.blogspot.fr/2014/01/using-content-security-policy-for-evil.html, 2014.

[207] Egor Homakov. Profilejacking - legal tricks to detect user profile. https://sakurity.com/blog/2015/03/10/Profilejacking.html, 2015.

[208] Collin Jackson and Adam Barth. Beware of Finer-Grained Origins. In *Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[209] Ashar Javed. CSP Aider: An Automated Recommendation of Content Security Policy for Web Applications. In *IEEE Oakland Web 2.0 Security and Privacy (W2SP'12)*, 2012.

[210] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 34–44. ACM, 2012.

[211] Martin Johns. Preparedjs: Secure script-templates for javascript. In Rieck et al. [241], pages 102–121.

[212] Martin Johns. Script-templates for the content security policy. *J. Inf. Sec. Appl.*, 19(3):209–223, 2014.

[213] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 641–654. USENIX Association, 2014.

[214] Christoph Kerschbaumer, Sid Stamm, and Stefan Brunthaler. Injecting CSP for Fun and Security. In Olivier Camp, Steven Furnell, and Paolo Mori, editors, *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016), Rome, Italy, February 19-21, 2016.*, pages 15–25. SciTePress, 2016.

[215] Engin Kirda and Thomas Ristenpart, editors. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017.

[216] Krzysztof Kotowitz. Intro to chrome addons hacking: fingerprinting. http://blog.kotowicz.net/2012/02/intro-to-chrome-addons-hacking.html, 2012.

[217] Balachander Krishnamurthy and Craig E. Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proc. of the 18th WWW*, pages 541–550, 2009.

[218] Pierre Laperdrix. *Browser Fingerprinting: Exploring Device Diversity to Augment Authentication and Build Client-Side Countermeasures. (Empreinte digitale d'appareil: exploration de la diversité des terminaux modernes pour renforcer l'authentification en ligne et construire descontremesures côté client)*. PhD thesis, INSA Rennes, France, 2017.

[219] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In Bodden et al. [174], pages 97–114.

[220] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Mitigating browser fingerprint tracking: Multi-level reconfiguration and diversification. In Paola Inverardi and Bradley R. Schmerl, editors, *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, pages 98–108. IEEE Computer Society, 2015.

[221] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 878–894. IEEE Computer Society, 2016.

[222] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *Proc. of the 25th USENIX Security*, 2016.

[223] Robin Linus. Your social media fingerprint. https://robinlinus.github.io/socialmedia-leak/, 2016.

[224] Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. Extensible web browser security. In Bernhard M. Hämmerli and Robin Sommer, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 4th International Conference, DIMVA 2007, Lucerne, Switzerland, July 12-13, 2007, Proceedings*, volume 4579 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2007.

[225] Jonathan R. Mayer and John C. Mitchell. Third-party web tracking: Policy and technology. In *Proc. of the 2012 IEEE SP*, pages 413–427, 2012.

[226] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *Proc. of the 2nd EuroSP*, Paris, France, 2017.

[227] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In Matt Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.

[228] Ben Newman. JavaScript Syntax Tree Transformer. https://www.npmjs.com/package/recast.

[229] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proc. of the 2012 CCS*, pages 736–747, 2012.

[230] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 541–555. IEEE Computer Society, 2013.

[231] Łukasz Olejnik, Claude Castelluccia, and Artur Janc. Why johnny can't browse in peace: On the uniqueness of web browsing history patterns. In *Hot Topics in Privacy Enhancing Technologies (HotPETs 2012)*, 07 2012.

[232] Kaan Onarlioglu, Mustafa Battal, William K. Robertson, and Engin Kirda. Securing legacy firefox extensions with SENTINEL. In Rieck et al. [241], pages 122–138.

[233] Kaan Onarlioglu, Ahmet Salih Buyukkayhan, William K. Robertson, and Engin Kirda. SENTINEL: securing legacy firefox extensions. *Computers & Security*, 49:147–161, 2015.

[234] Xiang Pan, Yinzhi Cao, and Yan Chen. I do not know what you visited last summer: Protecting users from stateful third-party web tracking with trackingfree browser. In *Proc. of the 22nd NDSS*, 2015.

[235] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In Weippl et al. [268], pages 653–665.

[236] Kailas Patil and Braun Frederik. A Measurement Study of the Content Security Policy on Real-World Applications. *I. J. Network Security*, 18(2):383–392, 2016.

[237] Ian Paul. Firefox will stop supporting plugins by end of 2016, following chrome's lead. https://www.pcworld.com/article/2990991/browsers/firefox-will-stop-supporting-npapi-plugins-by-end-of-2016-following-chromes-lead.html.

[238] Nicolas Perriault. CasperJS navigation and scripting tool for PhantomJS, 2011-2016. http://www.casperjs.org/.

[239] M. Pusara and C. Brodley. User re-authentication via mouse movements. In *ACM Workshop Visualizat. Data Mining Comput. Security*, page 1–8, 2004.

[240] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in javascript applications. In Mira

Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011.

[241] Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert, editors. *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings*, volume 7967 of *Lecture Notes in Computer Science*. Springer, 2013.

[242] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proc. of the 9th NSDI*, pages 155–168, 2012.

[243] Joseph Roth, Xiaoming Liu, and Dimitris Metaxas. On continuous user authentication via typing behavior. 23(10):4611–4624, 2014.

[244] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[245] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In Kirda and Ristenpart [215], pages 679–694.

[246] Justin Schuh. Canvas DefendeSaying Goodbye to Our Old Friend NPAPI, September 2013. https://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html.

[247] Manuel Serrano. Hop.js - Multi-tier JavaScript. http://hop.inria.fr/home/index.html.

[248] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the Incoherencies in Web Browser Access Control Policies. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 463–478, 2010.

[249] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 329–336. ACM, 2017.

[250] Ashkan Soltani, Shannon Canty, Quentin Mayo, Lauren Thomas, and Chris Jay Hoofnagle. Flash Cookies and Privacy. In *AAAI spring symposium: intelligent information privacy management*, pages 158–163, 2010.

[251] Dolière Francis Somé. Breaking the Same Origin Policy for free - On CORS headers manipulations by browser extensions. Submitted for review.

[252] Dolière Francis Somé. EmPoWeb: Empowering web applications with browser extensions. Submitted for review.

[253] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. On the Content Security Policy violations due to the Same-Origin Policy. Technical report. http://www-sop.inria.fr/members/Nataliia.Bielova/papers/CSP-SOP.pdf.

[254] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. Control what you include! - server-side protection against third party web tracking. In Bodden et al. [174], pages 115–132.

[255] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. On the content security policy violations due to the same-origin policy. In Barrett et al. [171], pages 877–886.

[256] Dolière Francis Somé and Tamara Rezk. DF-CSP: Dependency-Free Content Security Policy. Submitted for review.

[257] Dolière Francis Somé and Tamara Rezk. Extending Content Security Policy: Blacklisting, URL arguments filtering and Monitoring. Submitted for review.

[258] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 921–930. ACM, 2010.

[259] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In Barrett et al. [171], pages 1481–1490.

[260] Oleksii Starov and Nick Nikiforakis. XHOUND: quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 941–956. IEEE Computer Society, 2017.

[261] Brandon Sterne and Adam Barth. Content Security Policy 1.0. W3C Candidate Recommendation, 2012. http://www.w3.org/TR/2012/CR-CSP-20121115/.

[262] Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. Gradual typing embedded securely in JavaScript. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 425–438. ACM, 2014.

[263] Naoki Takei, Takamichi Saito, Ko Takasu, and Tomotaka Yamada. Web browser fingerprinting using only cascading style sheets. In *Proc. of the 10th BWCCA*, pages 57–63, 2015.

[264] Randika Upathilake, Yingkun Li, and Ashraf Matrawy. A classification of web browser fingerprinting techniques. In *Proc. of the 7th NTMS*, pages 1–5, 2015.

[265] Anne van Kesteren. Cross Origin Resource Sharing. W3C Recommendation, 2014. https://www.w3.org/TR/cors/.

[266] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. FP-STALKER: tracking browser fingerprint evolutions. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 728–741. IEEE, 2018.

[267] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In Weippl et al. [268], pages 1376–1387.

[268] Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016.

[269] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 212–233, 2014.

[270] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William K. Robertson, and Engin Kirda. Ex-ray: Detection of history-leaking

browser extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*, pages 590–602. ACM, 2017.

[271] Mike West. Content Security Policy: Embedded Enforcement, 2016. https://w3c.github.io/webappsec-csp/embedded/.

[272] Mike West. Content Security Policy Level 3. W3C Working Draft, 2016. http://www.w3.org/TR/CSP3/.

[273] Mike West. Mixed Content, 2016. https://www.w3.org/TR/mixed-content/.

[274] Mike West. Origin Policy. A Collection of Interesting Ideas, 2016. https://wicg.github.io/origin-policy/.

[275] Mike West, Adam Barth, and Dan Veditz. Content Security Policy Level 2. W3C Candidate Recommendation, 2015. http://www.w3.org/TR/CSP2/.

[276] Mike West and Ilya Grigorik. Feature Policy. W3C Draft Community Group Report, 2016. https://wicg.github.io/feature-policy/.

[277] Rob Wu. CRX Extension Source Viewer For Chrome, Opera, and Firefox. https://robwu.nl/crxviewer/.

[278] Imran Yusof and Al-Sakib Khan Pathan. Mitigating Cross-Site Scripting Attacks with a Content Security Policy. *IEEE Computer*, 49(3):56–63, 2016.

[279] Yu Zhong, Yunbin Deng, and Anil K. Jain. Keystroke dynamics for user authentication. In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, Providence, RI, USA, June 16-21, 2012*, pages 117–123, 2012.