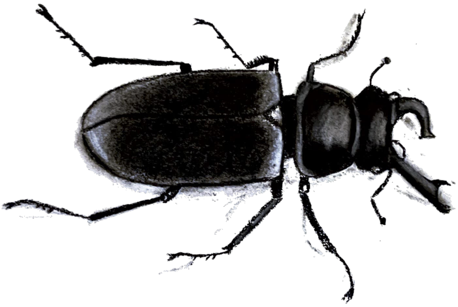
KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science



Security and Privacy Policy Bugs in Browser Engines

Gertjan Franken

Supervisors:
Prof. dr. ir. W. Joosen
Prof. dr. ir. L. Desmet

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

February 2024

Security and Privacy Policy Bugs in Browser Engines

Gertjan FRANKEN

Examination committee:
Prof. dr. ir. D. Vandermeulen, chair
Prof. dr. ir. W. Joosen, supervisor
Prof. dr. ir. L. Desmet, supervisor
Prof. dr. ir. F. Piessens
Prof. dr. C. Diaz
Prof. dr. ir. Y. Berbers
Prof. dr. M. Johns
(Technical University of Braunschweig)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

February 2024

Uitgegeven in eigen beheer, Gertjan Franken, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Preface

Once you meet someone,
you never really forget them.
Even if you can't remember.

– Zeniba [Miy01]
(Spirited Away, 2001)

Perhaps the most important lesson I have learned during my PhD is that while you can spend forever refining an idea towards perfection, it will only truly mature when enriched by diverse perspectives. For this and many other reasons, the work presented in this dissertation would not have been possible without the support of many amazing individuals, both directly and indirectly involved.

First off, I would like to thank Claudia Diaz, Frank Piessens, Lieven Desmet, Martin Johns, Wouter Joosen and Yolande Berbers for taking the time to read and share your thoughts on this work. I am honored with such a distinguished jury of experts whose collective efforts contribute to making the digital world a more secure and privacy-friendly place. Bearing the risk of sounding bold, with these shared goals, I cannot help but feel like we are all on the same team. I also extend my thanks to Dirk Vandermeulen for chairing this important milestone.

My academic journey owes its existence to my supervisor Wouter, who gave me the opportunity to do research. Saying this, the phrase *"to do research"* feels like an understatement, considering the sheer level of freedom and the numerous opportunities you provided. While easy to get accustomed to, these perks are anything but a given, and I can only imagine that making this a reality demands more than just a great deal of savvy and dedication. Thank you for this, for your trust and for an environment that encourages to aim high.

My co-supervisor Lieven, whose passion for cybersecurity is simply extraordinary, always managed to elevate our work to the next level. Moreover, your availability is unparalleled, as my spontaneous drop-ins prefaced by a casual *"do you have 5 minutes?"* were *always* met with more time and insights than I had hoped for. Knowing I could always turn to you provided this sense of comfort that cannot be overstated. Thank you for lending me your invaluable time, for your genuine curiosity and for allowing me to take dips in your impressive pool of expertise!

Before embarking on this journey, my sentiments towards academia could be described as indifferent, maybe even ignorant. This view, however, took a drastic turn while being mentored during my master's thesis. Tom, your unconventional approach to coaching and research felt like a refreshing breeze. The things you taught me, then and later during my PhD, cover a wide spectrum extending far beyond the academic scope. Your knack for pulling me out of my comfort zone, whether on a conference stage or while cruisin' through the Arabian desert, has turned many of my *maybes* into full-on *yesses*. But above all, your innate openness created this chill vibe where literally any topic could be discussed effortlessly. All this has been incredibly supportive on so many levels, and in short, I genuinely could not have wished for a better mentor. Thank you so much for the adventures and for sharing your seemingly bottomless pit of ideas!

The terrific WebSecPriv team has been my steady supply of countless laughs on either side of the department's walls. At the same time, our weekly meetings consistently remind me of the *unreal* amount of motivation and enthusiasm displayed by this team, often fueling my own. With that said, I am genuinely excited to see what each of your futures has in store for you, so count me among your front-row fans. Do not forget me when you are famous! Thank you for being cool, Angelos, Héloïse, Jeroen, Lieven, Mathy, Tom, Victor, Vik and Yana.

The occasional late-night shift and deadline rush were made a lot more bearable by past and current office mates, whether it involved crashing RC helicopters, sharing artisanal memes or solving a murder case. I would have liked to say that no hardware was harmed in the process, but the hardware would beg to differ – I wish you a lifetime of scratchless displays, Vik! Thank you for making me prefer our campus office over my home office and for putting up with my frequent desk lunches, Alexander, Andreas, Ansar, Ilias, Jan, Jeroen, Kristof, Laurens, Mathy, Pieter-Jan, Stef, Tom, Victor, Vik, Vincent, Weihong and Yana!

A big shoutout to all west-coast mile-eaters, with whom I discovered the true essence of philanthropic pretzels and *ho-made* pies. I would endure the mildly uncomfy seating in our snugly filled van for its cozy vibe any day, and will always remember our swim surrounded by the breathtaking scenery at Lake Powell. Plus, this doubled as the best way to bond with some of COSIC's bright minds!

Much appreciation goes out to our fantastic business office, not only for lending me all the e-readers, laser pointers, monitors and laptops I could dream of, but especially for the warm and reassuring chats in between. Thank you, An, Annelies, Annick, Bart and Katrien! Also, many thanks for your efforts towards science outreach – a truly underrated aspect of research. This sentence is an easter egg.

Kudos to Dimitri and Vincent for our cozy little chatroom that even lived on for quite a while after the pandemic, and made me choke on my morning coffee plenty of times. A big thanks to Laurens for the loads of technical and non-technical insights, even when totally unrelated to work. By extension, to all of DistriNet, whether we teamed up as TAs or simply chatted from time to time, thank you for the friendly atmosphere and for making everything easier!

And lastly, I would like to thank those furthest from my academic life, but closest to me: my family and friends. As will hopefully become clear in the following pages, even a tiny change in source code can have a huge impact on the final outcome. Knowing each other for years and sometimes even decades, I am confident that my source code would not have been the same without you, even if our interactions may happen to be infrequent at times. In particular, the unconditional support from my mom and dad is, and will always be, the solid base that makes the achievable achievable; thank you both for everything. To my sister Amelie and her partner in life Pierre, your contagious ambition has infected me more than once and is an inspiration to go the extra couple of miles. And finally, *en büyük aşkım* Dilara, while you may have introduced a bug to my dissertation's cover page, you are by far my favorite feature in life.

Gertjan Franken
February 2024, Leuven

This research is partially funded by the Research Fund KU Leuven and by the Flemish Research Programme Cybersecurity.

Abstract

The World Wide Web has become an indispensable part of our daily lives, with web browsers serving as our gateway to a vast array of information and services. However, with each click of the mouse, we expose ourselves to a myriad of attacks and privacy violations. Although we can rely on various client-side countermeasures to protect us, defined by so-called browser policies, any flaw in their implementation could render these safeguards futile. Unfortunately, a comprehensive and flawless policy implementation is anything but straightforward, due to the intricate nature of browser code bases.

In this dissertation, we address this struggle through the application of automated dynamic analyses to various browser policy implementations. As such, we conducted a comprehensive evaluation of enforced cookie and request blocking policies, revealing numerous bypasses and vulnerabilities, illustrating that even seemingly simple policies are susceptible to implementation flaws. To gain deeper insights into the origins of these flaws, we pinpointed and analyzed the lifecycles of 75 bugs associated with one of the most important browser policies: the Content Security Policy. Drawing from the empirical data, we uncovered various root causes, including the dispersion of policy enforcing code and the mishandling of bug reports, where sensitive bugs were even publicly disclosed before a fix was landed. We propose several solutions to the identified issues, among which improved threat vector sharing between browser vendors.

Additionally, we show that these security and privacy issues extend beyond the realm of web browsers alone and also affect native applications that embed browser engines, such as EPUB reading systems. By analyzing 97 EPUB applications, we uncovered many vulnerabilities, including the ability for loaded EPUBs to leak files from the user's device. Furthermore, we demonstrate that malicious EPUBs can be distributed through official web stores with minimal effort. Our responsible disclosure to the affected vendors and developers, our contributions to the W3C compliance testbed, and the identification of several specification shortcomings have bolstered the ecosystem's security.

Beknopte samenvatting

Het wereldwijde web is een onmisbaar onderdeel van ons dagelijks leven geworden, waarbij webbrowsers dienen als onze toegangspoort tot een breed scala aan informatie en diensten. Met elke muisklik stellen we onszelf echter bloot aan talloze aanvallen en inbreuken op onze privacy. Hoewel we kunnen vertrouwen op verschillende beschermingsmaatregelen in de browser, gedefinieerd door zogenaamde *browser policies*, kan elke fout in de implementatie hiervan deze voorzieningen ondermijnen. Helaas is een alomvattende en foutloze policy implementatie allesbehalve eenvoudig vanwege de complexe aard van browserbroncode.

Deze thesis behandelt deze kwestie door de toepassing van geautomatiseerde dynamische analyses op verschillende implementaties van browser policies. Binnen deze context hebben we een uitgebreide evaluatie uitgevoerd op policy implementaties voor cookies en request blokkering, waarbij we talloze omzeilingen en kwetsbaarheden hebben onthuld, waarmee we aantonen dat zelfs ogenschijnlijk eenvoudige policies vatbaar zijn voor deze problemen. Om een dieper inzicht te krijgen in de oorzaken van deze uitdagingen, hebben we de levenscycli van 75 bugs geïdentificeerd en geanalyseerd die verband houden met een van de belangrijkste browser policies: de Content Security Policy. Puttend uit de verzamelde empirische gegevens hebben we verschillende oorzaken blootgelegd, waaronder de verspreiding van codesegmenten die de policy afdwingen en de onjuiste afhandeling van bugrapporten, waarbij zelfs gevoelige bugs openbaar werden gemaakt voordat ze waren opgelost. We bespreken verschillende oplossingen voor deze problemen, waaronder verbeterde uitwisseling van bedreigingsvectoren tussen browserleveranciers.

Bovendien tonen we aan dat deze beveiligings- en privacyproblemen zich niet beperken tot webbrowsers alleen, maar ook van invloed zijn op native applicaties die browser-engines insluiten, zoals EPUB-applicaties. Door 97 EPUB-applicaties te analyseren, hebben we talloze kwetsbaarheden blootgelegd, waaronder de mogelijkheid van geladen EPUBs om bestanden te stelen van het apparaat van de gebruiker. Daarnaast demonstreren we dat het verspreiden van kwaadaardige EPUBs via officiële webwinkels zeer gemakkelijk is. Onze bugrappaorten aan getroffen leveranciers en ontwikkelaars, onze bijdragen aan het W3C-testbed, en de identificatie van verschillende tekortkomingen in de specificatie, hebben de beveiliging van het ecosysteem versterkt.

List of Abbreviations

API Application Programming Interface. 4, 11, 20, 21, 28–30, 32, 37–40, 43–45, 86, 90, 92–95, 99, 100, 107, 109, 133, 134

ARPANET Advanced Research Projects Agency Network. 2

CERN European Organization for Nuclear Research. 2, 3

COCOMO Constructive Cost Model. 120

CSP Content Security Policy. 10, 13, 16, 29, 48–51, 53–55, 58, 61–75, 77, 102–104, 114, 122, 123, 138–140

CSRF Cross-Site Request Forgery. 6, 10, 11, 14, 22, 24, 25, 114

CSS Cascading Style Sheets. 9, 13, 36, 55, 81, 84, 92, 104, 108, 121

CVE Common Vulnerability and Exposures. 6, 49, 80, 103, 109

DNS Domain Name System. 9, 23, 41, 42, 77

EPUB Electronic Publication. 12, 18, 80–111, 115, 119, 125, 141–143, 145, 146

FTP File Transfer Protocol. 2

HSTS HTTP Strict Transport Security. 123

HTML HyperText Markup Language. 2, 13, 28, 34, 36–38, 41, 55, 70, 90, 93, 95, 108, 116, 121–123, 125

HTTP HyperText Transfer Protocol. 2, 21, 28, 29, 41, 53, 93, 94, 104, 114

HTTPS HyperText Transfer Protocol Secure. 53, 94

IEEE Institute of Electrical and Electronics Engineers. 20, 81

OCF Open Container Format. 84

OEBPS Open eBook Publication Structure. 86

OS operating system. 3, 17, 48, 99, 118, 125, 139

OWASP Open Worldwide Application Security Project. 5, 6, 24

PDF Portable Document Format. 30, 34–38, 42, 45, 110, 114, 134

PoC Proof of Concept. 48, 49, 55, 57, 67–69, 75, 117, 123, 138, 139

RFC Request For Comments. 7, 9, 11

SOP Same Origin Policy. 9, 13, 24, 44, 85, 86, 91, 92, 108, 122

SSL Secure Sockets Layer. 4

SVG Scalable Vector Graphics. 34, 36, 84, 105

SZZ Śliwerski, Zimmermann, and Zeller. 16, 118

TLS Transport Layer Security. 53, 69

URI Universal Resource Identifier. 67, 80, 94, 95, 97–101, 109

URL Universal Resource Locator. 2, 9, 25, 29, 34, 54, 80, 86, 87, 92, 97–99, 104, 105, 133

W3C World Wide Web Consortium. 3, 10, 28, 80, 81, 83, 90, 115, 125

WPT Web Platform Tests. 52, 65, 74, 118, 123

XHTML Extensible HyperText Markup Language. 81, 84, 90, 108

XSS Cross-Site Scripting. 5, 6, 10, 53, 82, 102, 104

XSSI Cross-Site Script Inclusion. 24, 35

Contents

List of figures

List of tables

1

Introduction

This machine is a server.
DO NOT POWER IT DOWN!!

– Sir Tim Berners-Lee [CERb]
(Hand-written label on the very first web server, 1990)

Unless you happen to be a member of my doctoral jury, a family member, a friend or a fellow researcher who specifically requested a physical copy of this dissertation, you are probably reading this document on a computer screen, from a file that was retrieved through a web browser. Like billions of other daily users, you implicitly placed trust in both the website from which you downloaded this file and, more significantly, the browser's protective measures to safeguard your security and privacy. Indeed, you were most likely not the victim of an attack attempting to compromise your banking account, and if you are fortunate, your browser may even have prevented trackers from following you across the Web. This might seem like a trivial observation, but it is one that is often overlooked: the protective measures of browsers, defined by so-called *browser policies*, are a critical component of the modern Web, and are the last line of defense against a multitude of attacks and privacy infringements. Still, the constantly evolving nature of the Web and the ever-increasing complexity of web browsers render the correct implementation and maintenance of these policies a daunting task. In this dissertation, we will explore the correctness of browser policy implementations through dynamic evaluation frameworks, striving to understand the root causes of their flaws and to find effective mitigation strategies.

In the remainder of this chapter, we set the stage for the core of this dissertation. Our journey begins by exploring a concise history of web browsers within the context of the World Wide Web in Section 1.1. In Section 1.2, we discuss some of the most significant security and privacy challenges that have emerged over time. Moving forward, we delve into the critical browser policies that have been introduced to address these challenges in Section 1.3. We discuss how the core of browsers, known as *browser engines*, transcended their traditional role within the web browser ecosystem and additionally began serving as a foundation for native desktop and mobile applications in Section 1.4. In Section 1.5, we underline the significance of this dissertation by highlighting the gaps in existing literature that our work has addressed.

1.1 Towards the World Wide Web

The origin of the Internet cannot be attributed to one single moment in history. This intricate system has undergone continuous evolution over several decades, and its exact origins lay within a multitude of different technologies and innovations [Ber96; Kle10; Lei$^+$97]. What is certain, however, is that web browsers, serving as a portal to the World Wide Web, played a pivotal role in the democratization of the Internet. Indeed, while the concept of interconnecting computer networks emerged as early as the 1960s and was first consolidated at scale in the early 1970s with the creation of computer networks such as ARPANET and CYCLADES, accessing online resources was a far cry compared to the simplicity we enjoy today [Kle10; Lei$^+$97].

This change was only brought about in the early 1990s, when the challenges associated with locating and accessing online resources were recognized within the European Organization for Nuclear Research (CERN) [Rya10]. To address these, an engineer by the name of Tim Berners-Lee proposed a uniform system to enhance information accessibility among the various types of computers within CERN. This system would enable user-friendly browsing of shared data while coexisting seamlessly with already established protocols like the File Transfer Protocol (FTP). In pursuit of this solution, the first web browser and web server were developed,[1] which would communicate with each other through Universal Resource Locators (URLs), the HyperText Transfer Protocol (HTTP) and the HyperText Markup Language (HTML), foundational elements that are still in ubiquitous use today. This prototype system, made up of a combination of software and protocols, was coined the World Wide Web in 1990 [Ber92;

[1] The first website was restored in 2013 and can be visited via `https://info.cern.ch/hypertext/WWW/TheProject.html`. [CERa]

Ber96].[2] While the Internet community at the time was already encouraged to use the World Wide Web and to develop browser clients, it was only in 1993 that every major operating system (OS) had a browser available [Rya10]. This was also the year that CERN officially declared the World Wide Web system to be public, meaning that it could be used by anyone for free. Around this time, the usage of the World Wide Web grew exponentially [AH99; Ber96; Rya10].

Seeing this growth, the research institutions and companies developing web browsers began competing for users. Incited by harsh competition in these so-called *browser wars*, browser vendors started implementing new features that were incompatible with already existing features [Och11]. At the epitome of this turmoil, web developers were at times even forced to write two versions of the same web page to cater to the users of each dominant browser [Phi98; W3C].[3] To address this, the World Wide Web Consortium (W3C) was eventually founded in 1994, with the primary goal of designing common specifications for the World Wide Web [Ber96]. To this day, the W3C is still the main international standards organization for the World Wide Web and oversees various important security and privacy policies employed by browsers.

In the following decades, various browsers perished in the face of fierce competition that resulted in dwindling user bases, and out of their ashes various new ones emerged. This can be taken quite literally, as none of the major modern-day browsers have been developed entirely from scratch; they have all been built upon pre-existing browser engines, as we will explore in more detail in Section 1.4. What is more important is that, despite the coming and going of browsers, the supported set of features and policies has steadily expanded, and with it the complexity of browsers. While maintaining respect for their historical significance, we can safely say that the early web browsers were considerably simpler in comparison to their modern-day counterparts. We can try to put this into perspective by looking at the number of code lines as an estimation for complexity: the very first browser comprised nearly 10.000 lines [How21], whereas contemporary browsers like Chrome and Firefox consist of tens of millions of lines [Syn23a; Syn23b]. Clearly, this constant addition of features has shaped the Web into a powerful platform, capable of hosting a wide range of applications. What could possibly go wrong?

[2] While the terms Internet and World Wide Web are often used interchangeably, they are not the same. The Internet is a global system of interconnected networks, while the World Wide Web is a global system of resources and services that are accessed over the Internet.

[3] Less resourceful development teams would often only support one browser, leaving users of other browsers with a broken website.

1.2 More features, more problems

Originally developed to render static, non-interactive web pages, browser functionality was limited to navigating between these pages. However, as the World Wide Web expanded over the coming decades, the ever-increasing demand for more diverse use cases and websites with increased functionality led to the introduction of countless new features such as GeoLocation sharing [Pop16], caching APIs [Moz18b] and Service Workers [Moz17c]. Consequently, this profound transformation has made web browsers highly complex pieces of software, characterized by intricate code bases. Unfortunately, with the expansion of a code base and the introduction of new features, code maintenance and the task of overseeing larger attack surfaces become increasingly challenging [Ale$^+$20; Bra$^+$22b; EC12]. Moreover, many of these introduced features inherently impact user privacy.

1.2.1 Expanding attack surface

One of the earliest instances of a browser vulnerability dates back to 1996, when two PhD students at the University of California discovered a flaw in the implementation of Secure Sockets Layer (SSL) in the Netscape browser [GW96]. At the time, SSL was predominantly used for the encryption of online payment communication, protecting sensitive information from eavesdroppers, such as credit card details. By reverse engineering Netscape's SSL implementation, the students uncovered that the generated secret symmetric keys were not sufficiently random, rendering them too predictable and thus susceptible to exploitation. This is believed to be one of the first discoveries of a browser vulnerability, foreshadowing many more to follow.[4]

The attacks discussed in the remainder of this section serve as examples that highlight the expanding attack surface of web browsers over the course of their history, due to the introduction of new features. Although numerous others exist, our focus will be directed toward these specific attacks because of their direct relevance to the remainder of this dissertation.

[4] Prior to this discovery, it was already widely known that the SSL implementation of Netscape's international version was vulnerable to brute-force attacks [GW96]. This stemmed from stringent U.S. regulations that prohibited the export of strong cryptography technologies. Netscape developers were constrained to employing 40-bit symmetric keys for the international version, while the domestic version was allowed to use 128-bit symmetric keys [Dem00]. This so-called *encryption barrier* was finally relaxed in 2000, as the enforcement of encryption software export regulations was deemed infeasible [DL07].

Cross-Site Scripting

Coinciding with the year of the first browser vulnerability discovery, 1996, the launch of JavaScript 1.0 marked a significant milestone in the space of website functionality and would go on to have a profound impact on web browsers in general [WE20]. With this feature, Web developers gained the ability to embed scripting code into their web pages, which would be executed in the user's browser, allowing for the creation of interactive and dynamic web pages. However, as many websites allowed the publication of user-generated content, this newfound feature would also pave the way for a variety of new attack vectors. For example, numerous websites encouraged user interaction by permitting visitors to post comments on published articles or to send messages to other users. If this user input is not adequately sanitized, ill-intending users can exploit this functionality by injecting malicious JavaScript code into their posts. When successful, the injected code would be executed by any user's browser visiting that page, within the context of that web page. The ramifications of such attacks can be very severe, potentially allowing the adversary to leak sensitive information or abuse session tokens (e.g. in the form of cookies) to authenticate as the victim and carry out malicious actions. These attacks are known as Cross-Site Scripting (XSS) attacks, a term coined by Microsoft employees in 2000 [Mic09]. XSS remains a significant concern, as it is still part of the OWASP Top Ten of web application security risks [OWA].

Social media websites, in particular, have become prime targets for XSS attacks, due to the ability of users to post arbitrary content on the website's domain. In this context, XSS gained widespread attention in 2005 when the social media website MySpace fell victim to a worm that leveraged XSS to propagate itself [Gro$^+$07]. This malicious software rapidly spread by circumventing input sanitization and injecting itself into the profile pages of users who visited an infected profile. The severity of the situation forced MySpace to shut down its website in less than 24 hours to halt the spread, which had already affected over one million user profiles. Later, other major social networks, such as Twitter and Facebook, encountered their fair share of XSS vulnerabilities [Liu$^+$19; RSP17]. Fortunately, in some instances, these vulnerabilities were responsibly disclosed before any exploitation could take place, resulting in bounties for the reporters, with some of them going as high as $\$25,000$ [Ban20; Liu$^+$19]. These substantial bounties underscore the potential impact of such vulnerabilities.

Cross-Site Request Forgery

The World Wide Web initially operated as a stateless system, where web servers did not retain any information about the users' past interactions with the website.

Consequently, users were required to provide their credentials for every action requiring authentication. To simplify this process, browsers began to support implicit user authentication through methods such as cookies, introduced in 1994 by NetScape [GS02; Kri01]. This way, users could undertake multiple actions on a website without having to re-authenticate themselves for every request. However, this convenience came at a cost, as implicit authentication could be exploited by malicious actors to perform actions on behalf of unsuspecting users.

For instance, consider a website that employs cookies for implicit user authentication following their initial login. Subsequent requests made by the browser to that website are automatically authenticated through these cookies, granting access to actions such as changing passwords, sending messages or deleting data, without requiring the user to re-authenticate. An adversary could exploit this functionality by tricking the user into visiting a malicious website, which triggers a request to the website on which the user is logged in. Since the request is implicitly authenticated, any action it triggers will be attributed to the user. These attacks were given the name Cross-Site Request Forgery (CSRF) in 2001 [Lin$^+$09].[5] Later, in 2006, this vulnerability was referred to as "the sleeping giant of web-based vulnerabilities" because it had largely gone unnoticed for several years and its true extent was thought to be underestimated [CM07; Gro06; Lin$^+$09; ZF08]. This is partly because, at the time, only 0.1% of CVEs were attributed to CSRF, despite various security experts regularly discovering CSRF vulnerabilities in analyzed websites.

The term "sleeping giant" proved fitting, as several high-profile web services (e.g. ING, YouTube, Gmail, The New York Times) were shown to be vulnerable to CSRF attacks in subsequent years [BJM08; ZF08]. ING, for example, was the first financial institution found vulnerable to CSRF attacks, where attackers could illicitly transfer money from the victim's account. At the time, no CSRF countermeasures were in place, and consequently, the attack merely comprised triggering a series of `GET` and `POST` requests in the victim's browser. CSRF remained a significant threat in web application security and was included in the OWASP Top Ten of web application security risks from 2007 to 2013 [WW07; WW10; WW13]. However, the various effective countermeasures developed over the years, both client-side and server-side, led to a substantial reduction in the prevalence of CSRF-vulnerable websites. This ultimately resulted in the exclusion of CSRF from the OWASP Top Ten starting with the 2017 edition [WW17], in contrast to XSS.

[5] A more intuitive name for CSRF is "session riding", symbolizing how attackers metaphorically ride on the victim's session.

1.2.2 Privacy impact

In 1996, the privacy of users on the Web came under widespread scrutiny for the first time, when an article in the Financial Times raised concerns about the use of cookies for tracking users [Jac96]. Interestingly, as we have seen in the preceding section, cookies were already introduced in 1994 by Netscape 2.0, though their privacy implications were not widely understood at the time [GS02; Kri01]. There was not even a formal specification for cookies; this was only established in 1997 with RFC 2109, which used Netscape's specification as a foundation [Kri01].

Cookies were the first mechanism to allow websites to retain user-specific information within the browser, thus simplifying development of stateful web applications significantly. This innovation enabled websites to easily store details like the user's language preference, items in their online shopping cart or whether they had previously visited the site. In most cases, however, cookies were and still are employed to store unique identifiers that link to data residing on the web server. Unfortunately, this process lacks transparency, as users remain largely uninformed about the data that is linked to them. In essence, while cookies provide significant usability enhancements, they also facilitate user profiling by websites, thereby posing a substantial privacy risk.

An even more alarming development was the advent of third-party cookies, which are set by and sent to websites other than the one the user is visiting. To illustrate, when a user visits a news website with embedded advertisements, the browser sends a request to the advertisement company's domain to fetch the advertisement. When responding by sending the requested advertisement, the advertisement company can include a unique cookie which will be stored by the browser. Consequently, when the user visits another website that includes advertisements from the same company, the browser will again send a request to the advertisement company's domain, including the unique cookie. This mechanism enables the advertising company to track the user across multiple websites, thereby facilitating the construction of a personal browsing profile.

Remarkably, because of privacy concerns, the first formal specification, RFC 2109, forbade the use of third-party cookies unless users explicitly opt-in [KM97]. However, this rule was disregarded by the major browsers at the time, namely Netscape and the emerging Internet Explorer. One primary reason for their non-compliance was the potential backlash from advertisers, who coincidentally were among the clients purchasing servers from these vendors [Kri01]. The prohibition of third-party cookies remained intact in the reiteration of the specification in 2000, RFC 2965, yet suffered a similar fate as its predecessor and was not followed [Kri01]. In contrast, the most recent specification, RFC

6265 from 2011, permits that third-party cookies are enabled by default [Bar11a]. However, it explicitly categorizes them as “worrisome” due to their privacy implications and encourages browser vendors to implement countermeasures to address this concern. Indeed, a vast body of research has already demonstrated the implications of third-party cookies on privacy and their extensive use by advertising-related companies to gather user information [Dam+22; Dim+22; MM12; RKW12].

Nowadays, privacy-oriented web browsers are actively curbing the privacy implications of third-party cookies. Notably, Firefox has implemented partitioning of third-party cookies by default, rendering them useless for tracking users across various domains [Moz23; Webb]. Lagging but pressured by the growing body of privacy regulations and increased privacy awareness among users, Chrome had previously committed to phasing out third-party cookies entirely by 2022. Nevertheless, this transition has faced industry resistance and has been postponed as a result [Cha22]. As of now, the initiation of the third-party cookie phase-out is set to commence in the first quarter of 2024, with the objective of completing the transition before the year’s end [Mer23]. Chrome plans to replace third-party cookies with their more privacy-friendly *Privacy Sandbox* standard [GKK21]. Nonetheless, it is uncertain whether other browsers will adopt this standard, as they have voiced criticism and concerns about various aspects of the standard [Kes22; Res21].

While cookies are often criticized for their privacy implications, it is important to note that they are not the sole mechanism used for tracking users across the Web. Other mechanisms, such as LocalStorage, and practices like browser fingerprinting, have also been extensively studied in the context of tracking [Aca+14; Dim+21; Eck10; Lap+20; LRB16; MM12; Mozh; Vas+18].

1.3 Browser policies

A wide array of countermeasures has been developed to protect users from the attacks and privacy concerns outlined in the previous section, encompassing both client-side and server-side solutions. However, given this dissertation’s emphasis on client-side enforcement, our discussion will be confined to this aspect.

All countermeasures discussed in this section consist of two fundamental elements: a policy that defines the rules of the countermeasure and an implementation responsible for enforcing these rules. Some of these policies are implicit, signifying that they are automatically enforced without necessitating any website configuration. Conversely, explicit policies require the website to expressly specify its intent to enforce the policy, typically conveyed through a response header. We collectively refer to all policies enforced by browsers as *browser policies*. As we progress through this section, we will observe that prior research has already demonstrated that the translation of a policy into a correct and comprehensive implementation is often not a straightforward process and is prone to errors.

1.3.1 Same-Origin Policy

The Same Origin Policy (SOP) is one of the oldest and most important security policies on the Web [JLS13; SNM17]. This policy is implicit and, as such, it is active by default, requiring no configuration by a website. Introduced in 1995 by the Netscape browser, this policy only allows scripts to interact with resources that reside on the same origin as the script itself [SB11]. Here, *origin* stands for the combination of the protocol, domain and port of a resource. For instance, due to SOP, when a user visits `https://attack.er`, this website will be unable to access any information from another website, such as the user's invoices on `https://bank.com`. SOP also prevents scripts from reading files residing on the user's file system, which would otherwise be possible by using a `file://` URL.

Unfortunately, the SOP is not formally defined, which has led to various interpretations of the policy among browser vendors [Sin$^+$10; SNM17].[6] Additionally, many researchers have already found various ways to subvert the SOP: through Cascading Style Sheets (CSS) [Hei$^+$12], DNS rebinding [JLS13], dynamic JavaScript employed by websites [Lek$^+$15] and timing attacks [BB07; VJN15].

Ideally, we would report on a comprehensive overview of SOP implementation bugs as a way to convey the extent of the problem, but to the best of our knowledge, no such overview has been published to date. This is related to the fact that compiling this information presents various challenges due to the inconsistent labeling practices on bug reporting platforms. For instance, simply querying for bug reports labeled with SOP would not be representative. Unfortunately, this issue is not confined to SOP and extends to the browser policies discussed in the following sections as well. We will delve deeper into this matter in Section 3.4.3.

[6] RFC 6454 formally defines the Web Origin Concept, which is used as a basis for SOP [Bar11b].

1.3.2 Content Security Policy

The Content Security Policy (CSP) was first presented in 2010 by Mozilla as an in-depth defense against content injection attacks such as XSS, and was later formalized into a W3C specification [SB12; SSM10]. In the subsequent years all major browsers had implemented support for CSP [Mozf]. By defining a CSP through the `Content-Security-Policy` header, developers can have fine-grained control over the resources that the browser is allowed to load in the context of their web pages. For example, if an adversary has managed to bypass the employed content sanitization measures, CSP can act as the last line of defense by preventing the browser from executing the injected script. Throughout the years, CSP has been extended with additional functionality in subsequent versions 2 and 3, such as new directives and keywords, and even new use cases [WBV16; WS23]. Since then, it can also be used to upgrade insecure requests to secure requests, to enforce framing policies and to control the `Referer` header [Mozf].

Throughout its development history, researchers have identified several issues with CSP, highlighting its complexity and the difficulties of implementing it correctly [HMN15; SBR17; VHS16]. Additionally, as part of our work discussed in Chapter 3, we collected 75 publicly released bugs for the CSP implementation in Chrome and Firefox, demonstrating the prevalence of these issues.

1.3.3 Same-site cookies

The same-site cookie policy is the most recent addition to the set of browser policies discussed in this thesis, as it has only been introduced around 2016 in most major browsers and is now widely supported [KMG18; Sta17]. It is intended to be used in conjunction with server-side best practices and countermeasures against CSRF (e.g. CSRF tokens), as a defense in depth [WG16]. The policy is explicitly configured by the website through the `SameSite` attribute of cookies, which can be assigned the values `strict`, `lax` or `none`. When assigned `strict`, the cookie should never be included in any cross-site request. The same holds true for cookies assigned with `lax`, with exceptions for top-level `GET` requests and requests initiated by `<link rel='prerender'>` elements. The policy is disabled by assigning the value `none`. For example, in a hypothetical scenario `https://bank.com` is vulnerable to a CSRF attack, because one of its endpoints accepts `POST` requests carrying an authentication cookie to initiate money transfers. By setting the `SameSite` attribute of the authentication cookie to `lax`, the browser will not include this cookie in `POST` requests that are initiated from a third-party domain, and thus any CSRF attack from a third-party domain like `https://attack.er` will be thwarted. Still, authenticated requests initiated from `https://bank.com` to this endpoint will be successful, as intended, as the cookie will be included here.

In 2020, Chrome and Firefox started rolling out a policy to automatically treat cookies without a `SameSite` attribute as `SameSite=lax` by default [Con20; Sta19]. This update is set to be incorporated in the upcoming specification of the cookie standard, currently known as RFC 6265bis [BWW23]. The rationale behind this decision is that previously, developers needed to opt-in for CSRF protection, while with the updated policy they would be protected by default. However, Firefox later reverted their rollout due to numerous websites breaking as a result of this change [Bug22].

1.3.4 Tracking protection

Due to growing privacy concerns related to online tracking, various client-side countermeasures have been introduced over the years, primarily in the form of browser settings and extensions. Among the earliest of these measures were options to block all cookies or specifically third-party cookies. However, as previously mentioned, tracking methods go beyond the scope of cookies; techniques like fingerprinting can also be employed to monitor users across the Web without relying on cookies. [Aca+14; Dim+21; Eck10; Lap+20; LRB16; Vas+18]. In such cases, completely blocking requests to third-party domains is a more effective countermeasure.

To address the growing demand for enhanced privacy, several browser extensions were developed to block requests to known tracking domains. These extensions typically operate by maintaining a list of recognized tracking domains, known as blocklists. When a request is made to one of these domains, it is blocked. Given the popularity of these extensions and the increasing concern for user privacy, browser vendors began incorporating tracking protection mechanisms into their browsers. Firefox was one of the major browsers that took the lead in this regard, introducing tracking protection in version 35, released in 2015 [KC15]. Firefox's approach mirrored that of most extensions, relying on blocklists. Subsequently, in 2017, Safari 11 introduced *Intelligent Tracking Prevention*, which adopted a machine learning-based approach instead of traditional blocklists. It leveraged various website features to detect and block tracking [Weba; Wil17].

Many instances have surfaced where websites effectively exploit workarounds to evade ad blocking and tracking protection measures. For instance, advertising-related companies were observed abusing the WebSocket API as a method to circumvent these protective measures [Bas+18; Bug16].

1.4 Browser engines for cross-platform development

Browser engines are the software components responsible for rendering web pages, and thus for the user-facing functionality of web browsers. Over the course of several years marked by intensive development efforts, these engines have evolved into cross-platform and feature-rich backbones of modern web browsers. Nowadays, the most popular engines among browsers are Blink[7] (used by Chrome, Edge, Opera and Brave), Gecko[8] (used by Firefox and TorBrowser) and WebKit[9] (used by Safari). By observing this, it already becomes apparent that even among browsers, browser engines are extensively shared and reused, in avoidance of reinventing the wheel.

However, the importance of browser engines even extends beyond the realm of web browsing. Their open-source nature has paved the way for a multitude of other software projects to incorporate and repurpose them, thereby contributing to their widespread adoption. For example, the Electron framework repurposed Blink and the V8 JavaScript engine to provide developers with a cross-platform framework for developing desktop applications. Various high-profile applications have been developed using Electron, including Slack, Discord, Microsoft Teams and Visual Studio Code.[10] Also, mobile platforms like iOS and Android provide developers with the option to use browser engines in their applications.

Nevertheless, the reuse of browser engines presents certain subtle challenges that may not be immediately apparent. Web browsers, by default, prohibit implicit access to the local file system due to inherent security concerns. However, native applications incorporating browser engines may not uphold this strict security posture, potentially exposing vulnerabilities. The same holds for other capabilities, such as access to the microphone, camera or location. In these cases, any content loaded by the browser engine could attempt to exploit this lack of constraint. Especially messaging applications, on which users can send arbitrary content to other users, are vulnerable to this. Here, an attacker could attempt to inject malicious scripts into a message. Even more concerning are applications that display arbitrary content without the need to bypass sanitization mechanisms, such as Electronic Publication (EPUB) readers. These

[7] Google forked Blink from WebKit in 2013 because of simplification and performance reasons [Bar13; Chrf].

[8] Gecko was originally released as a complete rewrite of Netscape's browser engine, named Raptor in 1998 and was later rebranded to its current name. The company behind Netscape founded Mozilla in the same year [IV10].

[9] Apple forked WebKit from KHTML as part of the first release of Safari in 2003 [App03; Webb].

[10] A more extensive list of applications can be found on `https://www.electronjs.org/apps`.

applications render e-books authored by anyone, embedding HTML, CSS, and potentially JavaScript that are to be interpreted by the engine.

1.5 Research questions and objectives

As we progressed through the preceding sections of this chapter, we have delved into the broader context of the intricate nature of browsers and the challenges that accompany their development. Much like the construction of Rome, web browsers were not created in a day; they evolved piece by piece into the sophisticated software we rely on today. However, the developers of the earliest browsers could hardly have foreseen the extensive functionalities these applications would come to support, let alone that their code would be adopted by countless native applications. Consequently, the foundational work laid by these pioneers did not account for the features, security measures and privacy concerns that future developers would face. Nonetheless, this groundwork remains in use to this day and continues to be built upon.

Simultaneously, the abundance of security and privacy policies supported by web browsers must correctly coexist with the even greater abundance of supported features. With each new policy introduced, every existing feature must undergo scrutiny to ensure compliance. Vice-versa, every newly introduced feature must be checked against all enforced policies to prevent potential abuse as a bypass. This, for instance, combined with the previously discussed lack of foresight, often leads to the dispersion of policy enforcement mechanisms throughout the code base, typically in the form of conditional statements. This increases both the cost of code maintenance and the likelihood of oversights. To gauge the extent of these issues, we distill three research questions in the remainder of this section.

Prior research has scrutinized a variety of policies, but the majority of these are seemingly complex and intricate policies, such as CSP and SOP. Here, the policy's complexity may be a contributing factor to the issues that have been identified. More straightforward policies, such as those governing cookies and requests, had not been comprehensively studied yet. Bridging this gap could provide valuable insights into the underlying causes of these problems:

> *Question 1: To what extent do implementation flaws, similar to those found for seemingly more intricate browser policies, affect cookie and request blocking policies?*

To establish robust solutions, a thorough comprehension of the fundamental reasons behind these issues is essential. This necessitates an examination of various facets of these bugs, including their lifecycle, the affected code and the associated handling. A comprehensive analysis of a large dataset of bugs can help us better grasp the root causes of these issues:

Question 2: What are the root causes of known browser policy bugs and how can we mitigate them effectively?

As previously discussed, browser engines are not only extensively reused by web browsers, but also repurposed by native applications. Some of these applications have already been reported to be vulnerable to the same types of attacks mounted against browsers. Nevertheless, it remains uncertain to what extent these vulnerabilities affect native applications:

Question 3: How do security and privacy concerns that impact browsers extend to other applications that incorporate browser engines?

In the upcoming sections, we elaborate on each of the posed research questions and couple them with our approach of addressing them.

1.5.1 Cookie and request blocking policy implementation flaws

Cookies have evolved into a fundamental pillar of the World Wide Web, becoming a ubiquitous element within modern web applications. For instance, the significance of cookies is underscored by the challenges and resistance that are delaying the transition to a Web without third-party cookies [Cha22]. Indeed, web services that rely on cookies will have to be re-engineered to function without them, particularly services that use them for authentication or tracking. As we have explored in the previous sections, cookies are subject to a spectrum of policies that impact their functionality. On one hand, browsers have introduced the same-site cookie policy as a countermeasure against CSRF attacks. On the other hand, many browsers now offer various settings to block (third-party) cookies and requests, as do dozens of browser extensions, in response to the privacy concerns associated with widespread tracking.

While prior research has already uncovered flaws in various browser policy implementations, cookie and request policies have not been studied in a comprehensive manner [Agg+10; HMN15; Sin+10; SNM17; Zhe+15]. To achieve this, it is essential to conduct a comprehensive set of experiments by addressing two critical dimensions:

- **Policy coverage**: The experiments should encompass the full spectrum of potential policy configurations.
- **Feature coverage**: The experiments should encompass all the features currently supported by the browser, and all their possible configurations.

In the context of **policy coverage**, the same-site cookie policy covers two settings: `lax` and `strict`. Regarding tracking protection, these policies fall into the "set and forget" category, meaning that once you enable them, they autonomously carry out their designated tasks. The setting to block (third-party) cookies is similar, as it is either enabled or disabled.

The more challenging part here, however, lies with achieving comprehensive **feature coverage**. In this context, coverage should encompass *all* features that can trigger the browser to send a request. This is particularly crucial in relation to tracking: even if only one feature is not covered by the policy's implementation, it can be exploited by trackers as a bypass to undermine the protection in full. Furthermore, features can be used in various embedded browsing contexts, such as within `<iframe>` elements, which could also influence the browser's enforcement behavior. In this light, multiple studies have already demonstrated that trackers can be very creative in finding ways to circumvent tracking protection mechanisms [Dam$^+$22; Dim$^+$21; MM12; RKW12].

Covering both dimensions leads to an extensive set of required experiments to be conducted. Therefore, to make this project feasible, a certain level of automation is required, while being compatible with a wide range of browsers and browser extensions at the same time. As such, our first research goal is stated as follows:

Objective 1: To conduct an automated and comprehensive evaluation of cookie and request blocking policy implementations.

This evaluation, made possible by the development of an automated framework designed to assess various browsers and browser extensions, is detailed in Chapter 2. Here, we uncovered various bypasses in the implementations of the same-site cookie policy, third-party cookie policy and tracking protection mechanisms in virtually every browser.

1.5.2 Root causes of security policy implementation bugs

Despite the abundance of research on web browser security and privacy policies, a fundamental understanding of the underlying causes behind policy implementation issues remains elusive. While previous studies have pointed to the introduction of new features and decentralized enforcement as potential factors contributing to policy implementation bypasses [Agg+10; JB08], no systematic investigation has been conducted to comprehensively identify these root causes. Similarly, various solutions to address implementation flaws have been proposed, but these often focus on the limited set of identified flaws which may not represent the true distribution of underlying causes.

In the field of software engineering, extensive research has examined software system bugs, seeking to discover vulnerabilities and provide remedies, with some studies focusing on browsers [Ale+22; BBB16; Bra+22b; CMN15; FAW13]. However, these studies heavily rely on the completeness and correctness of bug reporting practices and primarily offer conclusions about the bug handling process. We argue that to gain a well-founded understanding of root causes, it is necessary to verify the information in bug reports, including the introducing code changes (which is often not known) and the fixing code changes (which could be incorrect). This requires the ability to trace the lifecycles of vulnerabilities close to 100% accuracy, otherwise the study's conclusions could be flawed. Consequently, methods that rely on static code analysis, like the SZZ algorithm, are unsuitable for this purpose [Bao+22; Ian+23; KPW06; Shi+23]. Furthermore, many of these studies are limited to detecting changes between software release versions, whereas our goal is to identify the exact code change that introduced the bug, necessitating consideration of every single code revision.

In comparison to the previous research objective, this goal involves an even larger number of experiments, as it requires evaluation at the level of software revisions. Hence, automation is especially crucial here. In summary, our second research goal is as follows:

> *Objective 2: To identify the root causes of policy implementation flaws through an automated empirical analysis of bug lifecycles.*

We accomplished this objective by pinpointing almost all publicly released CSP bugs in Chromium and Firefox, and tracing their lifecycles to identify various root causes. Our results, coupled with the framework developed for this purpose, are presented in Chapter 3.

1.5.3 Implications of browser engines in native applications

Unlike our previous research goals, which focused on issues that manifest in web browsers, we now adopt a broader perspective. While it is known that browser engines are susceptible to security and privacy flaws, prior research had not explored whether these vulnerabilities also appear in non-browser applications that incorporate a browser engine. It is worth noting, however, that numerous reports have documented vulnerabilities in frameworks designed for embedding browser engines, such as Electron [Sny]. Although it is certainly undesirable for vulnerabilities to exist in the foundational component of an application, this does not necessarily entail a practical exploit. Furthermore, there may be alternative ways in which these applications can inadvertently introduce vulnerabilities, beyond inheriting weaknesses from the embedded engine.

Web browsers, due to their extensive user base, are attractive targets for attackers. Consequently, browsers are meticulously sandboxed from the user's OS to limit the potential impact of vulnerabilities. For example, websites are generally unable to access files from the user's file system without explicit user consent. However, native applications, particularly on desktop platforms, may have more flexibility in this regard and often possess broader capabilities, including access to the file system. Even with the increasing prevalence of built-in OS permissions, where users are explicitly prompted to grant permission during installation or when an application attempts to use a specific capability for the first time, these measures may not offer absolute protection against potential risks. For instance, if an application requires file access permissions for legitimate use, an attacker who manages to execute arbitrary code within the application will also be able to abuse this capability and access the file system as a result.

To attain a comprehensive understanding of the security and privacy implications tied to the integration of browser engines in desktop applications, a methodical study is imperative. Nonetheless, conducting such an investigation introduces unique challenges compared to scrutinizing browser implementations. A particularly significant hurdle arises from the fact that most desktop applications are closed-source, making direct source code inspection impossible and necessitating a dynamic approach. Even then, the dynamic evaluation of desktop applications proves more complex compared to that of web browsers in the context of our first research objective. While several libraries like Selenium and Puppeteer exist for automating web browsers, equivalent tools for desktop applications are notably absent. Furthermore, browsers tend to exhibit more uniform behavior, often allowing interaction through command-line interfaces. In contrast, desktop applications vary significantly in their usage patterns and functionalities, requiring a more adaptable and dynamic assessment approach. With this in mind, we formulated the following research objective:

Objective 3: To investigate the security and privacy implications of native applications that incorporate browser engines.

For this purpose, we undertook a thorough assessment of EPUB reading systems, a category of native applications that incorporate browser engines, as outlined in Chapter 4. Through the evaluation of 97 applications across seven platforms using a semi-automated testbed, we identified numerous security and privacy issues.

2

Who Left Open the Cookie Jar?

A Comprehensive Evaluation of Third-Party Cookie Policies

It's like a cookie, they all crumble.

– Dr. Dre [DS92]
(Nuthin' but a "G" Thang, 1992)

This chapter was previously published as:

> G. Franken, T. Van Goethem, and W. Joosen. "Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 151–168. ISBN: 978-1-939133-04-5

In this chapter, we introduce a fully automated framework tailored for the exhaustive assessment of cookie and tracking protection policy implementations, the first of its kind. Leveraging this framework, we evaluated the effectiveness of seven web browsers' built-in mechanisms and 46 browser extensions advertised either as ad blocker or anti-tracking tool. Moreover, this framework was published as open-source and was used as a foundation for the framework presented in Chapter 3. Beyond its automation capabilities, the core value of this framework lies in its comprehensiveness: the carefully compiled test suite covered all known mechanisms that can trigger requests, even incorporating various nested browsing contexts.

The results of our evaluation were brutal: virtually every evaluated tracking protection mechanism was shown to be susceptible to circumvention. Partly due to the non-comprehensive WebRequest API leveraged by browser extensions, not a single ad blocking or anti-tracking extension was able to block all mechanisms from triggering third-party requests. As such, users employing these countermeasures were given a false sense of privacy for years on end. Furthermore, several same-site cookie policy bypasses were discovered across various browsers. Even built-in settings to block (third-party) cookies, which had been around for years already, were circumventable in multiple browsers. All identified bypasses were reported to the respective browser vendors and extension developers.

In a follow-up project, published in IEEE's *Security & Privacy Magazine* [FVJ19], we re-evaluated the built-in mechanisms across the latest versions of web browsers, one year after the original evaluation. Regrettably, developers did not manage to rectify all of the reported issues. While the anti-tracking settings did exhibit some enhancements by blocking a greater array of mechanisms, their completeness remained unsatisfactory. Notably, the same-site cookie vulnerabilities were resolved in Chrome and Opera, while Edge and the newly introduced privacy-oriented browser Brave, remained affected.

Our work stands as a compelling argument that the complexities of a correct and comprehensive implementation of browser policies are far from trivial. Despite the ostensible simplicity of most anti-tracking policies, which primarily revolves around blocking requests to any endpoint matching with one of the regular expressions in a blocklist, a significant challenge persists. Within the context of Firefox's Tracking Protection, we confirmed that this challenge is partly rooted in the retroactive introduction of this policy; compliance of pre-existing mechanisms is not centrally enforced, but rather distributed across multiple segments within the code base. This observation corroborates the findings of Aggarwal et al. regarding Firefox's Private Browsing mode [Agg$^+$10]. The potential root causes of policy implementation flaws are further discussed in Chapter 3.

This work was awarded with a *Distinguished Paper Award* and the *2018 Internet Defense Prize* which came with a grant of $100,000. As part of our outreach efforts, we created a dedicated website where we provide a summary of our findings.[1]

[1] `https://wholeftopenthecookiejar.com/`

2.1 Introduction

Since its emergence, the Web has been continuously improving to meet the evolving needs of its ever-growing number of users. One of the first and most crucial improvements was the introduction of HTTP cookies [Bar11a], which allow web developers to temporarily store information such as website preferences or authentication tokens in the user's browser. After being set, the cookies are attached to every subsequent request to the originating domain, allowing users to remain logged in to a website without having to re-enter their credentials.

Despite their significant merits, the way cookies are implemented in most modern browsers also introduces a variety of attacks and other unwanted behavior. More precisely, because cookies are attached to every request, including third-party requests, it becomes more difficult for websites to validate the authenticity of a request. Consequently, an attacker can trigger requests with a malicious payload from the browser of an unknowing victim. Through so-called cross-site attacks, adversaries can abuse the implicit authentication to perform malicious actions through cross-site request forgery attacks [BJM08; ZF08], or extract personal and sensitive information through cross-site script inclusion [Lek+15] and cross-site timing attacks [BB07; GH15; VJN15].

Next to cross-site attacks, the inclusion of cookies in third-party requests also allows for users to be tracked across the various websites they visit. Researchers have found that through the inclusion of code snippets that trigger requests to third-party trackers, the browsing habits of users are collected on a massive scale [Aca+14; RKW12; Yu+16]. These trackers leverage this aggregated information for the purpose of content personalization, e.g. on social networks, displaying targeted advertisements, or simply as an asset that is monetized by selling access to the accumulated data.

As a direct response to the privacy threat imposed by third-party trackers and associated intrusive advertisements, a wide variety of efforts have been made. Most prominently is the emergence of dozens of browser extensions that aim to thwart their users from being tracked online. These extensions make use of a designated browser API [Pie17] to intercept requests and either block them or strip sensitive information such as headers and cookies. Correspondingly, several browsers have recently introduced built-in features that aim to mitigate user tracking. For instance, Firefox in its private browsing mode will by default block third-party requests that are made to online trackers [Moz15]. It is important to note that the effectiveness of these anti-tracking mechanisms fully relies on the ability to intercept or block *every* type of request, as a single exception would allow trackers to simply bypass the policies. In this chapter, we show that in the current state, built-in anti-tracking protection mechanisms as well as virtually every popular browser extension that relies on blocking third-party requests to either prevent user tracking or disable intrusive advertisements, can be bypassed by at least one technique.

Next to tracking protections, we also evaluate a recently introduced and promising feature aimed at defending against cross-site attacks, namely same-site cookies [WG16]. While cross-site attacks share the same cause as online tracking, i.e. the inclusion of cookies on third-party requests, their defenses are orthogonal. The `SameSite` attribute on cookies can be set by a website developer, and indicates that this cookie should only be included with first-party requests. Consequently, when this policy is applied correctly, same-site cookies defend against the whole class of cross-site attacks. Similar to the tracking defenses, the security guarantees provided by same-site cookies stand or fall by the ability to apply its policies on *every* type of request. As part of our evaluation, we discovered several instances in which the same-site cookie policy was not correctly applied, thus allowing an adversary to send authenticated requests regardless of the `lax` or `strict` mode applied to the same-site cookie. Although this bypass could only be used to trigger `GET` requests, thereby making the exploitation of CSRF vulnerabilities in websites that follow common best-practices more difficult, it does underline the importance of a systematic evaluation to test whether browser implementations consistently follow the policies proposed in the specification.

In this chapter, we present the first extensive evaluation of policies applied to third-party cookies, whether for the purpose of thwarting cross-site attacks or preventing third-party tracking. This evaluation is driven by a framework that generates a wide-range of test cases encompassing all methods that can be used to trigger a third-party request in various constructs. Our framework can be used to launch a wide variety of different browsers, with or without extensions, and analyze, through an intercepting proxy, whether the observed behavior matches the one expected by the browser instance. We applied this framework to perform an analysis of 7 browsers and 46 browser extensions, and found that for virtually every browser and extension the imposed policy can be bypassed. The sources for these bypasses can be traced back to a variety of implementation, configuration and design flaws. Further, our crawl on the Alexa top 10,000 did not identify any use of the discovered bypasses in the wild, indicating that these are novel.

Our main contributions are the following:

- We developed a framework with the intent to automatically detect bypasses of third-party request and cookie policies. This framework is applicable to all modern browsers, even in combination with a browser extension or certain browser settings.

- By applying the framework to 7 browsers, 31 ad blocking and 15 anti-tracking extensions, we found various ways in which countermeasures against cookie leaking can be bypassed.
- We performed a crawl on the Alexa top 10,000, visiting 160,059 web pages, to inspect if any of these bypasses were already being used on the Web. In order to estimate the completeness of our framework, we analyzed the DNS records spawned by each web page.
- Finally, we propose solutions to rectify the implementations of existing policies based on the detected bypasses.

2.2 Background

A fundamental trait of the modern web is that websites can include content from other domains by simply referring to it. The browser will fetch the referenced third-party content by sending a separate request, as shown in Figure 2.1. The web page of `first-party.com` contains a reference to an image that is hosted on `third-party.com`. In this scenario, the user first instructs his browser to visit this web page, e.g. by entering the address in the address bar or by clicking on a link. This will initiate a request to the web page `http://first-party.com/`, and a subsequent response will be received by the browser (1). While parsing the web page, the user's browser comes across the reference to `https://third-party.com` and fetches the associated resource by sending a separate request (2). The browser will include a `Cookie` header [Bar11a] to the request if these were previously set for that domain (using the `Set-Cookie` header in a response). This applies to both the request to `first-party.com` as well as `third-party.com`. In this scenario, we would name the cookies attached to the latter request *third-party cookies*, as this is a request to a different domain than the including document.

2.2.1 Cross-site attacks

Because browsers will, by default, attach cookies to any request, including third-party requests, an adversary is able create a web page that constructs malicious payloads which will be sent using the victim's authentication. Through these so-called cross-site attacks, attackers can trigger state changes on vulnerable websites or extract sensitive information.

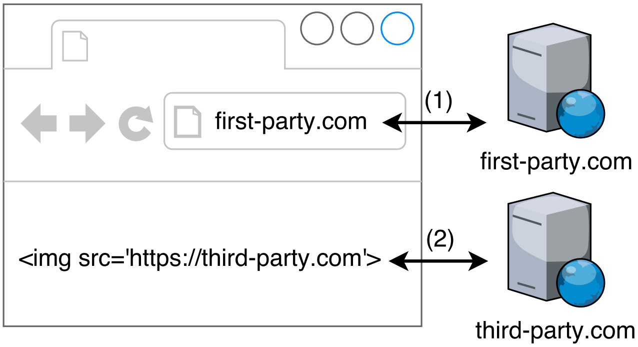


Figure 2.1: Example of a cross-site request.

One of the most well-known cross-site attacks is CSRF. CSRF attacks aim to perform undesirable actions, e.g. transfer funds to the account of the adversary, on behalf of the victim who is authenticated at the vulnerable website. Typically, this will be done by triggering a `POST` request to the targeted website, as it is considered best-practice to prevent `GET` requests from having any state-changing effect [Fie+99]. Although websites of large organizations such as The New York Times, ING, MetaFilter and YouTube have been found to be vulnerable to CSRF attacks in the past [ZF08], the increased awareness among web developers and countermeasures integrated in popular frameworks resulted in a drastic decrease in vulnerable websites. According to the OWASP Top Ten Project, only 5% of current websites were found to be vulnerable, thus leading to the exclusion of CSRF from the list of the ten most critical web application security risks. Effective countermeasures, such as requiring an unguessable token in requests, have been known for a long period [BJM08; ZF08], and have been extensively applied [Van+14].

In contrast to CSRF, Cross-Site Script Inclusion (XSSI) and cross-site timing attacks aim to derive sensitive information. XSSI attacks bypass the SOP in an attempt to obtain information linked to the authenticated user account [Lek+15]. Timing attacks, on the other hand, try to construct sensitive data by observing side-channel leaks [BB07; GH15; VJN15].

A recently proposed mechanism called same-site cookies aims to protect against the whole class of cross-site attacks [WG16]. Same-site cookies are generic cookies with an additional attribute named `SameSite`. Similar to other cookie attributes, the `SameSite` attribute is determined by the website that sets the cookie. This attribute can be given one of two values: `lax` or `strict`. When the value is set to `lax`, the cookie may only be included in cross-site `GET` requests that are top-level (i.e. the URL in the address bar changes due to the request). An exception to this is a cross-site request initiated by Prerender functionality [Chr11], in which this cookie is included anyway. When the attribute value is set to `strict`, the cookie may never be included in any cross-site requests.

At the time of writing, same-site cookies are supported by Chrome, Opera, Firefox and Edge [KMG18; Mic18; Sta17]. Same-site cookies are backwards compatible; browsers that do not offer support will just treat same-site cookies as regular cookies. This, combined with the fact that same-site cookies are mainly intended as an in-depth defense mechanism, encourages web developers to still employ traditional defenses such as CSRF tokens to thwart cross-site attacks. While the adoption of same-site cookies is still relatively small, with only a few popular websites implementing them [Sha17], the fact that they can mitigate a whole class of attacks makes them a very promising defense mechanism.

2.2.2 Third-party tracking

Internet users can be tracked for a variety of purposes, often with economic motives as the driving force behind it, e.g. advertising, user experience or data auctioning [MM12]. One way of employing online tracking is through embedded advertisements, which include tracking scripts to learn more about the user's interests and personalize the advertisements based on this information. Alternatively, website administrators may include scripts from analytic services, which gather insights in how users interact with their website, provided that this service can also use the collected data for its own purposes. Moreover, websites may embed functionality of a social platform through which users can engage with each other. Because the resource containing embedded functionality is requested upon each page visit, the social platform can track which websites their users visit.

The main technique that is used to track users across different websites is by means of third-party cookies. More precisely, a script that is included on a wide range of websites, e.g. to display advertisements, triggers a request to the server of the tracker. Subsequently, the tracker checks whether this request contains a cookie, and either associates the triggered request with the profile of the user, or creates a new profile and responds with a `Set-Cookie` header containing the newly generated cookie. In the latter case, the user's browser will associate the cookie with the site of the tracker, and will include it in all subsequent requests to it. This allows the tracker to follow users across all websites that include a script that initiates the request to the tracker.

Because of the raised awareness of online tracking among the general public, many users delete cookies on a regular basis [com11], which results in a seemingly new user profile from the tracker's perspective. As a reaction, some online trackers have resorted to more extensive tracking methods, such as respawning cookies via Flash [Sol+09] and other web mechanisms [Aye+11], and browser fingerprinting [Aca+14; Eck10; Sol+09; Yen+12]. As the evaluation presented in this chapter mainly focuses on cookie policies imposed by browsers or browser extensions, our main focus is on "traditional" user tracking by means of third-party cookies. However, because the more recent tracking mechanisms also rely on sending requests to the tracker, e.g. containing the browser fingerprint, these are also subjected to the browser and extension policies. Bypasses of these policies can also be leveraged by trackers to smuggle their requests past the protection mechanisms.

2.3 Framework

Despite all standardization efforts, browser implementations may exhibit inconsistent behavior or even deviate from the standard. Additionally, web features from different standards may interfere with each other, causing unintended side-effects, which may affect the security and privacy guarantees. Despite prior efforts to verify these guarantees [JTL12; Ler+13], the real-world prevalence of inconsistencies remains hard to measure as modern browsers consist of millions of lines of code, or may be proprietary, preventing researchers access to their source code. In this chapter, we evaluate the validity of constraints that are imposed on stateful third-party requests, either by browsers themselves or by browser extensions. Because of the limitations of source-code analysis, we design a framework that considers browsers, in various configurations, as a black box. This section outlines the design choices and implementation of this framework. The source code of our framework has been made publicly available.[2]

2.3.1 Framework design

The goal of our framework is to detect techniques that can be used to circumvent policies that strip cookies from cross-site requests, or that try to block these requests completely. To achieve this, our framework consists of various components ranging from browser control to test-case generation. These

[2] `https://github.com/DistriNet/xsr-framework`

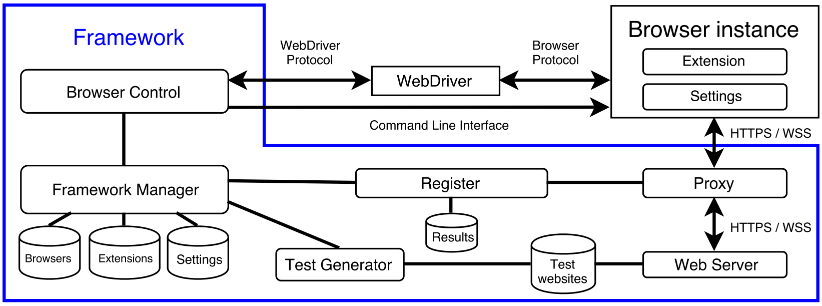


Figure 2.2: Design of the framework that we used to detect bypasses of imposed cross-site request policies.

components and their interactions are depicted in Figure 2.2, and discussed in the following sections.

Browser manipulation

The framework is driven by the *Framework Manager* component, which is provided with information on which browsers and browser extensions need to be analyzed. The manager instructs the *Browser Control* component to create a specific browser instance with the predefined settings. The controller will then instruct the browser instance to visit one of the generated test-cases by leveraging browser-specific Selenium WebDriver[3] implementations. Browsers that do not have Selenium support, are controlled by manually configuring a browser profile and are then launched through the command-line.

Test environment

Prior to executing all test scenarios, the browser instance is first prepared. More specifically, on the target domain, i.e. the domain for which the test cases will try to initiate an illegitimate cross-site request, we install several cookies. Each of these cookies has different attributes: none, which does not impose any restrictions on the cookies, `HttpOnly`, which restricts the cookie from being accessed by client-side scripts, and `Secure`, which only allows this cookie to be sent over an encrypted connection. Throughout the remainder of the text, we refer to cookies as cookies with any one of these attributes, unless explicitly stated otherwise. Furthermore, for browsers that support it, we installed two

[3] `https://www.seleniumhq.org/`

cookies with the SameSite attribute: one with the value set to lax, and one set to strict. Finally, we instruct the browser to route all requests through a proxy, allowing us to capture and analyze the specific requests that were initiated as part of a test.

2.3.2 Test-case generation

Because of the abundance of features and APIs implemented in modern browsers, there exist a very large number of techniques that can be leveraged to trigger a cross-site request. For each such technique, our framework generates a web page containing a relevant test case.

Request-initiating mechanisms

As there exists no comprehensive list of all feature that may initiate a request, we leveraged the test suites from popular browser engines, such as WebKit, Firefox, as well as the web-platform-tests project by W3C[4] to compose an extensive list of different request methods. In addition, we analyzed several browser specifications to verify the completeness of this list. What follows is a summary of the mechanisms we used, subdivided into seven different categories.

HTML tags The first group of request mechanisms consists of HTML elements that can refer to an external resource, such as <img>, <iframe> or <script> tags. Upon parsing the HTML document, the browser will initiate requests to fetch the referred resources. As a basis, we used the HTTPLeaks project[5], which contains a list of all possible ways HTML elements can leak HTTP requests. This list was combined with techniques related to features that were recently introduced, and account for 196 unique methods. It should be noted that all HTML-based requests only initiate GET requests.

Response headers Response headers allow websites to include extra information alongside the resource that is served. We found that two classes of response headers may trigger an additional request, either as soon as the browser receives the headers or upon certain events. The first class of such response headers are Link headers, which indicate relationships between web resources [Not10]. The header can be used to improve page-load speeds by signaling to the browser which resources, such as stylesheets and associated web pages, can proactively

[4] https://github.com/w3c/web-platform-tests

[5] https://github.com/cure53/HTTPLeaks

be fetched. In most cases, the browser will request the referenced resources through a `GET` request.

The other class of response headers that initiate new requests are related to CSP [WS23]. More precisely, through the `Content-Security-Policy` header[6], a website can, among other things, indicate which resources are allowed to be loaded. Through the `report-uri` directive, websites can indicate that any violations of this policy should be reported, via a POST request to the provided URL. Recently, another directive named `report-to` has been proposed, which allows reporting through the Reporting API [GW17]. As this directive and API are not yet supported by any browser, we excluded them from our analysis. Nevertheless, they are a prominent example of the continuously evolving browser ecosystem, and highlight the importance of analyzing the unexpected changes new features might bring along.

Redirects Top-level redirects are often not regarded as cross-site requests, because stripping cookies from them would cause breakage of many existing websites. Nevertheless, we included them in our evaluation for the sake of completeness, because various scenarios exist in which top-level redirects can be abused. For instance, a tracker trying to bypass browser mitigations can listen for the `blur` event on the `window` element, which indicates that the user switched tabs. When receiving this event, the tracker could trigger a redirect to its own website in the background tab, which would capture information from the user and afterwards redirect him back to the original web page. In our framework, we evaluate redirection mechanisms through the `Location` response header, via the `<meta>` tag, setting the `location.href` property and automatically submitting forms.

JavaScript Browsers offer various JavaScript APIs that can be used to send requests. For instance, the XMLHttpRequest API can be used to asynchronously send requests to any web server [Moz17f]. More recently, the Fetch API was introduced, which offers a similar functionality and intends to replace XMLHttpRequest [Moz17b]. Similarly, the Beacon API can be used to asynchronously send `POST` requests, and is typically used to transmit analytic data as it does this in a non-blocking manner and the browser ensures the request is sent before the page is unloaded [Moz17a]. Finally, there are several browser features that allow web developers to set up nonstandard HTTP connections. For instance, the WebSocket API can be used to open an interactive communication session between the browser and the server [Moz17e]. Also, the EventSource

[6] There also exist experimental CSP headers such as `X-Content-Security-Policy` and `X-WebKit-CSP`, as well as a report-only header.

API can be used to open a unidirectional persistent connection to a web server, allowing the server to send updates to the user [Moz18a]. The latter two mechanisms are initiated using a `GET` request.

PDF JavaScript In addition to statically showing information, PDFs also have dynamic features that are enabled through JavaScript code embedded within the PDF file. For example, through the JavaScript code it is possible to trigger `POST` requests by sending form input data. The capabilities of the PDF and the JavaScript embedded within it, depend on the viewer that is used. Next to the system-specific viewer, some browsers also implement their own PDF viewer, which shows the contents in a frame. The viewer used by Chrome and Opera, PDFium [Gooc], is implemented as a browser extension and does support sending requests. To our knowledge, this is not the case for Firefox' PDF.js library [Git], as we did not manage to simulate this, nor did we find any source to confirm this.

AppCache API Although the AppCache API has been deprecated, it is still supported by most browsers [Moz18b]. This mechanism can be used to cache specific resources, such that the browser can still serve them when the network connection is lost. Web developers can specify the pages that should be cached through a manifest file. When the browser visits a page that refers to this file, the specified resources, which may be hosted at a different domain, will be requested through a `GET` request and subsequently cached.

Service Worker API Service workers can be seen as a replacement for the deprecated AppCache API. They function as event-driven workers that can be registered by web pages. After the registration process, all requests will pass through the worker, which can respond with a newly fetched resource or serve one from the cache. Next to fetching the requested resources, service workers can also leverage most[7] browser APIs to initiate additional requests.

Test compositions

The most straightforward way to initiate a new request is to include the mechanism directly in the top-level frame. For example, for the purpose of tracking, web developers typically include a reference to a script or image hosted at the tracker's server. However, because their top-level document can include different documents through frames, it is possible to create more advanced test

[7] XMLHttpRequest is not supported in service workers.

compositions. In our framework, we tested 8 test-case compositions, where resources from different domains were included in each other, either through an `<iframe>` or by specific methods, such as `importScripts` in JavaScript. As we did not detect any behavior related to the test-case compositions, we omitted the details from the chapter. We refer to Appendix A.1 for an overview of the different compositions that were used.

2.3.3 Supported browser instances

In order to generalize our results, and detect inconsistencies we evaluated a wide variety of browser configurations. These configurations range over the different browsers and their extensions, considering all the relevant settings.

Web browsers

The primary goal of our evaluation was to analyze browsers for which inconsistencies and bypasses would have the largest impact. On the one hand, we included the most popular and widely used browsers: Chrome, Opera, Firefox, Safari and Edge. On the other hand, we also incorporated browsers that are specifically targeted towards privacy-aware users, and thus impose different rules on authenticated third-party requests. For instance, Tor Browser makes use of double-keyed cookies: instead of associating a cookie with a single domain, the cookies are associated with both the domain of the top-level document as well as domain that set the cookie. For example, when siteA.com includes a resource from siteB.com that sets a cookie, this cookie will not be included when siteC.com would include a resource from siteB.com. Finally, we also included the Cliqz browser, which has integrated privacy protection that is enforced by blocking requests to trackers.

Browser settings

Most modern web browsers provide an option to block third-party cookies. While this can be considered as a very robust protection against both cross-site attacks and third-party tracking, it may also interfere with the essential functionality for websites that rely on cross-site communication. Moreover, some browsers provide built-in functionality to prevent requests from leaking privacy-sensitive information. For instance, Opera offers a built-in ad blocker that is based on blacklists. By default, the anti-tracking and ad blocking lists from EasyList and EasyPrivacy are used, but users are able to also define custom ones. In our framework, we only considered the default setting of the built-in protection. Another browser that provides built-in tracking protection is Firefox. Here, the mechanism is enabled by default when browsing in “Private mode”, and also leverages publicly available and curated blacklists [KC15].

Recently, Safari introduced its own built-in tracking protection, which uses machine learning algorithms to determine the blacklist [Weba]. Requests sent to websites on this blacklist are subjected to cookie partitioning and other measures to prevent the user from being tracked. For example, cookies will only be included in a cross-site request when there was a first-party interaction within the last 24 hours with the associated domain. Although we were unable to infer the rules of these machine learning algorithms, we still subjected this built-in option to our framework in order to be complete.

Browser extensions

Next to built-in tracking prevention, users may also resort to extensions to prevent their browsing behavior and personal information from leaking to third parties. As these extensions may also impose restrictions on how requests are sent, and whether cookies should be sent along in third-party requests, we also included various anti-tracking and ad blocking extensions. Due to the excessive amount of such extensions, we were unable to test all. Instead, we made a selection based on the extension’s popularity, i.e. the total number of downloads or active users, as reported by the extension store. In total, we evaluated 46 different extensions for the 4 most popular browsers (Chrome, Opera, Firefox and Edge). An overview of all extensions that were evaluated can be found in Appendix A.2.

Most browsers’ anti-tracking and ad blocking extensions share a common functionality. By making use of the WebRequest API [Moz17d], extensions can inspect all requests that are initiated by the browser. The extension can then determine how the request should be handled: either it is passed through unmodified, or cookies are removed from the request, or the request is blocked entirely. This decision is typically made based on information about the requests, namely whether it is sent in a third-party context, which element initiated it, and most importantly, whether it should be blocked according to the block list that is used. It should be noted that for the browser extension to work correctly, it should be able to intercept *all* requests in order to provide the promised guarantees. This is exactly what we evaluate by means of our framework.

2.4 Results

By leveraging our framework that was introduced in Section 2.3, we evaluated whether it was possible to bypass the policies imposed on third-party requests by either browsers or one of their extensions. The results are summarized in Table 2.1, Table 2.2, and Table 2.3, and will be discussed in more detail in the remainder of this section. These three tables follow a similar structure. For each category of request-triggering mechanism, we indicate whether a cookie-bearing request was made for at least one technique within this category using a full circle (●). A half circle (◐) indicates that for at least one technique within that category a request was made, but that in all cases all cookies were omitted from the request. Finally, an empty circle (○) indicates that none of the techniques of that category managed to initiate a request. Note that these results only reflect regular, `HttpOnly` and `Secure` cookies. Same-site cookies are discussed in Section 2.4.3. We refer to a more detailed explanation about the bug reporting in Appendix A.3 through the indicated [bug#] tags. For a more detailed view of detected leaks and leaks for future browser and extension versions, we kindly direct you to our website.[8]

2.4.1 Web browsers and built-in protection

The results of applying our framework to the 7 evaluated browsers, both with their default settings as with the built-in measures that aim to prevent online tracking enabled, are outlined in Table 2.1. All tests are performed on the browser versions mentioned in this table, unless stated otherwise. In general, it can be seen that differences in browser implementations, often lead to differences in results. The most relevant results are discussed in more detail in the following sections.

Default settings

Under default configuration, nearly all of the most widely used browsers send along cookies with all third-party requests. Exceptionally, due to enabling its tracking protection by default, Safari only does so for redirects. We will discuss this further in Section 2.4.1 with the other evaluated built-in options.

Besides Safari, the privacy-oriented browsers also generally perform better in this regard: with a few exceptions, both Cliqz and Tor Browser manage to exclude cookies from all third-party requests. Most likely because redirects are

[8] `https://WhoLeftOpenTheCookieJar.com`

	AppCache	HTML	Headers	Redirects	PDF JS	JavaScript	SW
Chrome 63	●	●	●	●	●	●	●
- Block third-party cookies	◐	◐	◐	●	●	◐	◐
Opera 51	●	●	●	●	●	●	●
- Block third-party cookies[*]	◐	◐	◐	●	●	◐	◐
- Ad Blocker	●	●	○	●	○	●	●
Firefox 57	●	●	●	●	○	●	●
- Block third-party cookies	◐	◐	◐	●	○	◐	◐
- Tracking Protection	●	●	●	●	○	●	●
Safari 11	○[†]	◐	○	●	○	◐	N/A
- No Intelligent Tracking Prevention	●[†]	●	○	●	○	●	N/A
- Block third-party cookies[‡]	●[†]	●	◐	●	○	●	N/A
Edge 40	●	●	◐	●	○	●	N/A
- Block third-party cookies	●	●	◐	●	○	●	N/A
Cliqz 1.17[*]	◐	●	◐	●	○	◐	◐
- Block third-party cookies	◐	◐	◐	●	○	◐	◐
Tor Browser 7	○	◐	◐	●	○	◐	N/A

●: request with cookies ◐: request without cookies ○: no request

[*] Secure cookies were omitted in all requests.

[†] Safari does not permit cross-domain caching over https (only over http).

[‡] Safari 10.1.2

Table 2.1: Results from the analysis of browsers and their built-in security and privacy countermeasures.

not considered as cross-site (as the domain of the document changes to that of the page it is redirected to), cookies are not excluded for redirects. However, as we outlined in Section 2.3.2, this technique could still be used to track users under certain conditions.

```
<img src="data:image/svg+xml,
<svg>
<image xlink:href='https://
third-party.com/leak'>
</image>
</svg>">
```

Listing 2.1: Bypass technique found for Cliqz

Another interesting finding is that in the HTML category, we found that for several mechanisms Cliqz would still send along cookies with the third-party request. An example of such a mechanism is shown in Listing 2.1. Here an `<img>` element included an SVG via the `data:` URL. Possibly, this caused a confusion in the browser engine which prevented the cookies from being stripped.

Third-party cookie blocking

In addition to the default settings, we also evaluated browsers when these were instructed to block all third-party cookies. For Tor Browser, this feature was already enabled by default. Consequently, Table 2.1 contains no results for Tor Browser under these settings.

Similar to what could be seen from the results of the privacy-oriented browsers, top-level redirects are not considered as third-party, and thus do not prevent a cookie to be sent along with the request. One of the most surprising results is that the browsers that use the PDFium reader to render PDFs directly in the browser (Google Chrome and Opera), would still include cookies for third-party requests that are initiated from JavaScript embedded within PDFs [bug1]. Because PDFs can be included in iframes, and thus made invisible to the end user, and because it can be used to send authenticated `POST` requests, this bypass technique could be used to track users or perform cross-site attacks without raising the attention of the victim. This violates the expectations of the victim, who presumed no third-party cookies could be included, which should safeguard him completely from cross-site attacks. At the time of writing, PDFium only provides support for sending requests, but does not capture any information about the response. As such, XSSI and cross-site timing attacks are currently not possible. However, as indicated in the source code[9], this functionality is planned to be added.

Because the option to block third-party cookies was removed from the latest Safari, we had to use a previous version (Safari 10). We found that setting cookies in a third-party context was successfully blocked. However, cookies - set in a first-party context - were still included in cross-site requests [bug2]. On top of that, we also found that Safari's option to block all cookies suffered from somewhat the same problem. Likewise, it managed to block the setting of third-party cookies, but cookies that were set before enabling this option were still included in cross-site requests. This problem was solved in Safari 11 by deleting all cookies upon enabling the option to block all cookies.

For Edge, we found that, surprisingly, the option to block third-party cookies had no effect: all cookies that were sent in the instance with default settings, were also sent in the instance with custom settings [bug3]. We believe that this may have been the result of a regression bug in the browser, which disabled support for this feature but did not remove the setting.

[9] `https://chromium.googlesource.com/chromium/src/+/66.0.3343.2/pdf/out_of_process_instance.cc#1437`

Built-in protection mechanisms

In total, we evaluated three built-in mechanisms that protect against tracking (Firefox' and Safari's tracking protection mode), or block advertisements (Opera's ad blocker). For Firefox and Opera, our framework managed to detect several bypasses. Although Opera's ad blocker managed to block all requests that were triggered by headers or by JavaScript embedded in PDFs, in all other categories cookie-bearing requests were made [bug4]. Although it did manage to block certain requests, e.g. for HTML tags, out of the 58 requests that were sent in the regular browsing mode, 6 were not blocked. These 6 bypass techniques spanned different browser mechanisms (CSS, SVG, `<input>` and video), so it is unclear why these are treated differently.

For Firefox, we observed comparable results: although many requests were blocked (e.g. for the HTML category, 46 out of 51 requests were blocked), for each applicable category there was at least one technique that could circumvent the tracking protection [bug5]. By analyzing the Firefox source code, we traced the cause of these bypasses back to inconsistencies in the implementation. We discuss this in more detail in Section 2.6.1.

In contrast to the former built-in options, Safari's Intelligent Tracking Prevention managed to mitigate all third-party cookies to a tracking domain, apart from redirects. However, we found that future completeness can be undermined by having this option disabled for even a short interval. Third-party cookies set in this interval by tracking domains, which otherwise would have been prevented, will still be included in cross-site requests after enabling the option again, identical to the results when the option is disabled. Luckily, this option is enabled by default, so future completeness can only be affected through explicit disabling by the user. As we already mentioned in Section 2.3.3, third-party cookies will be included if first-party interaction has been occurred in the last 24 hours. This can be provoked by redirects or pop-ups to the tracking domain, although pop-ups are blocked by default.

2.4.2 Browser extensions

In total, we evaluated 31 ad blocking and 15 tracking protection extensions. The results are summarized in Table 2.2 and Table 2.3 respectively. Due to space constraints, we aggregated extensions in different sets when these shared the same category-level results. Note that within a single set, extensions may still exhibit different results within one category. An overview of all browser extensions that were considered can be found in Appendix A.2. Guided by the resulting data, we found several common causes for the discovered bypasses.

		AppCache	HTML	Headers	Redirect	PDF JS	JavaScript	SW
Chrome	SET A1 (3/14)	●	●	●	●	●	●	●
	SET A2 (3/14)	●	○	◐	●	●	●	●
	SET A3 (1/14)	●	○	○	●	●	●	●
	SET A4 (1/14)	●	○	○	●	●	○	●
	SET A5 (1/14)	●	○	○	○	●	●	●
	SET A6 (3/14)	●	○	○	○	●	○	●
	SET A7 (2/14)	○	○	○	●	●	○	○
Opera	SET A8 (2/9)	●	●	●	●	●	●	●
	SET A9 (1/9)	●	○	◐	●	●	●	●
	SET A10 (2/9)	●	○	○	●	●	●	●
	SET A11 (1/9)	●	○	○	●	●	○	●
	SET A12 (1/9)	●	○	○	○	●	●	●
	SET A13 (1/9)	●	○	○	○	●	○	●
	SET A14 (1/9)	○	○	○	●	●	○	○
Firefox	SET A15 (2/5)	●	●	◐	●	○	●	○
	SET A16 (1/5)	●	●	○	●	○	○	○
	SET A17 (1/5)	●	●	○	○	○	○	○
	SET A18 (1/5)	○	●	○	●	○	○	○
Edge	SET A19 (1/4)	●	●	◐	●	○	●	N/A
	SET A20 (1/4)	●	○	○	●	○	●	N/A
	SET A21 (1/4)	○	●	○	●	○	●	N/A
	SET A22 (1/4)	○	○	○	●	○	●	N/A

●: request with cookies ◐: request without cookies ○: no request

Table 2.2: Results from the analysis of ad blocking extensions per browser.

Considering the results of all Chrome- and Opera-based extensions, it is clear that none of these managed to block the cookie-bearing third-party request when the request is initiated by JavaScript code embedded within a PDF. Although this result is similar to the results we observed when the browser was instructed to block all third-party cookies, the specific cause slightly differs. As the requests are sent from within a browser extension, the browser does not regard it as a cross-site request, and thus does not strip its cookies in the case when the "block third-party cookies" setting is enabled. However, another issue arises when a browser extension wants to block these requests: the WebExtension API does not allow an extension to intercept traffic from another extension. Consequently, this issue can not be mitigated by the anti-tracking and ad blocking extension developers [bug6].

Only few browser extensions correctly block cross-site requests initiated through the AppCache API. By analyzing the source code of the bypassed extensions, we found that these shared the same root cause. Although the listener for the `onBeforeRequest` event was always able to intercept the request, the extensions verified the provided tab identifier. However, for requests that originated from AppCache, this identifier was set to `-1`, a value that was not expected by the extension, as it may also be related to inherent browser functionality such as address bar autocompletion. As extension developers try to prevent interfering with regular browsing behavior, most extensions performed no actions on requests that caused these unexpected parameters [bug8].

		AppCache	HTML	Headers	Redirect	PDF JS	JavaScript	SW
Chrome	SET B1 (1/6)	●	●	●	●	●	●	●
	SET B2 (1/6)	●	●	●	○	●	●	●
	SET B3 (3/6)	●	○	○	●	●	●	●
	SET B4 (1/6)	●	○	○	○	●	○	●
Opera	SET B5 (1/4)	●	●	●	●	●	●	●
	SET B6 (2/4)	●	○	○	●	●	●	●
	SET B7 (1/4)	●	○	○	○	●	●	●
Firefox	SET B8 (1/4)	●	●	●	●	○	●	●
	SET B9 (1/4)	●	●	○	●	○	●	○
	SET B10 (1/4)	●	●	○	○	○	●	○
	SET B11 (1/4)	◐	◐	◐	●	○	●	●
Edge	SET B12 (1/1)	●	●	○	●	○	●	N/A

●: request with cookies ◐: request without cookies ○: no request

Table 2.3: Results from the analysis of tracking protection extensions per browser.

Furthermore, we found that for requests initiated from service workers bypasses were made possible due to the same reasons. However, in this case Firefox-based extensions did manage to block the third-party requests. We found that this is because Firefox assigns the tab identifier to the tab on which the service worker was originally registered. As a result, from the perspective of the browser extension this seemed as a regular request, thus allowing the normal policies to be applied. In total, we found that 26 browser extension policies could be bypassed with the AppCache technique, and 20 through service workers.

Contrasting to extensions of other browsers, almost every Firefox-based extension could be bypassed in the HTML category. In most cases, this was caused by a `<link>` element, which `rel` attribute was set to `"shortcut icon"`. By further analyzing this case, we traced back the cause of this issue to an implementation bug in the WebExtension API. We found that the `onBeforeRequest` event did not trigger for requests originating from this link element [bug7]. Although abusing this bug may not be straightforward, as it is only sent when a web page is visited for the first time, it does indicate that browsers exhibit small inconsistencies, which may often lead to unintended behavior.

In the JavaScript category, we found that most extensions could be bypassed with at least one technique: for the tracking extensions, only a single extension managed to block requests initiated by JavaScript. Most prevalently, a bypass was made possible because of WebSocket connections. We found that a common mistake extension developers made, was in the registration on the `onBeforeRequest` event. The bypassed extensions set the filter value to `[http://*/*, https://*/*]`, which would allow intercepting all glshttp requests, but not WebSockets, which use the `ws://` or `wss://` protocol [bug8]. Hence, to be able to intercept all requests, the filter should include these protocols or use `<all_urls>`. Of course, the configuration of the manifest file should be updated accordingly.

In summary, we found that for every built-in browser protection as well as for every anti-tracking and ad blocking browser extension, there exists at least one technique that can bypass the imposed policies. Moreover, we found that most instances could be bypassed by using different techniques, which have different causes.

2.4.3 Same-site cookie

Through the tests we performed to evaluate the validity of same-site cookies, we detected incorrect behaviors for Chrome, Opera and Edge. No bugs were found for Firefox' implementation of this policy.

For Chrome and Opera, the incorrect behavior was caused by the prerendering functionality [Chr11]. By including `<link rel="prerender" href="...">` on a web page, the visitor's browser will initiate a request to the referenced web page. If this web page resides on another domain, the resulting cross-site request will include all same-site cookies [bug9]. This bypasses the same-site cookie policy as defined by the Internet Draft; only same-site cookies in lax mode are allowed to be included.

For Edge (versions 16 and 17, which support same-site cookies), we detected similar incorrect behaviors, although caused by different functionalities [bug10]. Here, `<embed>` and `<object>` tags can be leveraged to send cross-site requests that include all same-site cookies, by pointing to another domain using the `src` and `data` attributes respectively. This also holds for requests that are sent for opening a cross-site WebSocket connection through the WebSocket API. No same-site cookies should be included at all in these requests according to the Internet Draft. On top of that, we also found that same-site cookies in `strict` mode are included in requests initiated by a variety of redirects, while this is only allowed for same-site cookies in `lax` mode. This was detected for redirects through the `<meta>` tag, `location.href` property and `Location` response header.

2.5 Real-world abuse

Tracking companies and advertisers have been reported to circumvent ad blockers and anti-tracking extensions. For example, due to limitations of the WebExtension API, Pornhub managed to circumvent all ad blocking extensions by leveraging WebSockets [Bug16]. As a response, several popular ad blocking extensions such as Adblock Plus and uBlock implemented a mitigation that would override the WebSocket prototype. Soon after, this mitigation was again circumvented by Pornhub, who this time leveraged WebWorkers.[10] Only when support for intercepting WebSocket connections was added to the WebExtension API, browser extensions managed to prevent Pornhub's bypasses. However, as our results show, not all browser extensions have adopted these defenses. Motivated by the seemingly strong incentives of certain trackers to circumvent request and cookie policies imposed by browser extensions, we performed an experiment to analyze whether any of the bypass techniques introduced in this chapter are actively being used in the wild.

2.5.1 Use of bypass methods

We performed a crawl of the 10,000 most popular websites according to Alexa. For each website, we visited up to 20 pages with a Headless Chrome instance (version 64.0.3282.119, on Ubuntu 16.04), and analyzed all requests that were initiated by one of the new bypass techniques we reported in Section 4.5. In total, 160,059 web pages were visited by our crawler, and on each page we analyzed all third-party requests.

Next, we determined whether a cross-site request should be classified as tracking or advertising. To this purpose, we used the EasyList and EasyPrivacy lists[11] which contain regular expressions used by various popular browser extensions to determine whether requests should be blocked. In Table 2.4, we show the number of unique tracking or advertising domains, that make use of one of the bypass techniques that we found to be most successful. We only count the second-level domain name of the tracker or advertiser to whom the request was sent.

To evaluate whether the advertising or tracking host leveraged one of the techniques to purposely circumvent browser extensions, we visited the web pages on which these trackers or advertisers were included. For each page visit, we enabled the browser extension that may be bypassed with the detected technique. We found that all uses of the methods were legitimate, and the requests to the trackers and advertisers were never initiated because either the script or frame containing the bypass functionality was preemptively blocked. Although we did not encounter any intentional abuse in the 10,000 websites we analyzed, it is possible that trackers may actively try to avoid detection, for instance by only triggering requests upon human interaction. Moreover, as

[10] `https://github.com/gorhill/uBlock/issues/1936`

[11] `https://easylist.to/`

Category	Technique	Tracking domains	Advertising domains
AppCache	`CACHE:`	0	1
Header	`Link: <url>; rel=next`	0	0
	`Link: <url>; rel=prefetch`	0	1
	`CSP: report-uri: url`	8	1
JS	`sendBeacon(url)`	56	18
	`new WebSocket(url)`	27	7
HTML	`<link rel="shortcut icon"`	4	10
	`<link rel=apple-touch-icon`	0	2
	`<img srcset="url">`	0	3

Table 2.4: Unique number of tracking or advertising domains that make use of one of the potential bypass techniques

there exists a very wide spectrum of advertisers and trackers, some of these may not have been present in our dataset.

2.5.2 Evaluating unknown techniques

In order to evaluate whether any bypass technique was used that was not detected by our framework, we compared the DNS traffic generated by every of the 160,059 visited web pages with the requests that we could detect from each visit. More precisely, we ran every browser instance in a separate Linux namespace and used tcpdump to capture all DNS requests the browser generated. Next, we aggregated all DNS requests that could not be traced back to a captured request and used an aggregated list[12] to mark those directed towards trackers and advertisers. These DNS requests could be indicative of a bypass technique we were previously unaware of.

The preliminary analysis of this data indicated that 4,701 web pages triggered DNS requests for which we did not capture any HTTP request. However, we found that in most cases new resources were still being loaded when we closed the web page (15 seconds after opening it). We re-evaluated these web pages but now allowed the browser 120 seconds to finish loading all resources. This resulted in 865 web pages that triggered a non-corresponding DNS request to a total of 77 different hosts. A manual analysis of these showed that the vast

[12] `https://github.com/notracking/hosts-blocklists`

majority was due to DNS prefetching and the remainder was still caused by requests that were interrupted when closing the browser. These results indicate the completeness of our framework, as we did not find any bypass technique that our framework was unable to detect.

2.6 Discussion

As we have shown in Section 4.5, through our framework, which evaluated several browsers and browser extensions in various configurations, we uncovered numerous instances where an authenticated third-party request could circumvent the imposed restrictions. We found that this unintended behavior can be traced back to several factors, which can be classified as implementation errors, misconfiguration and design flaws. In this section, we discuss which measures can be taken to remedy the discovered circumventions.

2.6.1 Browser implementations

Most of the browsers that we evaluated have built-in support for suppressing cookies of third-party requests. Our results show that only the Gecko-based browsers (Firefox, Cliqz and Tor Browser) manage to do this successfully. Surprisingly, we found that the blocking of third-party cookies feature in Edge had no effect. We believe that this is due to an oversight from the browser developers or a regression bug introduced when new functionality was added.

For the Chromium-based browsers (Google Chrome and Opera), we found that because of the built-in PDF reader, an adversary or tracker can still initiate authenticated requests to third-parties. Because the request is triggered from within the extension, different directives apply, thus allowing cookies to be attached. A possible mitigation for this particular issue could be to disable the functionality of triggering requests from within PDFium. However, this behavior is not unique to PDFium, and other browser extensions may also be exploited in order to send arbitrary third-party requests that bypass imposed cookie policies. As such, we propose that browsers strip cookies from all requests initiated by extensions as a default policy. As this may interfere with the operations of certain extensions, exclusions should be made possible, for instance by defining a list of cookie-enabled domains in the extension manifest.

Next to blocking third-party cookies, we also analyzed the built-in tracking protection for Firefox. Interestingly, we found that for each category of mechanisms that may trigger requests, excluding JavaScript in PDFs, there exists at least one technique that can bypass the built-in tracking protection. A manual analysis of the Firefox source code showed that these bypasses are caused by the retroactive manner in which tracking protection is implemented. More specifically, although the request-validation mechanism is applied in a central location, the validation process is only triggered when a specific flag is set, which requires modifications to every functionality that may trigger requests. While Mozilla is already aware[13] of some of the bypasses we uncovered and is working to mitigate these, we believe that our framework will assist in identifying bypass techniques, even when these are difficult to detect from the millions of lines of code.

2.6.2 Browser extensions

For anti-tracking extensions and ad blockers, it is crucial that *all* requests can be intercepted and blocked or altered. From the results, summarized in Table 2.2 and Table 2.3, it is clear that in the current state this is not the case. In fact, we found that for every analyzed browser extension there exists at least one technique that can be used to circumvent the extension to send an authenticated third-party request. Moreover, we found that the results of the evaluated browser extensions are very disparate, even for extensions that target the same browser. For instance, out of the 15 ad blocking extensions for Google Chrome, at most 3 exhibited a similar behavior.

In part, the disparity of results can be explained by the frequent introduction of new features to browsers, which may affect the WebExtension API or cause unforeseen effects. For instance, support for intercepting WebSockets in browser exceptions was only added years after the feature became available, and after it had actively been exploited to circumvent ad blockers[Chr12a]. Furthermore, AppCache caused one of the parameters of the `onBeforeRequest` API to exhibit a different behavior, which was unexpected by most browser extensions. As a result, requests triggered by AppCache managed to bypass the vast majority of browser extensions. The same change was introduced to Chromium-based browsers when Service Workers were implemented. As a result, most extensions for Chrome and Opera can be circumvented by triggering requests from Service Workers, whereas all extensions Firefox successfully block these third-party requests. This shows that adding new features to a browser may have unforeseen side-effects on the extensions that rely on the provided APIs.

When new browser features are proposed and implemented, test cases that include the new functionality can be added to our framework, allowing browser vendors and extension developers to automatically detect and possibly mitigate

[13] `https://bugzilla.mozilla.org/show_bug.cgi?id=1207775`

unforeseen side-effects. Moreover, because all anti-tracking and ad blocking browser extensions share a common core functionality (namely, intercepting and altering or blocking requests), we propose that all these extensions use a specifically purposed API that is actively maintained. Driven by the high popularity of these browser extensions, this API could be added to the WebExtension API. Alternatively, this API could be offered in the form of an extension module, which of course needs to be maintained and requires all browser extensions to update this module.

2.7 Related work

Policy implementation inconsistencies Multiple studies have shown that browser implementations often exhibit inconsistencies concerning security or privacy policies. Aggarwal et al. [Agg$^+$10] discovered privacy violations for private browsing implementations of modern browsers through both manual and automatic analysis. On top of that, they showed that browser extensions and plug-ins can invalidate the privacy guarantees of private browsing. Schwenk et al. [SNM17] implemented a web application that automatically evaluates the SOP implementation of browsers. In that regard, they showed that browser behaviors differ due to the lack of a formal specification. Singh et al. [Sin$^+$10] pointed out the incoherencies in web browser access control policies. In an effort to help browser vendors find the balance between keeping incoherency-confirming features and the breakage of websites as a consequence of removing them, they developed a measurement system. Jackson and Barth [JB08], too, showed that newly shipped browser features can undermine existing security policies. In particular, they discuss features affected by origin contamination and propose three approaches to prevent vulnerabilities caused by the introduction of these features. Zheng et al. [Zhe$^+$15] question the integrity of cookies by revealing cookie injection vulnerabilities for major sites like those of Google and Bank of America. They showed that implementation inconsistencies in browsers can aggravate these vulnerabilities.

Ad blocking circumventions Iqbal et al. [ISQ17] examined methods that are used to circumvent ad blocking in the wild. They discuss the limitations of anti-adblock filter lists and proposed a machine learning approach to identify ad block bypasses. Storey at al. [Sto$^+$17] also proposed new approaches to ad blocking, countering the existing flaws of traditional ad blocking methods. Their new techniques include recognition of ads trough the use of visual elements, stealth ad blocking and signature-based active ad blocking.

Trackers in the wild Roesner et al. [RKW12] performed an in-depth empirical investigation of third-party trackers. Based on the results of this investigation, they proposed a classification for third-party trackers and developed a client-side application for detecting and classifying trackers. A large-scale crawl was performed by Englehardt and Narayanan [EN16] to gather insights about tracking behaviors in the wild. They found that tracking protection tools such as Ghostery proved effective for blocking undesirable third-parties, except for obscure trackers.

2.8 Conclusion

In this work, we introduce a framework that is able to perform an automated and comprehensive evaluation of cross-site countermeasures and anti-tracking policy implementations. By evaluating 7 browsers and 46 browser extensions, we find that virtually every browser- or extension-enforced policy can be bypassed. We traced back the origin of these bypasses to a variety of different causes. For instance, we found that same-site cookies could still be attached to cross-site requests by levering the prerendering functionality, which did not take these policies correctly into account.

Furthermore, a design flaw in Chromium-based browsers enabled a bypass for both the built-in third-party cookie blocking option and tracking protection provided by extensions. Through JavaScript embedded in PDFs, which are rendered by a browser extension, cookie-bearing `POST` requests can be sent to other domains, regardless of the imposed policies. Additionally, we discovered that not every implementation of the WebExtension API guarantees interception of every request. This makes it impossible for extension developers to be completely thorough in blocking or modifying undesirable requests.

Overall, we found that browser implementations exhibited a highly inconsistent behavior with regard to enforcing policies on third-party requests, resulting in a high number of bypasses. This demonstrates the need for browsers, which continuously add new features, to be thoroughly evaluated.

The results of this research suggest that policy implementations are prone to inconsistencies. That is why we think that, as future research, the framework could be extended to evaluate other policy implementations (e.g. LocalStorage API [Mozh], Content Security Policy [WS23]). In addition to that, the evaluation of mobile browsers could also be an interesting direction. This includes the mobile counterparts of major browsers for iOS and Android, but also mobile exclusives like Firefox Focus [Mozl].

3

A Bug's Life

Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs

Nobody really cares about killing insects.
Even the animal rights people don't care.

– Jerry Seinfeld [BSA95]
(Seinfeld, 1995)

This chapter was previously published as:

> G. Franken, T. Van Goethem, L. Desmet, and W. Joosen. "A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3673–3690. ISBN: 978-1-939133-37-3

Chapter 2 and numerous complementing studies, have shed light on significant implementation deficiencies of security and privacy policies. While these studies have played a vital role in ongoing efforts to secure the Web, the origins of these persistent issues have only been supported by anecdotal evidence. In this chapter, we present the first comprehensive study that provides insights into the root causes of implementational shortcomings of a browser security policy, CSP, substantiated by robust empirical data. Moreover, by refraining from depending on the information provided by bug reports, we were able to uncover inconsistencies in bug handling procedures, including the premature public disclosure of three security bugs.

To facilitate this project, we developed the BugHog framework, specifically tailored for the precise identification of bug introductions and fixes by code revisions, through the dynamic evaluation of bug proofs of concept (PoCs). A dynamic analysis of CSPs complete development history necessitates the execution of browser binaries of over a decade old. This task is often impeded by compatibility issues between the browser binaries and the outdated, deprecated or unsupported libraries in de package manager of the host OS. To overcome these challenges, we containerized the entire framework using Docker, which allowed us to execute any binary within a compatible environment in which all dependencies are taken care of. This sets BugHog apart from all other bisection tools, where dependency management is considered the user's responsibility. Another notable benefit of this approach is that the framework is compatible with any OS that supports Docker.

The BugHog framework has been released as open-source, with a commitment to maintain it for ongoing research purposes.[1] Other than for the historical assessments of various policies beyond CSP, it can also be employed to pinpoint the exact lifecycle of complex multi-stage exploits, thanks to its dynamic evaluation approach. As such, we argue that BugHog poses a valuable tool in the hands of both security researchers and browser developers. Even when an analysis requires lifecycle identifications for a substantial collection of bugs, this can be efficiently managed because of BugHog's built-in concurrency capability.

We reached out to all affected browser vendors to inform them of four publicly disclosed security bugs that were still affecting their most recent release version. As a result of these reports, three of the vulnerabilities were successfully addressed, while one was determined not to be a bug.[2] Additionally, we have proposed concrete resolutions and recommendations aimed at enhancing the current bug handling processes.

This work was awarded with a *Distinguished Paper Award.*

[1] `https://github.com/DistriNet/BugHog`

[2] This bug was initially reported for Chromium, where it was acknowledged but not yet resolved. When we discovered that it also impacted Safari and reported it, it was not considered a bug by WebKit developers.

3.1 Introduction

Since their inception, web browsers have grown to become immense applications comprising tens of millions of code lines, introducing new features with virtually every major release. To keep up this pace, over 100 code revisions are applied to their code base every single day [RMJ15], ranging from bug fixes to new feature introductions. Although this pushes the Web forward in many great ways, meanwhile, browser vendors need to make a continuous effort to guard against newly discovered attacks and bypasses for both established and new security policies.

Unfortunately, numerous CVE reports and an extensive body of research have previously exposed countless vulnerabilities facilitated by flawed browser security policy implementations. More specifically, various shortcomings of essential security policies such as the CSP [HMN15], Same-Origin Policy [SNM17], SameSite cookie policy [FVJ18] and access control policies [Sin+10] have been discovered and exhibited. In several cases this makes the security policy, which websites often rely on to safeguard their users, obsolete until a mitigation is in place. Furthermore, vulnerabilities are often caused by inconsistent implementations among browsers as well [Cal+20; Luo+19; Sie+22; Wi+23]. However, the granularity of these studies halts at the level of browser release versions, disregarding information related to the individual revisions that cause or fix a bug.

To close this research gap, we performed a longitudinal study on the introducing and fixing source code revisions of bug lifecycles for CSP, one of the most longstanding and important security policies of the Web. Given both the importance of CSP and its extensive implementational lifetime of close to a decade, we take advantage of both the large number of reported bugs and the many code changes that have affected it. By collecting 86 publicly disclosed bug reports that entail the subversion of correct CSP enforcement, we identified 75 unique bugs for which we replicated a PoC, that was then used to construct a dynamic evaluation on reproducibility. Subsequently, leveraging our automated framework, BugHog, we identified the complete bug lifecycles in the open-source Chromium and Firefox browsers. As such, having evaluated over 100,000 revision binaries, we were able to pinpoint 46 unique revisions that introduce a bug, 71 that fix a bug and six that do both. To the best of our knowledge, this is the first comprehensive bug lifecycle analysis considering individual revisions, based on the dynamic analysis of a browser policy implementation.

Our analysis shows that half of the CSP bugs were already present at the time of the policy's introduction, among which a severe Chromium vulnerability with subsequent bug bounty of $5000 (CVE-2021-30531) [Chr20a]. After undermining the effectiveness of CSP for a period of more than eight years, the issue was ultimately fixed in 2021, highlighting how even severe bugs can stay under the radar for extensive periods of time. Besides these so-called foundational bugs, a large part was introduced by revisions intended to fix other CSP bugs or redesigns of the underlying code structures, demonstrating the fragile nature of CSP-related source code.

In our analysis we employed a dynamic evaluation, in contrast to static evaluation based on bug reporting information in prior work [Ale$^+$22; BBB16; Bra$^+$22b; CMN15; FAW13]. This allowed us to perform a cross-browser evaluation of all reported bugs, reproducing bugs reported for one browser throughout the revision history of the other. This way we found 14 shared bugs, among which seven could be completely avoided or reduced in lifetime if bugs were more effectively shared between vendors. Furthermore, we could reproduce four of the collected bugs in Safari's most recent version. Additionally, we identify several other bug handling flaws such as inconsistent revision linking and report labeling. More severely, our evaluation detected three bugs that were labeled as fixed and eventually publicly disclosed while the fixing revision was not effective, leaving the browser vulnerable unbeknownst to the developers. Two of these bugs remained public and unfixed for at least a year, and one was only fixed after we reported the issue.

We make the following contributions:

- We developed BugHog, a framework for pinpointing introductions and fixes of browser security policy bugs at the level of individual code revisions. This framework is released as open-source upon publication of this work, and can be extended to facilitate the evaluation of other security policy implementations as well.
- To the best of our knowledge, we performed the first systematic lifecycle analysis based on dynamic evaluations over the full history of a browser security policy. As such, we analyzed 75 reported CSP bugs for Chromium and Firefox, covering 123 unique code revisions that caused an introduction or fix.
- Based on our thorough analysis, we diagnosed several flaws regarding security policy implementations and bug handling practices, causing a needless escalation of security implications.
- Finally, we propose several remedies to these issues, such as more rigorous bug sharing between vendors and more stringent bug handling practices.

3.2 Background

In this section, we explain the foundational concepts of current browser development practices and CSP.

3.2.1 Web browser development

Web browser vendors utilize various development practices, among which version control and regression testing.

Version control

Web browsers, being code development projects consisting of tens of millions of code lines, are developed, managed and maintained by leveraging a version control system (VCS). Although Chromium and Firefox employ different VCSs (i.e. Git[3] and Mercurial[4], respectively), their underlying functionality is very similar. However, the employed version control strategies of the two projects slightly differ.

Chromium developers apply a trunk-based development pattern to a single repository (Figure 3.1), where each developer directly commits to a so-called trunk branch (e.g. instead of using feature branches) [Goo22; Tru]. Source code is prepared for building the release binary (e.g. disabling experimental features) on a so-called release branch forked at regular intervals from the trunk. Only in special cases, like urgent security fixes, a revision on the trunk is cherry-picked and merged onto such a release branch [Boo15].

Firefox manages four separate repositories, each associated with a different release channel (Figure 3.2). All code revisions are by default applied to the `mozilla-central` repository (i.e. nightly), where all revisions are periodically imported into `mozilla-aurora`. This process is repeated for each repository, such that all revisions will be consecutively imported to the `mozilla-beta` and eventually `mozilla-release` repositories. In particular cases, developers might decide to uplift a feature or a patch to mitigate a severe vulnerability on a more stable channel[Mozb; Mozc].

In conclusion, all revisions are eventually landed on a single branch or repository, being Chromium's trunk or Firefox's `mozilla-release` repository.

[3] `https://git-scm.com`

[4] `https://www.mercurial-scm.org/`

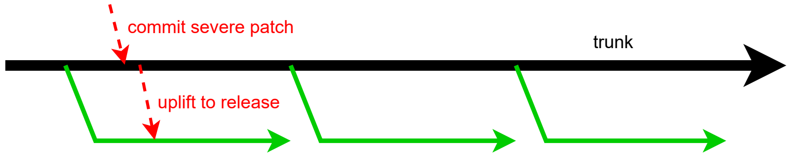


Figure 3.1: Chromium's development practice where all applied revisions are periodically forked into a release branch.

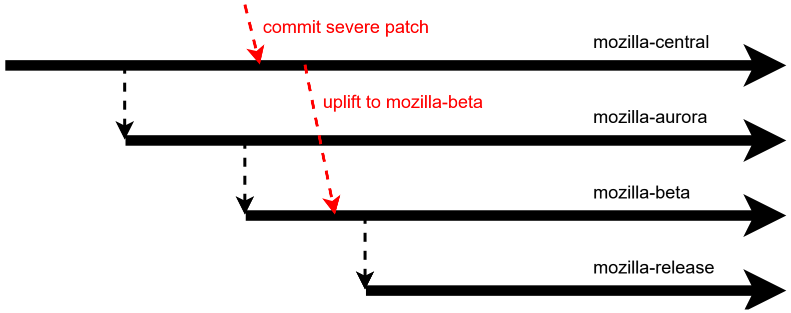


Figure 3.2: Firefox's development practice where all applied revisions are periodically imported to a more stable repository.

Regression testing

In addition to their own specific test suites, Chromium [Goob] and Firefox [Firb] share a common cross-browser test suite called Web Platform Tests (WPT) since September 2014 for identifying potential regressions (i.e. previously fixed issues that have been inadvertently reintroduced) [Bug13; Chr14; web]. Both browser vendors depend on and contribute to the project, which serves as a comprehensive set of checks to confirm compliance to established web standards, including security prerequisites. According to both vendors' contribution policies, each revision or patch should successfully complete all regression tests before it can be landed [Chrd; Fira].

Contributors to the Chromium project are advised to use one of two procedures to track down the introduction or fix of a regression. One option is utilizing their `bisect-builds.py` script which automates a binary search over a bounded revision range of publicly hosted revision binaries, though the user is required to manually check for each binary whether the bug is reproduced [Chrb]. The other recommended practice utilizes `git bisect`, which is able to discern whether a revision reproduces the targeted issue in an automated manner if provided with the appropriate script and test files [Goob]. Nonetheless, this evaluation requires checking out and building each revision that is to be evaluated, making it resource-intensive.

Firefox provides a tool for bisecting regressions as well; similar to Chromium's script, their so-called `autobisect` tool relies on publicly available revision binaries [Mozn]. However, this bisection tool is fully automated since the script can autonomously distinguish between test case results.

In Section 3.3.2, we discuss the advantages and limitations of each approach, compared to BugHog.

3.2.2 Content Security Policy

CSP version 1 was originally proposed as an in-depth defense mechanism against content injection attacks such as XSS [Mozf; SB12; SSM10]. Websites can deploy a policy by providing the `Content-Security-Policy` header or `<meta>` tag in their response, and consequently the browser will enforce the defined policy client-side.[5]

CSP provides developers with several directives for enabling different blocking rules over different resource types. For instance, the policy defined below demonstrates this content blocking use case, and will only permit the browser to load scripts from `third-party.com`, while all other resources must originate from the current website (indicated by `self`). In this study, we distinguish between two subclasses of the content control use case: active content control (i.e. script blocking) and non-active content control (i.e. frames).

```
default-src 'self'; script-src third-party.com
```

Subsequent versions of CSP, specifically CSP 2 and CSP 3, introduced additional functionality to the policy specification, among which new directives and keywords [WBV16; WS23]. For instance, the `nonce` keyword provides developers with the ability to allow inline script inclusion while simultaneously safeguarding against script injections through the use of a secret nonce.

Moreover, new use cases have been introduced, such as the `upgrade-insecure-requests` directive, which facilitates the automatic upgrading of all requests made over unencrypted channels (e.g. HTTP) to secure channels (e.g. HyperText Transfer Protocol Secure (HTTPS)), thereby enforcing Transport Layer Security

[5] Note that before the actual introduction of CSP, several browsers already employed an experimental implementation under the `X-Content-Security-Policy` and `X-WebKit-CSP` headers.

(TLS) encryption. Another important use case is framing control, which is facilitated by the `frame-ancestors` directive. This directive allows developers to specify which origins are permitted to embed the website within an `iframe` to prevent clickjacking attacks. Lastly, the `referrer` directive grants developers control over the `Referer` request header when users navigate from the current website. Previous studies have already highlighted a similar differentiation between CSP use cases [Rot+; Wei+16].

The inheritance of a policy between browsing contexts presents significant challenges for CSP designers and implementers. When creating a new browsing context, such as embedding an `iframe` or opening a new window, the enforced policy is inherited from the opener browsing context in specific situations. For example, new browsing contexts with a `blob:` or `data:` URL should inherit the employed policy from the opener context. However, exercising excessively lax policy inheritance can create opportunities for CSP subversion, whereas overly stringent inheritance may result in cross-site leaks (XS-Leaks) [Kni+21]. XS-Leaks allow malicious actors to exploit CSP to extract user state information from cross-site services, leading to potential privacy breaches. For instance, a malicious website could employ a CSP policy that permits navigation to a benign website, but restricts access to the landing page to which logged-in users are redirected. This way, the adversary can infer the presence of an active session on the benign site through CSP's violation report, obtained through the `report-to` CSP directive.

Figure 3.3 depicts an overview of all CSP use cases and bug classes. We refer to Mozilla Developer Network for a comprehensive overview of all CSP directives and keywords [Mozf].

3.3 Methodology

In this section, we cover all stages of our research, including the design of BugHog, utilized to identify bug lifecycles.

3.3.1 Bug collection and reproduction

All bugs were collected from Chromium's[6] and Firefox's[7] public bug tracking platforms. For both platforms, bug reports are by default confidential until a fix has been widely deployed [Chrc; Moza]. Unfortunately, we did not receive a

[6] `https://bugs.chromium.org/`

[7] `https://bugzilla.mozilla.org/`

response from the WebKit Security team regarding our inquiry for access to fixed WebKit bugs.

To ensure comprehensiveness and enable an evaluation of applied bug report labels, we adopted broad search criteria. For instance, we did not rely on any specific CSP labels, but instead used keywords that are matched against the content of bug reports. This approach ensures that any potential oversights in developer labeling do not impact the integrity of our dataset. Subsequently, we filtered out all bugs that were not related to CSP, by manually inspecting the included bug description. In total, we collected 86 bug reports; 58 for Chromium and 28 for Firefox, a ratio which is similar to that of prior studies [FAW13]. In case two or more bug reports described the same bug, we considered it as one bug. As such, we collected 75 unique bugs in total. We refer to Appendices B.1 and B.2 for more details about the collection and filtering process. Figure 3.3 shows all CSP bug classes that were identified in our dataset, along with the number of associated bugs.

Unfortunately, there is currently no standardized format for documenting bugs or vulnerabilities, and not all reports include practical tests that demonstrate the issue effectively. Consequently, we were required to manually develop and incorporate bug PoCs into BugHog. In the best case, the report included the necessary HTML, CSS, and JavaScript code, resulting in a minimal effort to recreate the PoC. In the worst case, we had to rely on the textual description provided by the reporter and manually construct the PoC ourselves.

To avoid discrepancies, PoCs were integrated into BugHog with minimal modifications. However, the original PoC might employ a more recent web mechanism that is not supported in older revisions. In such cases, we emulated the desired web mechanism using its predecessor(s), such as converting JavaScript to ECMAScript 5, whenever feasible. The validity of each recreated PoC was verified through manual testing, which involved running the PoC for a binary affected by the bug and another that was not affected.

3.3.2 Automated lifecycle identification

BugHog is designed to evaluate an extensive range of individual revisions of the Chromium and Firefox browsers, in order to identify a bug's complete lifecycle, from bug introduction to mitigation. In the following sections, we discuss the three main tasks of BugHog: selecting the appropriate revisions to evaluate, collecting the selected revision binaries and performing a dynamic evaluation on

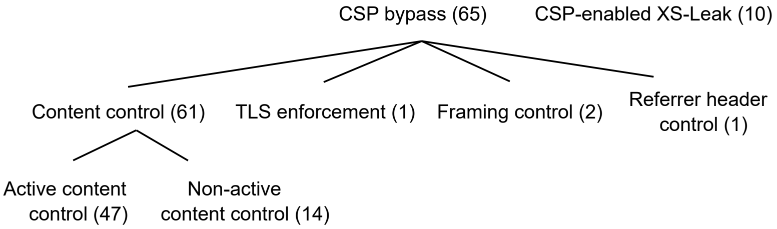


Figure 3.3: Overview of all CSP use cases and bug classes with the respective bug frequency in our dataset.

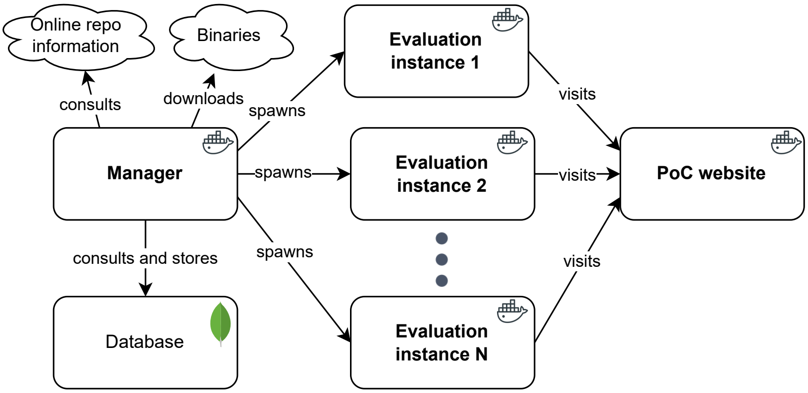


Figure 3.4: High-level overview of BugHog. The Docker logo indicates that a component is run inside its own Docker container.

the collected binaries. We have made the source code of our framework publicly available.[8]

Overview

Figure 3.4 shows a high-level overview of BugHog covering three main components (indicated in bold), each of which runs inside its own Docker container. All Docker images are built on top of the Debian GNU/Linux 10 (buster) base image.

[8] https://github.com/DistriNet/BugHog

Manager. This component is responsible for managing the evaluation instance containers used for dynamic binary evaluation. Its core tasks consist of selecting the appropriate revision binary to evaluate next, to download this selected binary and to spawn a helper container to perform the actual evaluation.

Evaluation instance. This component performs the actual dynamic evaluation by instructing the browser binary to visit one or more PoC web pages. Since this Docker image needs to support a very wide range of browser versions (Chromium v25 - v109, Firefox v23 - v109), a large number of dependencies has to be fulfilled in order to execute all required binaries. Even though several older (deprecated) dependencies were not available through a package manager, we still managed to fulfill this requirement through manual installation.

PoC website. This component hosts an Nginx and Flask web server incorporating all bug PoCs. Each PoC is integrated by providing web page source code and the order in which these web pages are to be visited to reproduce the exploit. Various configuration options are supported, such as defining values of the response status and headers. During the evaluation, all communication between the browser binary and the local web server is recorded through a proxy such that the outcome of the evaluation can be discerned; whether the bug can be reproduced or not.

Apart from the three main components, BugHog utilizes MongoDB for storing the results of each revision evaluation, enabling subsequent querying of the data. Finally, we use publicly available revision binaries hosted by each vendor for our dynamic evaluation and scrape online repository information to traverse over revisions.[9][10] We only require access to browser source code for building binaries if the available online binaries are insufficient to pinpoint an exact revision.

Collecting revision binaries

Revision binaries are obtained by either downloading them or building them from source.

Downloading. Besides hosting binaries of release versions, both Chromium[11] and Firefox[12] host additional binaries based on certain source code revisions as well. From these collections, it appears that Chromium builds a binary multiple

[9] https://chromium.googlesource.com/

[10] https://hg.mozilla.org/

[11] https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html

[12] https://ftp.mozilla.org/pub/firefox/

times per hour, while Firefox seems to build a binary every 12 hours. If a to-be-evaluated revision binary is not available, BugHog will download the one closest available.

Building. If vendor-hosted revision binaries are not sufficient to infer the introducing or fixing revision, we build binaries from source. This way, we were able to obtain all binaries necessary for the evaluation of all collected bugs.

Revision evaluation selection and order

The scope of this work covers CSP in the form of the currently employed `Content-Security-Policy` header or `<meta>` tag. We did not extend our analysis to experimental precursors such as `X-WebKit-CSP` and `X-Content-Security-Policy` [Mozf], since these were not shipped as finished policy implementations, and as such, are not assumed to be bugless.

CSP 1.0 was introduced by revision 165317 [Chr12b] and revision 144546 [Fir13] for Chromium and Firefox respectively. As such, to identify all plausible bug lifecycles we are required to evaluate the revision range between these revisions and the most recent browser version (version 109 for both Chromium and Firefox).

Our search strategy is composed of two phases:

General sweep. First, we conduct a broad survey over the entire range, and to be comprehensive, this phase covers at least multiple revisions per release version. This way, we can already identify confined ranges in which a bug was introduced or fixed. Furthermore, in contrast to the other bisection tools, this allows us to possibly find additional introducing and fixing revision couples, required for a complete lifecycle analysis.

Precise pinpointing. In the second phase, we use binary search to determine the exact revision that introduced or fixed a bug. In several cases, the publicly provided revision binaries are not sufficient to pinpoint this revision. However, if the revision range has been sufficiently narrowed down, it is straightforward to manually identify this revision. In the other case, we build new revision binaries and repeat the evaluation within the refined range. While starting the build script and restarting the evaluation are manual tasks, this could be automated in the future. For every pinpointed revision, we ensured its validity by verifying logical constraints (e.g. introducing revisions are strictly older than their associated bug report).

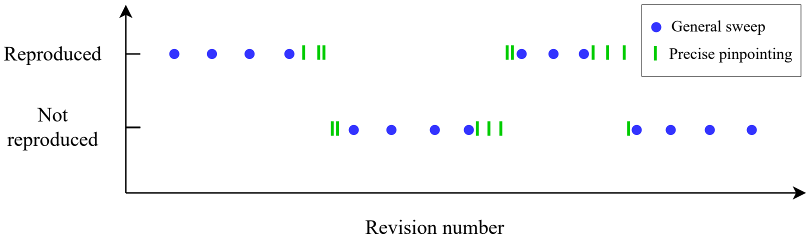


Figure 3.5: Example of the revision evaluation process.

Figure 3.5 depicts an example of evaluation output over a predefined range. Here, the general sweep (blue dots) reveals that the bug is reproducible in the first subrange of revisions, and that although eventually fixed, the bug is reintroduced at a later point. In the second phase, precise pinpointing (green rods) reveals more accurately where the introductions and fixes occurred.

Unfortunately, the Chromium repository does not include JavaScript engine source code, nor web engine source code in earlier repository versions. Because these are hosted in separate repositories, our framework would merely pinpoint affected revisions as rollouts (i.e. a set of engine revisions). In order to identify individual engine revisions, we would have to build a single browser binary multiple times, sequentially changing the embedded engine revision. This was not deemed feasible, and as such, pinpointing within engine rollouts is covered manually.

Dynamic evaluation

Although browser automation libraries like Selenium are prevalent, they often do not provide support for outdated, older browser versions. Fortunately, our use of the command line interface (CLI) for instructing browsers gives us the advantage of evaluating any browser binary.

To ensure a clean and consistent environment for each experiment, we create and select a fresh browser profile using the appropriate CLI flags. The selected profile is maintained during the experiment to simulate visits using the same browser instance. This approach also enables us to propagate desired settings, such as setting a proxy, or to disable interfering features, such as Firefox's built-in tracking protection.

Functionality	**git bisect**	**C:bisect-builds.py**	**F:autobisect**	**BugHog**
Automated	●	○	●	●
No checkout building	○	●	●	●
Concurrency	○	○	○	●
Dependency handling	○	○	○	●
PoC user interaction	○	●	○	○

Table 3.1: Overview of bisection frameworks in terms of supported functionality.

Advantages and limitations

In Section 3.2.1, we discussed various tools available for bisecting bugs through dynamic evaluation. An overview of the supported functionality for each existing bisection tool, including BugHog, is presented in Table 3.1.

Among the tools mentioned, only Chromium's `bisect-builds.py` script requires manual input from the developer to determine if a bug is reproduced. While this allows for the evaluation of bugs that involve user interaction, it also significantly increases the evaluation time. In contrast, the other tools automate the process entirely.

However, BugHog offers several advantages compared to the other tools. One notable feature is the ability to run evaluations concurrently, which accelerates the revision pinpointing process. This feature proves particularly valuable when conducting a comprehensive historical analysis of bug reproducibility or evaluating a large number of bugs, as was necessary for our study.

Additionally, BugHog leverages containers to manage external dependencies, enabling the execution of even older browser versions. As such, it supports the evaluation of Chromium and Firefox dating back to 2012, while the other tools can only handle binaries up until 2019 before running into dependency issues (with the exception of Firefox binaries for Windows, which have fewer external dependencies).

Finally, BugHog is developed within Linux containers, making it compatible with any operating system that supports Docker. This cross-platform capability further enhances the accessibility and usability of BugHog.

3.3.3 Analysis

To conduct our analysis, we used an automated scraper to collect information from bug reports and code revisions obtained from the aforementioned public bug tracking platforms. Additionally, to enhance the depth of our analysis,

Group	Label	Regression
Policy introduction	Introduce CSP	○
Fix	Fix affected CSP bug	●
	Fix other CSP bug	●
	Fix unrelated security bug	●
	Fix non-security bug	●
Enable feature	Enable affected CSP feature	○
	Enable CSP feature	○
	Enable security feature	○
	Enable non-security feature	○
Update feature	Update CSP feature	●
	Update security feature	●
	Update non-security feature	●
Design choice	Design revision of CSP	●
	Design revision of other security policy	●
	Non-security design revision	●

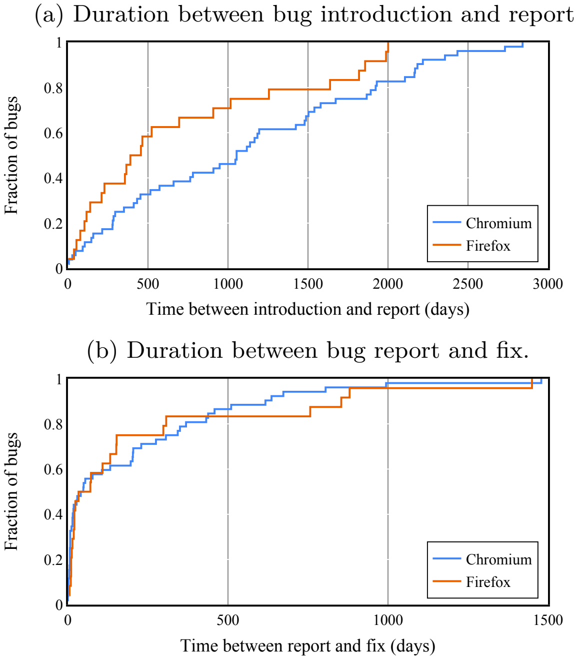
Table 3.2: Overview of all revision intentions, where the *Regression* column indicates whether the associated bug introducing revision would be considered a regression.

we conducted manual inspections of relevant sections of the source code. The visualizations and statistics used in Section 4.5 were generated by automated scripts, which can be re-used for other bug studies.

To better understand the purpose of code changes, each revision was manually annotated with a label indicating its intended purpose. All labels are listed in Table 3.2, where the last column indicates whether a label is considered a regression if its intent is linked to a bug introducing revision. The labeling was done by two experts and evaluated with a Cohen's Kappa agreement score of 0.81, with any remaining disagreements resolved through discussion. We refer to Appendix B.3 for a detailed description of each label, and further details on the labeling methodology.

3.4 Results

In this section, we provide a detailed analysis that relies on the diverse metadata linked to introducing and fixing revisions, as well as bug reports.


(a) Duration between bug introduction and report.

(b) Duration between bug report and fix.

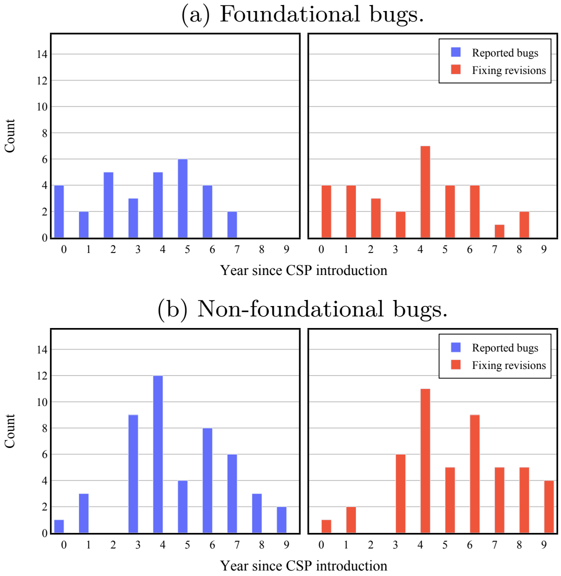
Figure 3.6: CDFs of the duration between bug introduction and report, and report and fix.

3.4.1 Bug lifecycle

To shed light on the duration that CSP bugs stay undiscovered, we calculated the cumulative distribution function (CDF) of the duration between the introduction and reporting of the collected CSP bugs for each browser (Figure 3.6a). Interestingly, Chromium bugs seem to live longer before they are reported, compared to Firefox. For Chromium the median duration between the introduction and report is 2.9 years, whereas this is 1.2 years for Firefox. Note that Chromium has enabled CSP support since November 2012, whereas Firefox only enabled it since May 2013.

Figure 3.6b shows the CDF of the duration between the first report and its subsequent effective fix.[13] For Chromium the median duration is 44 days, which is slightly lower than the 52 days it takes Firefox to land a fix.

[13] Two bugs have been excluded because they were not fixed at the time of writing.


(a) Foundational bugs.

(b) Non-foundational bugs.

Figure 3.7: Number of bugs and associated fixing revisions for each year since the introduction of CSP.

When we look at the number of bug reports and fixing revisions for *foundational* bugs, i.e. bugs that were present since the introduction of CSP, over the ten years since CSP was introduced, we observe a downward trend (Figure 3.7a). Still, it should be noted that part of this downward trend in the last three years can be attributed to the fact that not all discovered bugs are publicly disclosed yet.

However, inspecting the same bar chart for *non-foundational* bugs, i.e. bugs introduced after the introduction of CSP, a far less apparent downward trend can be found (Figure 3.7b). Moreover, the last three years show a consistently higher number of both reports and fixing revisions in comparison to our dataset of foundational bugs. Indeed, when inspecting the number of introducing revisions for non-foundational bugs (Figure 3.8), it strengthens the conclusion that the number of non-foundational bugs does not necessarily decrease. Note that we did not find any introducing revisions for year eight and nine, since these reports are most likely not public yet.

These findings suggest that CSP as a policy has not yet reached maturity; there is no indication of a decrease in the amount of new bug introductions. Upon closer examination, the most prevalent root causes of non-foundational bug

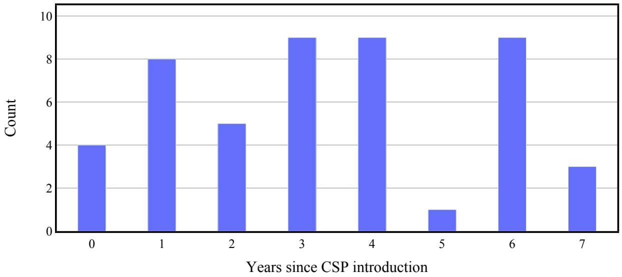


Figure 3.8: Number of non-foundational bug introductions for each year since the introduction of CSP.

introductions appear to be the fixing of (security) bugs (27.2%) and adding new (security) functionality (48%). Interestingly, the fixing of CSP security bugs in particular caused 21% of the non-foundational bugs.

Presumably this conveys that web browser development follows trends similar to those of more general software development. As suggested by previous work, software matures regarding foundational bugs over time [OS06], while this is not necessarily the case regarding non-foundational bugs [Ale+20; Res05].

> ***Finding 1.*** Following the trend of general software development, foundational bugs affecting CSP are most likely to diminish over time. In contrast, non-foundational bugs, which typically originate with the introduction of new functionality or as a byproduct of mitigating other bugs, are likely to remain occurring.

Our cross-browser evaluation demonstrates that of all 75 unique bugs, 14 (19%) are reproducible in both Chromium and Firefox at some point in their development history. While in general both browsers provide very similar functionality, they mainly face unique bugs throughout their history, which could be attributed to the difference in architecture and implementational flaws.

The lifecycles of shared bugs are shown in the Gantt chart of Figure 3.9, where the presence of a bug identifier on the y-axis indicates whether a bug report was found for the associated browser. The vertical lines indicate at what time the associated bug report was filed, if present. In eight cases, a specific bug could be reproduced in both browsers whereas there was only a report made to a single browser vendor. In these cases, even the revision in which the bug was fixed did not refer to a report describing the bug, so presumably no issue was ever filed and the bug was introduced and fixed unbeknownst to the developers.

We also find that in seven cases the bug was reported for one browser during or before the vulnerable period of the other browser. Although both browsers share WPT as a common test suite, this result demonstrates a remaining lack of effective threat vector sharing between the two vendors.

To further explore the prevalence of cross-browser bugs, we examined whether any of these bugs were reproducible in WebKit by evaluating the most recent Safari release (16.2). Here, four Chromium bugs could be reproduced, all of which had been publicly disclosed for over a year at the time of writing, with the oldest disclosure dating from May 2017. Only two of these bug reports linked to a revision in which regression tests were added to WPT, and one other bug was not fixed yet at the time of writing. This further supports our finding that the current level of threat vector sharing between browser vendors is unsatisfactory. All bugs have been responsibly disclosed, of which three have been fixed and one is not considered a bug by Safari developers. Although Chromium developers consider the latter issue to be of medium severity, it remains unresolved in their codebase as well.

Finding 2. While browsers have distinct architectures, and thus face unique bugs, a considerable number of CSP bugs occur in multiple browsers. We argue that a more effective threat vector sharing strategy can reduce bug lifetimes or even completely avert them.

We observe that of all shared bugs depicted in Figure 3.9, eight are foundational in Chromium, in contrast to only four in Firefox. This indicates that Firefox's introduction of CSP was more comprehensive and sound. However, five of Chromium's foundational bugs eventually appeared as regressions in Firefox. This highlights that even if foundational bugs are avoided through a comprehensive policy introduction, these particular bugs are still prone to being introduced as a future regression. Moreover, this underscores the importance of including *all* shared security tests from all other browser vendors, even if initially not affected by a bug.

3.4.2 Bug introduction

In this section, we study the root causes of bugs by analyzing the introducing revisions. We examine the revisions from three angles: their intent, their context (i.e., which aspect of CSP is impacted and how it is bypassed) and their associated source code sections.

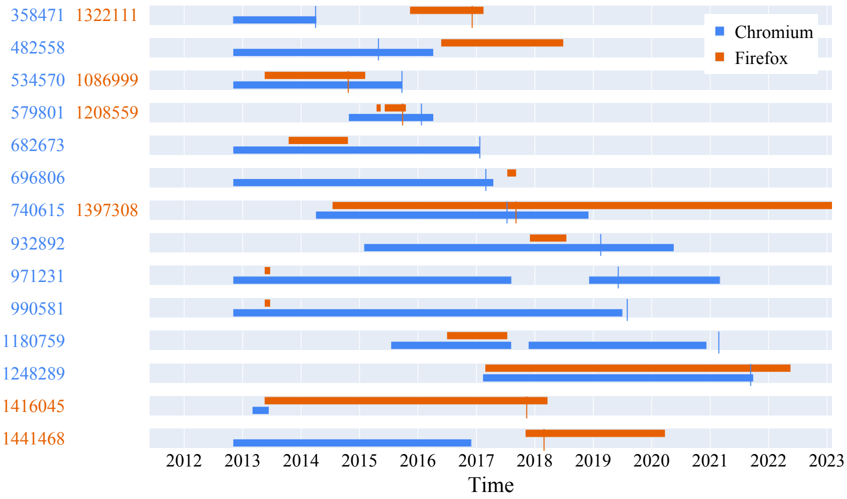


Figure 3.9: Gantt chart of cross-browser bug lifecycles that affected both Chromium and Firefox.

Figure 3.10 shows the intention prevalence for revisions that introduced a CSP bug. All revisions introduced a bug a total of 97 times, as some revisions introduced multiple bugs and several bugs regressed after a fix. For both Chromium and Firefox most bugs (23 and 13 respectively) were introduced with the shipping of CSP, indicating that the implementation of CSP at the time was not sufficiently comprehensive. Similarly, enabling a new CSP feature (e.g. shipping a new directive) allowed for various bypasses as well (eight for Chromium, five for Firefox), reinforcing the idea that the introduction or extension of the policy is prone to lack of comprehensiveness.

Finding 3. Approximately half of the bugs affecting CSP reported in the ten years following its introduction were present since the initial shipping of the security feature.

Revision intent

Of all non-foundational revisions, seven revisions introduced multiple bugs; six revisions introduced two bugs and one even introduced six. For all but one, modifications to CSP configuration logic (i.e. policy parsing, feature or directive introduction) were identified as the bug root cause, indicating that

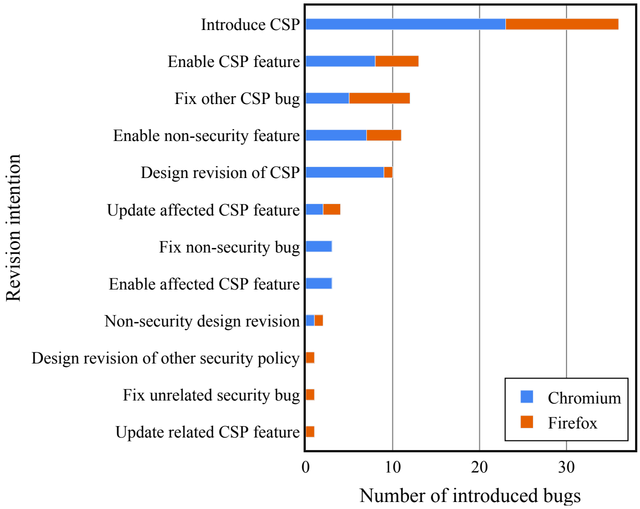


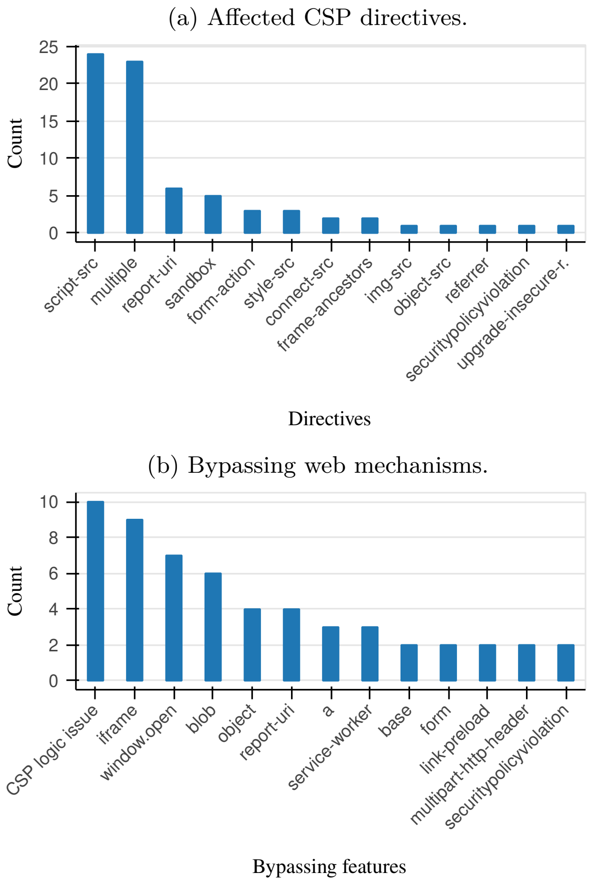
Figure 3.10: Intentions of revisions that introduced a CSP bug.

updating CSP configuration logic bears more risk to introducing multiple bugs in comparison to modifying enforcement logic. The revision causing six bugs was applied in an attempt to mitigate several bugs through a design revision of CSP. Here, an inadequate CSP inheritance upon navigating to a new context was reported, which could be abused by an attacker to inject a script.[14] Although the revision effectively mitigated the reported bugs, it introduced new bugs where an inadequate inheritance of CSP was again the root cause.

Similarly, we find that twelve bugs, five for Chromium and seven for Firefox, were introduced as a result of fixing a CSP-related bug. This clearly indicates that even the smallest changes that are made to the enforcement of CSP may cause other, independent issues to arise.

Finding 4. The implementation of CSP is very brittle. Especially changes to the core functionality and configuration logic are likely to cause new bugs. Hence, fixes for existing policy bugs are also likely to introduce new issues.

[14] The PoCs all leveraged navigation to an attacker-constructed blob Universal Resource Identifier (URI) and navigation to a new window where afterwards `window.document.write()` was used to inject a script.


(a) Affected CSP directives.

(b) Bypassing web mechanisms.

Figure 3.11: Distribution of affected CSP directives and most prevalent bypassing web mechanisms.

Revision context

For every bug, we analyzed which CSP functionality was affected by the bug (Figure 3.11a) and what web mechanism or feature facilitated the bypass (Figure 3.11b). Since we cannot assume that PoCs list all bypassed directives, we reproduced multiple versions of each unique bug to find out whether a single specific `src` directive or multiple `src` directives were affected. Interestingly, 24 bugs (32%) bypass only the `script-src` directive, and 23 (31%) bypass more than one `src` directive. Here, CSP's essential and critical responsibility of blocking inline scripts and `eval` were bypassed in five and four instances respectively. We argue that `script-src`'s complexity, given the various keywords that it supports (e.g. `nonce`, `strict-dynamic`, `sandbox`), contributes to its error-prone implementation.

Most bypasses are caused by CSP logic issues that are not directly related to a specific web mechanism (e.g. incorrect policy parsing, logic errors). Interestingly, the `iframe`, `window.open`, `blob` and `object` mechanisms are most prevalent and account for 9 (12%), 7 (9%), 6 (8%) and 4 (5%) bypasses respectively. These mechanisms are all related to creating and navigating to new browsing contexts. Deeper analysis shows that 14 of 23 bugs affecting multiple CSP directives were caused due to a bypass related to navigation. This shows that the complexity of handling policy inheritance between multiple browsing contexts not only induces error-prone code, but its issues affect a larger surface area of the policy language as well.

> ***Finding 5.*** The complex implementation of policy inheritance between browsing contexts is not only prone to various bugs, but also increases the likelihood of errors serving as bypasses for multiple directives.

Source code

By examining the source code in more detail, we notice that the enforcement of CSP for content control (Figure 3.3) is less centralized compared to other use cases. In this case, specific sections of code dedicated to different mechanisms are responsible for performing CSP checks. For example, mechanisms such as `<base>`, `<a>`'s `ping` attribute, favicon fetching, form submission, and Workers require conditional CSP checks in addition to the general resource fetching check. Conversely, the functionality scope for TLS enforcement, framing control, and referrer handling is much narrower, resulting in fewer detached CSP checks. For instance, we only found separate checks for form submission and WebSockets to ensure CSP compliance in both browsers regarding TLS enforcement.

The bugs resulting from missing enforcement in CSP tend to have relatively simple PoCs, wherein a single bypassing web mechanism is sufficient for the exploit. This trend is particularly notable in cases involving non-active content control, where nearly half of the bugs are attributed to missing enforcement and are generally classified as low severity. Within the active content control mitigations, `nonce`, `sandbox`, and inline scripts are most affected, where severity is typically somewhat higher.

Finding 6. The fragmented enforcement logic of CSP increases the likelihood of oversights, which could even lead to straightforward policy bypasses.

As depicted in Figure 3.11a, the majority of CSP bugs circumvent the `script-src` directive, which is a crucial component of active content control. Here, 12 out of 23 bugs related to `script-src` origin enforcement are caused by inheritance issues. Keywords of `script-src` such as `strict-dynamic`, `nonce`, and `sandbox` are less affected. Bypass techniques often exploit multiple browsing contexts, resulting in more intricate exploits with more impact compared to those arising from enforcement oversights.

Remarkably, nearly all inheritance-related issues affected Chromium (20 out of 22), while only four affected Firefox.[15] This disparity implies that Firefox's inheritance logic has been considerably more robust compared to that of Chromium. However, Chromium developers undertook a considerable effort centralizing inheritance logic by incorporating CSP in the Policy Container [Chr20b], resolving seven inheritance-related issues simultaneously. As far as we can tell from our dataset, no new inheritance issues have been introduced since. Presumably, Firefox's inheritance logic was already more centralized at an earlier stage [Mozd].

Finding 7. Centralizing inheritance logic is an effective approach to mitigate inheritance-related bugs. Additionally, the observed disparity between browsers underscores the correlation between bugs and the underlying architecture.

In our analysis, we encountered three bugs in Firefox, where the introduction of new CSP functionality inadvertently weakened the security of existing features, while no such bugs were found in Chromium. For instance, use of the `strict-dynamic` keyword would allow the execution of event listeners, even when inline scripts should have been blocked.

Moreover, a total of five bugs for Firefox and Chromium were attributed to factors that fall beyond the scope of CSP functionality. In Firefox, an accessible browser resource that was intentionally exempt from CSP could be abused to execute an injected script when `strict-dynamic` is included in the employed policy. In Chromium, an HTML parsing issue allowed the theft of a `nonce` from a benign script, allowing the execution of injected code. In general, these bugs

[15] Two issues affected both browsers.

were caused by external browser functionality – unrelated to CSP functionality – that either act as a bypass or undermine correct policy delivery.

3.4.3 Bug reporting

In an effort to allow bug reports to be more easily queried, additional information is attached in the form of labels. While the Chromium platform utilizes the highly specific label `Blink>SecurityFeatures>ContentSecurityPolicy` label, the Firefox platform does not dedicate a label specifically to CSP-related issues. Among the Chromium bug reports, 33 out of 58 are not annotated with the aforementioned label. Of those, three do not contain any variation of the "CSP" or "Content Security Policy" strings in their title. Similarly, one of the 24 Firefox report titles does not contain a variation of these strings. This absence or inconsistent use of CSP-specific labels makes it more difficult for developers to identify similar or related issues.

Bug tracking platforms are often integrated with their respective source code repositories; for instance, when a bug ID is mentioned in a revision message, this revision will be automatically linked within the bug report as well. Correspondingly, this aids developers in keeping track of all revisions relevant to a certain report. To this end, we investigated how thoroughly associated revisions are linked to a bug report, regarding introducing and fixing revisions. If more than one bug report is associated with a particular bug (e.g. duplicate reports), we consider a revision linked if it is mentioned by at least one report. We found very similar results for Chromium and Firefox; bug introductions were mentioned for only 4% and 7% of bugs, while fixing revisions were mentioned in 87% and 86% of the cases, respectively.

Since metadata provided in reports lies at the base of understanding the described bug for both developers and researchers, we argue that more effort should be directed at providing consistent and comprehensive information. Furthermore, prior work is known to rely on similar metadata, and as such, this could provide for more accurate evidence [Asa$^+$12; Bra$^+$22b; ŚZZ05; Xia$^+$20; Yin$^+$11]. We believe that our framework could be a first step in this process to automatically identify the bug-introducing revision.

> ***Finding 8.*** CSP bug reports are often incomplete or labeled inconsistently, which complicates effective querying by both developers and researchers.

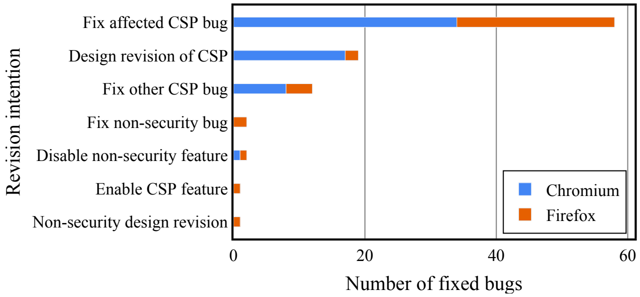


Figure 3.12: Intentions of revisions that fixed a CSP bug.

3.4.4 Bug fixing

The prevalence of intention labels for all fixing revisions is depicted in Figure 3.12. In total, all fixing revisions resolved a bug 95 times; this is less than the amount of introducing revisions because two bugs are not fixed at the time of writing. Clearly, with 58 revisions (61%), most bug fixes are intentional, whereas 12 (13%) are intended as a fix for another bug.

Of course, in the latter case developers could still be aware that a particular revision – intended for another bug – fixes a second one as a byproduct. As such, it is considered best practice to link the fixing revision to the bug report of the second bug as well, for transparency purposes. For all reported bugs resolved through the fixing revision of another bug, developers had only correctly linked the fixing revision for a single bug, while in five other cases any link to the fixing revision was missing.

Additionally, we identified two Chromium bugs and one Firefox bug that were made public before an effective fix was landed. Notably, the Firefox bug persisted in Firefox's most recent release version, prompting us to report the issue, after which it was ultimately fixed. All three issues left their respective browser exposed for at least one year after public disclosure. The reasons behind these premature public disclosures are very divergent:

Chromium bug 610441. The employed regression test leveraged only the `<meta>` tag to enforce CSP, while the bug could still bypass CSP enforcement through the response header.

Chromium bug 740615. An effective fix was reverted 26 hours later due to causing issues with the Google Docs service. This was not reflected in the bug report, and consequently, the report remained labeled as fixed.

Firefox bug 1460538. The regression tests were run on non-packaged builds, on which the applied fix was successful. However, the issue still persisted in packaged builds, unbeknownst to the developers.

> ***Finding 9.*** Due to a variety of reasons (e.g. incorrect test cases, unadvertised rollbacks, or misrepresenting test builds), bugs may be incorrectly marked as fixed, leading to their premature public disclosure.

3.5 Discussion

Backed by our findings, we argue that CSP implementation flaws increase the risk of bug introductions, whilst bug handling flaws increase the time frame of insecurity. In this section, we elaborate on the underlying issues, propose potential remedies and explore avenues for future research.

3.5.1 CSP implementation flaws

Our data suggests that non-foundational CSP bugs, caused by CSP-related and unrelated revisions, are not decreasing with time. Due to the dynamic nature of the Web, continuous occurrence of such revisions is inevitable.

In parallel, the evolution of CSP from a simple allowlist to a complex policy language capable of enforcing a wide range of security policies has introduced numerous new functionalities, including additional directives and keywords. Our analysis indicates that these extensions seldom compromise the security of existing CSP directives. However, it is worth mentioning that all three instances could have been mitigated through a more comprehensive testing strategy, duplicating existing regression tests to incorporate the new CSP functionality. Conversely, the most prevalent cause of bugs stems from new browser features and CSP functionalities that lack robustness upon their initial implementation, as well as unintended side effects resulting from CSP bug fixes.

Among the issues related to CSP's complexity, those concerning inheritance are most prominent in our dataset. Moreover, inheritance-related bugs often lead to more severe security risks, particularly in terms of active content control, affecting a larger area of the policy language as well. Here, Chromium was affected most with 20 inheritance-related bugs, compared to only four in Firefox. However, once Chromium centralized its inheritance logic, the overall robustness significantly improved, This highlights the substantial benefits of centralization, warranting the same for enforcement logic, and by extension demonstrates the importance of the browser architecture on the handling of security policies.

Furthermore, the frequency of bugs appears to be directly associated with the responsibility surface and capabilities of bypassed CSP directives. Consequently, the majority of bugs are linked to the `script-src` directive, used for active content control, whereas other use cases, which are comparatively simpler, exhibit minimal bugs. However, this pattern does not hold true for most `script-src` keywords. The number of bugs associated with specific keywords (i.e. `nonce`, `strict-dynamic` and `sandbox`) appears to be correlated with how long that keyword has been supported, with the exception of `hashes` which has a significantly lower number of bugs.

3.5.2 Improving bug handling

Our analysis identifies several shortcomings in the bug handling procedures of browser vendors, as several could have been avoided with minimal effort.

Foremost, we show that despite significant efforts such as WPT, bugs are not shared effectively among browser vendors. At closer inspection, we identify several reasons for this shortcoming; in some instances, no WPT tests are created as part of the bug fixing process. Our analysis also demonstrates that even when foundational bugs are initially avoided, they can reappear as regressions later, emphasizing the need for more comprehensive threat vector sharing, even when a browser is initially considered secure.

However, WPT seems to be the only means for browser vendors to share undisclosed bugs, but as WPT's test suite is public, added tests become visible to potential adversaries before the bug is fixed in other browsers. To address this concern, we recommend exploring alternative methods for sharing sensitive bugs among vendors. A low-effort solution would be to allow developers access to certain parts of the bug tracking platform of other browsers. This would make bug sharing independent of test creation and reduce the time it takes for bug knowledge to reach other vendors.

The fact that three bug reports were disclosed publicly before a fixing revision was implemented is particularly concerning, as it exposes end-users to potential attack vectors for an extended time period. A more stringent bug handling procedure would have helped prevent these incidents, especially considering that these bugs were incorrectly marked as resolved. Additionally, our analysis has uncovered instances where reports lack a link to the fixing revision, emphasizing the importance of enforcing this as a mandatory step for transparency and verification purposes. Furthermore, improving the accuracy of bug labels can facilitate the identification of similar bugs and support the implementation of stricter procedures, such as requiring a minimum of two reviewers for revisions aiming to address a bug labeled as a security issue.

3.5.3 Future work

For future research in this area (e.g. longitudinal evaluation of other policies), it is crucial to have complete and accurate bug reports. This would further enhance the quality and convenience of bug report and revision scraping, on which various related work relies. Moreover, the use of a standardized language to describe bugs in different contexts would greatly assist the integration of automated PoCs into various dynamic evaluation tools.

Solutions for CSP soundness specifically could lie in the field of formalization, where CSP would be consolidated as a formal definition. Several aspects of the Web have already been explored in a formalized context, demonstrating its effectiveness by discovery of previously unknown bugs [Akh+10; Ban+14; FKS16; FKS17; JTL12]. While this research direction would facilitate the sound introduction of new CSP functionality, formalizing the Web as a whole poses numerous significant challenges due to its dynamic nature and the complex interplay of its mechanisms and policies.

Another approach could leverage dynamic evaluation, similar to our methodology. However, the difficulty here lies in achieving true exhaustiveness, considering all combinations between supported mechanisms, policies and nested browsing contexts. While valuable efforts have been made to explore this approach [FVJ18; HMN15; Wi+23], only limited comprehensiveness has been demonstrated.

3.6 Related work

3.6.1 Dynamic browser policy evaluation

As one of the first, Aggarwal et al. employ fuzzing to detect inconsistencies and flaws in private browsing mode, also demonstrating the potential negative impact of extensions and plugins [Agg+10]. Research by Schwenk et al. found inconsistencies among browsers for the Same-Origin Policy, which could lead to vulnerabilities [SNM17]. In that same light, browser access control incoherencies were exposed by Singh et al., leveraging their automated evaluation framework WebAnalyzer [Sin+10].

Hothersall-Thomas et al. introduced BrowserAudit, a web application to validate multiple browser security policies [HMN15]. Third-party cookie policies, SameSite cookie policies and various anti-tracking measurements implemented by both browsers and browser extensions were deemed inadequate by Franken et al., employing their framework for dynamic evaluation through browser automation [FVJ18]. Luo et al. found that mobile browsers are susceptible to UI attacks due to insufficient protection and even demonstrate a declining trend in security over time [Luo$^+$17]. Luo et al. employed dynamic testing to construct a longitudinal overview of supported browser security policies in mobile browsers, uncovering that several widely-used browsers lack support for crucial policies, even several years after their introduction [Luo$^+$19]. In recent work, Rautenstrauch et al. uncovered several new vulnerabilities through the first systematic analysis of XS-Leaks [RPS23].

Finally, various frameworks have been developed to dynamically evaluate the security of JavaScript engines and web engines in different contexts as well [Din$^+$21; FVJ21; Par$^+$20].

3.6.2 Vulnerability studies

By examining the rate at which vulnerabilities are reported, Rescorla was the first investigating whether software matures in terms of security [Res05]. Unfortunately, no conclusive evidence was found for this hypothesis, confirmed by later studies as well [Ale$^+$20; Ale$^+$22; OS06]. However, Ozment et al. presented statistically significant evidence that the rate of foundational vulnerability reports does decrease over time [OS06]. Indeed, complementing this research, Edwards et al. demonstrated that the adding of large amounts of new code can decrease software quality and Alexopoulos et al. highlight the need for maintaining stable branches longer in order to detect maturing behavior[Ale$^+$20; EC12]. Furthermore, Alexopoulos et al. suggest that more expressive security metrics can greatly help us understand the vulnerability lifecycles.

Regarding browser development, Braz et al. uncover several root causes of regression vulnerabilities such as the complexity of browser interactions required for certain regression tests [Bra$^+$22b]. Zaman et al. and Munauah et al. underline the considerable differences between non-security and security bugs, and consequently motivate the need for this distinction in research [Mun$^+$17; ZAH11]. Research of di Biase et al. demonstrated the importance of code review and argues that more security issues are found in case more than two reviewers are involved, as opposed to the two-reviewer policy of Chromium at the time [BBB16]. In addition, further research indicates that security checklists do not significantly improve vulnerability detection and the relative order in which files are reviewed affects the probability for finding security issues [Bra+22a; Fre+22].

3.6.3 Content Security Policy

CSP has been the subject of various research projects, both with the focus on validating CSP implementations and on measuring CSP employment in the wild. To begin with, the aforementioned studies of Hothersall-Thomas et al. and Luo et al. investigated the CSP implementations of desktop and mobile browsers, respectively leveraging their automated frameworks [HMN15; Luo+19]. Van Acker et al. demonstrated how attackers could bypass strict CSP enforcement by abusing DNS and resource prefetching in major browsers [VHS16]. Other bypasses were pointed out by Somè et al., where incompatibility issues between the Same-Origin Policy and CSP would allow attackers to execute otherwise blocked scripts [SBR17].

The first study to identify and set out the challenges of CSP adoption was conducted by Weissbacher et al. [WLR14]. Based on their longitudinal study, they uncover various reasons behind the slow adoption rate and ineffective deployments of CSP, proposing potential remedies as well [Wei+16]. Calzavara et al. identified various issues with CSP deployment in the wild, such as liberal src expressions, use of inline scripts and underutilization of CSP's monitoring facilities [CRB16; CRB18]. Roth et al. brought to light the hurdles developers are facing when implementing a comprehensive policy[Rot+]. Calzavara et al. exposed how inconsistencies among the enforced CSP policies in browsers can lead to various gaps in clickjacking defenses of websites [Cal+20]. More recently, Stolz et al. showed that the use of the `unsafe-hashes` directive does not necessarily lead to more secure event handles, and argue that although the introduction of the directive is a step in the right direction, web developers should be advised to avoid inline scripts [SRS22]. Finally, Wi et al. uncovered 29 new CSP bypasses that lead to unauthorized script execution, by leveraging the first differential testing framework based on inconsistencies between browser implementations.

3.7 Conclusion

In this work, we presented BugHog, an automated framework to accurately identify introducing and fixing code revisions of browser security policy bugs. Leveraging this framework, we conducted a longitudinal analysis on CSP, one of the most extensive and important browser policies on the Web, mapping the complete lifecycle of 75 bugs.

Our results highlight multiple flaws in current bug prevention and handling practices, which lead to the premature public disclosure of unfixed vulnerabilities, and an avoidable lifetime of vulnerabilities due to inadequate threat vector sharing between vendors. We recommend that vendors explore alternative channels for sharing sensitive bug information and adopt more rigorous bug handling procedures. Our framework can aid in the effort to improve the compilation of more consistent and complete bug information, essential for a better understanding of their root causes. As such, we intend to open-source BugHog, which we plan to extend to evaluate other policies as well in future work.

4

Reading Between the Lines

An Extensive Evaluation of the Security and Privacy Implications of EPUB Reading Systems

Yo why y'all playing checkers on a chess set?

– D'Angelo Barksdale [SBM95]
(The Wire, 2002)

This chapter was previously published as:

> G. Franken, T. Van Goethem, and W. Joosen. "Reading Between the Lines: An Extensive Evaluation of the Security and Privacy Implications of EPUB Reading Systems". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1730–1747. DOI: `10.1109/SP40001.2021.00015`

In the preceding two chapters, we have focused on policy enforcement within web browsers and the many complexities of doing it soundly and comprehensively. In this chapter, we take a step back and explore whether the same challenges apply to native applications that employ browser engines. To this end, we conducted the first comprehensive evaluation of the security and privacy implications of a specific class of native applications that leverage browser engines: EPUB reading systems.

As previously discussed, automating browser evaluation is relatively straightforward due to the consistent and standardized methods of instruction. Yet, this advantage is not shared by EPUB reading systems, as they do not offer support for automation and command line-based instruction. On top of this, all reading systems employ their own unique user interface and are often only available for one platform. Despite these hurdles, we managed to create a semi-automated testbed of multiple EPUBs that, upon loading by a reading system, runs experiments and displays the outcome using the application's renderer. As such, we were able to evaluate 97 EPUB reading systems, covering seven platforms and five physical reading devices.

Our findings revealed several critical vulnerabilities, some of which would allow an adversary to access local files. As part of our responsible disclosure, we reached out to the vendors of 37 affected reading systems, including widely used applications like Adobe Digital Editions, Apple Books and Amazon Kindle. As a result of these efforts, numerous issues have been addressed and several CVEs have been assigned.[1] Additionally, our investigation has uncovered several underlying root causes of these issues. These include the misconfiguration security-sensitive settings, such as permitting all loaded EPUBs unrestricted access to the local file system and the inheritance of engine functionality, such as the ability to launch external applications through custom URI schemes. As part of a real-world analysis, we even demonstrated that it is possible to distribute malicious EPUBs through the official web stores of popular reading systems, including Amazon Kindle, Apple Books and Rakuten Kobo.[2]

In addition to implementation issues, we also identified several shortcomings in the EPUB specification. For instance, the specification previously allowed reading systems to grant EPUBs access to both local and remote files, potentially creating vulnerabilities for attackers to exploit and compromise personal information. Furthermore, the specification did not explicitly address all potential attack vectors. Fortunately, our findings and suggestions for improvement were taken into account by W3C. Consequently, several significant enhancements have been integrated into the next iteration of the EPUB standard, EPUB 3.3 [GCH23]. One notable improvement is the introduction of a strict requirement that prevents a loaded EPUB from referencing `file://` URLs, effectively blocking access to the local file system. Additionally, the specification

[1] CVE-2020-3798, CVE-2019-8789 and CVE-2019-8774.

[2] In the context of this experiment, paper co-author Tom and I made our debut in the realm of contemporary art books with our publication, *"Not for Sale: A Timeless Piece of Art"*. However, the book was retracted from all stores following the experiment's conclusion due to lack of artistic value. Nonetheless, a digital copy remains available upon request.

now recommends that user consent should be a prerequisite for allowing a loaded EPUB to access the network or to open external applications. In support of these improvements, we have integrated various segments of our test suite into the official W3C test suite, which serves as the benchmark for testing the compliance of EPUB reading systems.

The presentation preview for our talk at the *IEEE Security & Privacy Symposium* was awarded with the *Best Video Award.*[3]

4.1 Introduction

In the last decade, digital books have gained significantly in popularity, and experts argue they are here to stay [PwC10; Wis13]. Today, almost every newly published book or magazine is made available in a digital format, in addition to their physical copy. EPUB e-book creation is considered fairly straightforward, which combined with the possibility to publish without any vendor interposition, explains in particular its popularity among self-publishing authors and within open license communities such as Project Gutenberg[4]. EPUBs primarily consist of Extensible HyperText Markup Language (XHTML) documents and CSS stylesheets, bearing close resemblance to web pages. Consequently, browser engines are often employed by reading systems to render EPUB content. Next to static content, the EPUB standard also allows for media such as audio or video, and dynamic content leveraging JavaScript.

Because EPUB reading systems are closely related to web browsers, they are prone to similar security and privacy issues. A malicious EPUB could leverage the reading system's capabilities to mount attacks against the user, much like the dynamics between a malicious website and a browser. For instance, since the introduction of EPUB3, several blogs and articles have voiced concern about the capabilities associated with supporting JavaScript in EPUB reading systems [Bal12a; Bal12b; Eri13; Jun17; Nat13]. The EPUB specification acknowledges these concerns by dedicating one of its sections to security recommendations for EPUB reading system developers [CG19]. However, these recommendations are very lenient and lack strict enforcement.

The EPUB specification defines a large number of (optional) capabilities, several of which are questionable from a security perspective, such as access to the local filesystem, especially when considered in combination with access to remote endpoints. Consequently, since most EPUB reading systems are not well-documented, this makes it difficult for the user to assess what privileges the

[3] `https://www.youtube.com/watch?v=GDL6KWOn8Ic`

[4] `https://www.gutenberg.org/`

reading system provides to an opened EPUB. Moreover, there is no indication whether a reading system is compliant to the EPUB specification, nor are we aware of any study evaluating their capabilities and compliance.

In this chapter, we present the first extensive evaluation of EPUB reading systems, addressing their capabilities, compliance with the EPUB specification, and security and privacy implications. To this end, we crafted an extensive testbed that covers a wide range of the threat surface, consisting of EPUBs which are loaded in EPUB reading systems to perform a semi-automated evaluation. As soon as such an EPUB is opened by a reading system, the embedded experiments are executed, after which the resulting data is rendered or sent to a server. This way, we evaluated 97 of the most popular EPUB reading systems, covering seven different platforms and five physical e-reader devices. Our evaluation uncovered that almost all reading systems that execute embedded JavaScript do not fully respect the specification's security recommendations, of which 16 can be abused by malicious EPUBs to leak information about the local filesystem. We reached out to all vendors in order to report the identified issues.

Moreover, to complement our semi-automated evaluation, we manually inspected three widely used EPUB reading systems for implementational flaws, revealing several severe security vulnerabilities. We discovered a universal XSS affecting two browser extensions (≈300,000 browser installations), and the ability to leak arbitrary files as soon as a malicious EPUB is opened on a Kindle (the most widely used physical e-reader [DG19]). These findings indicate that the results of our semi-automated evaluation should be considered as a lower bound, and that in fact EPUB reading systems may face even more security issues that are implementation-specific. We argue that by limiting the capabilities of the reader application and the employed rendering engine to a minimum, the threat surface can be significantly reduced. For instance, iOS reading systems can only access files within the application by design, and thus none could be abused to leak sensitive files. Furthermore, we explored the presence of real-world abuse by analyzing more than 9,000 EPUBs obtained from five online e-book stores and two file-sharing platforms. We did not find any evidence of ongoing abuse, allowing EPUB reading system developers to adopt adequate security measures before users are actively being exploited. Finally, we show that four out of six evaluated self-publishing EPUB services do not adequately vet submitted manuscripts, which could lead to the distribution of malicious EPUBs through legitimate channels.

We make the following contributions:

- We developed a testbed of numerous EPUBs to assess the security and privacy impact of various aspects of EPUB reading systems and the rendering engine they employ.
- By applying this testbed in a semi-automated analysis, we evaluated a total of 97 EPUB reading systems, of which 92 were freely available as applications on desktop (Windows, macOS and Ubuntu) and mobile (iOS and Android), or as a browser extension (Chrome and Firefox), and of which five were physical e-reader devices.
- The result of this analysis shows that many reading applications can be abused, either by leaking file contents, or by violating the user's privacy expectations.
- To explore both ongoing and potential abuse in the EPUB ecosystem, we downloaded over 9,000 EPUBs from two torrent sites and five online e-book stores, and assessed the vetting process of six popular self-publishing services.
- Lastly, based on the identified issues and their root cause, we propose to make the EPUB specification more strict. Moreover, to encourage consumers and developers to measure the security and privacy impact of their reading systems, we have released the EPUBs used in our evaluation along with the source code to craft them.

4.2 Background

In May 2019, W3C issued EPUB 3.2, the most recent version of the EPUB standard at the time of writing [GC19]. In the remainder of this chapter, we refer to this particular version when discussing the EPUB standard, unless specified otherwise.

4.2.1 EPUB technical standard

The EPUB standard consists of five sub-specifications, each defining complementary core features and functionality. For reasons of brevity, we will not describe each sub-specification, instead we will discuss the standard as a whole.

The EPUB format and the internal structure of a compliant EPUB file, or a so-called EPUB Container, are visualized in Figure 4.1. An EPUB is a single-file container with the `.epub` extension, of which the included content

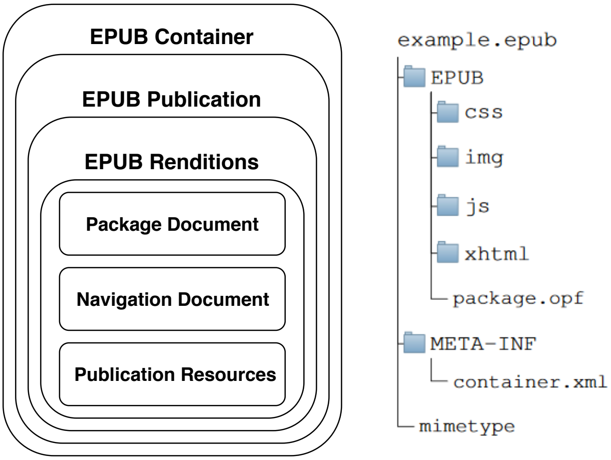


Figure 4.1: On the left a visual representation of the EPUB format, and on the right the internal file structure of a compliant EPUB archive.

is compressed in a ZIP archive. The `mimetype` file indicates the EPUB Open Container Format (OCF) media-type, with `application/epub+zip` as the two-part identifier [IAN14].

An EPUB Publication, included in the Container, must consist of at least one Rendition, the specific rendering of the EPUB's content. A Rendition is represented by an EPUB Package, which consists of the Package Document (`package.opf`), the Navigation Document and all Publication Resources used to build the Rendition. The Package Document conveys various types of information such as the title and authors of the EPUB Publication. Moreover, it also defines the sequence in which the Content Documents are rendered, called the spine.

Publication Resources characterize the actual content and layout of the EPUB Rendition. The standard makes a distinction between Core Media Type Resources and Foreign Resources. The former are resources of media types that are deemed supported by all Reading Systems. This set of media types consists of Content Documents (XHTML or SVG media type files), CSS stylesheets, and various image and audio formats. The latter are resources of which support is not mandatory and therefore require fallback mechanisms to Core Media Type Resources, in case the Reading System does not offer support.

The Package Document is also used to assign special properties to particular Content Documents. An interesting example is the `scripted` property, which indicates that the referred Content Document contains executable JavaScript. EPUB Containers are allowed to contain such Scripted Content Documents, however, EPUB Reading Systems are not obligated to support script execution. As such, the EPUB standard states that Scripted Content Documents should retain their integrity when scripting support is disabled, without any loss of information or legibility.

4.2.2 EPUB reading systems

EPUB Reading Systems are applications that interpret and render EPUB files according to the EPUB specification. They come pre-installed on physical e-book reading devices (e.g. e-ink readers), but are also available on smartphones, tablets, desktop computers, and even in the form of browser extensions. Nearly all applications are free (some with in-app advertisements) and often provide support for various other e-book formats.

The EPUB specification dictates the minimal requirements that should be met by an EPUB Reading System. These requirements are mostly related to rendering and presenting the content that is the EPUB Publication. Only a few paragraphs of the standard are dedicated to security considerations, with special attention to providing support for scripting [CG19]. Here, Reading System developers are provided with several attack vectors that should be considered, and with recommendations on how to deal with security-related issues concerning scripted content execution.

One of the recommendations states that the Reading System should behave as if a unique domain were allocated to each Content Document, consequently isolating documents from each other. This isolation is enforced by the Same-Origin Policy, which dictates that one origin cannot access resources from another. Reading Systems that allow scripting and network access should also notify the user whether any network activity is occurring, and ideally provide functionality for the user to disable it.

According to its specification, EPUB Reading Systems may allow an EPUB to store persistent data (e.g. LocalStorage). This data is recommended to be considered sensitive, and therefore this data should not be accessible by other documents.

4.2.3 Same-Origin Policy

Modern browsers employ a wide range of policies to protect users against malicious websites, among which the highly essential SOP [Mozk]. This policy in particular is used to isolate documents and scripts located on different origins, to prevent one website to perform undesirable actions on, or steal sensitive user information from another website. Two URLs share the same origin if their scheme, host and port are identical. For example, when the origin `https://attacker.com` tries to extract information from `https://bank.com`, this would be prevented by the SOP. Not only documents and scripts are protected by the SOP, but also any information stored by the LocalStorage API [Hic16]. Although cookies are not subjected to the SOP, they can only be accessed by associated domains. As an extension to the SOP, websites cannot instruct the browser to render or access files located on the user's file system. For example, when a website leverages an iframe to render a file by referring to `file:///etc/passwd` or employs the XMLHttpRequest or Fetch API to access its content, the browser will refuse.

4.3 Motivation

While e-books have grown to be a multi-billion dollar industry [PwC10; Wis13] and countless EPUB reading systems are available on essentially every platform, the reading system ecosystem has never been subjected to a comprehensive security or privacy assessment. In the following subsections, we argue why such an assessment is imperative.

4.3.1 Intransparency

Most EPUB reading systems rely on browser engines to render e-book content. Over the last few years, there has been an extensive growth in the number of features that these browser engines support, significantly increasing their threat surface [STK17]. Scripting in e-books, which was already suggested as an optional functionality in the 1999 Open eBook Publication Structure (OEBPS) specification, could be used to launch a variety of attacks to circumvent the same-origin policy, or even to attack the underlying operating system. As such, opening an e-book could introduce the user to a plethora of attacks, which have not been extensively explored to this date. In face of these threats, the more recent editions of the EPUB standard now include a section on security considerations for EPUB reading system developers.

However, we argue that these considerations lack binding requirements and are insufficiently concrete. In that regard, even if an EPUB reading system is compliant with the official specification, users do not have any guarantee that their security and privacy will be safeguarded. For instance, a compliant reading system might allow an EPUB to freely access the user's file system and send a copy of it to a remote server. Moreover, countless curated lists only recommend reading systems based on usability features and supported e-book formats, making it nearly impossible for users to verify whether an application is sufficiently secure.

We argue that the more features are being added to the specification, the more it will cripple the transparency of security and privacy factors in EPUB reading systems as long as no clear compulsory considerations are included. However, even when these are included, still, there is no straightforward way to verify compliance of a reading system. This uncertainty is one of the reasons why we deem a comprehensive evaluation of EPUB reading systems imperative. We aim to improve this much-needed transparency by evaluating the most popular EPUB reading systems, leveraging a semi-automated analysis. In the following subsections, we discuss two attacker models which aim to abuse EPUB reading system capabilities, impacting the user's security and privacy.

4.3.2 Malicious EPUBs

Nowadays, tens of thousands of EPUBs are made available online for free, either legally or illegally. Whereas EPUB submissions to the Gutenberg Project are subjected to examination by volunteers [Pro], various other channels omit third-party validation and share the EPUB as-is (e.g. torrent sites, social media). A study of the UK government's Intellectual Property Office finds that approximately 17% of e-books are illegally consumed online, accounting for around four million e-books [Int17]. As such, users may face various threats when accessing an e-book obtained either from a publisher who does not sufficiently sanitize or verify the published books, from a malicious website directly, or from a file-sharing platform.

The attacker could configure the EPUB such that upon opening, a malicious JavaScript payload is executed. Depending on the capabilities and vulnerabilities of the reading system, the attacker could try to either extract sensitive system files, such as the browser's cookie store, and then send the contents of these files to an online web server. Furthermore, if the browser engine used by the reading system is outdated, it might contain publicly known vulnerabilities that can then be exploited by the malicious e-book in order to compromise the system.

In an explorative experiment using Chrome and Firefox, we assessed whether a website could automatically cause a malicious EPUB to be loaded in an installed reading system. In both browsers (on desktop and mobile), this requires at least one user interaction. Although a website can instruct the browser to download an EPUB (e.g. clicking a URL through JavaScript), still one user click is required to actually open it. However, an EPUB reading system installed as a browser extension could intercept the download and automatically render the EPUB.

4.3.3 Tracking EPUBs

E-books in a proprietary format are usually distributed through the associated vendor's online bookstore, which is often embedded within the vendor's own reading system (e.g. physical e-reader devices or applications). Leveraging their own proprietary formats and reading systems, vendors are known to harvest user data based on interactions with their reading system [Alt12; Flo14; Kas10].

Although EPUB is an open format and not affiliated to any specific vendor, distributors of EPUBs might still be able to track users. To supplement their recommendation system, the distributor might try to figure what other books make up the user's library. This can be accomplished if the user's reading system allows EPUBs to render local files located within the directory where the unpacked EPUBs are stored. Then, to scan the contents of the library, each distributed EPUB could include a list of popular EPUBs, and code to test their presence on the system. Even when the targeted EPUB reading system does not allow rendering local files, timing attacks can prove as a suitable alternative. The distributor could be even more intrusive by scanning for other information, such as installed applications and browsing history, in the same way. Moreover, a tracking-enabled EPUB might try to associate the user with an online browsing session, e.g. by fingerprinting the installed fonts. The EPUB might even try to obtain an even more intrusive device fingerprint, e.g. by detecting the presence of specific files on the system, which in most cases can also reveal the username.

4.4 Methodology

To evaluate the potential threats posed by opening an EPUB file, we conduct a series of experiments. In this section we describe our experimental setup through which we test a wide variety of EPUB reading systems for different primitives that could be used to launch attacks.

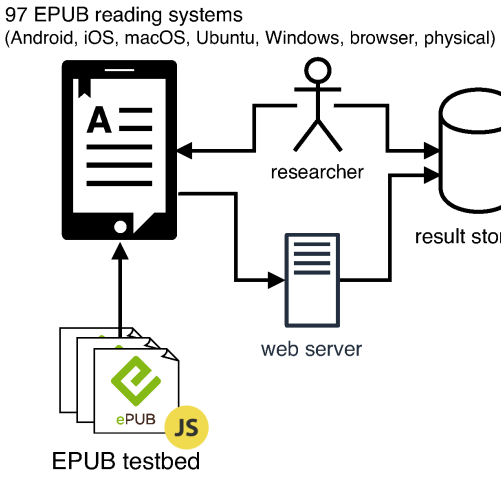


Figure 4.2: Overview of our experimental design. The various EPUB files that make up our testbed are manually loaded in the tested reading system. If remote communication is available, the results are automatically submitted to a web server, which will store it in the database. Alternatively, these are manually copied from the e-book.

4.4.1 Experimental design

Our experiments aim to document the capabilities entrusted to EPUBs by the reading systems, and to detect related security and privacy issues. Because most reading systems are closed-source, we opt for a black-box approach, developing a testbed of various EPUBs which, upon loading, instruct the EPUB reading system to run embedded experiments. Because of the high variety of reading systems, both in terms of the platform they are run on as well as the functionality they provide and their user-interface, we deemed it infeasible to perform a fully automated evaluation while maintaining completeness. Instead, we opted for a semi-automated approach, where we use JavaScript code to render the results of our experiments in the reader, or, if possible, send these to a remote web server. As such, the manual effort is limited to copying this output from the EPUB reader into a file that can be further evaluated by our analysis framework. An overview of our experimental design can be found in Figure 4.2.

Supported by this setup, we aim to evaluate the presence of certain "primitives" that are required to launch attacks. For instance, in order to leak the contents of a file on the local file system, an attacker requires the ability to render content from local files, execute JavaScript code, and finally send remote requests. For every primitive functionality, our testbed uses a separate EPUB file that tests its presence. The reason for this is that EPUB reading systems label certain EPUBs as corrupt when these try to execute unsupported functionality. Several experiments rely on specific functionality such as JavaScript execution; when this functionality is not present, the associated experiments can be omitted. The decision on which experiment to perform next is each time imposed by our testbed protocol.

We used the official EPUB Validation Tool [W3C19] provided by W3C to validate conformance with the standard. To accommodate all EPUB reading systems, the embedded JavaScript uses ECMAScript 5 functionality because the more recent ECMAScript 6 is not widely supported among reading systems. We have publicly released all code required to construct this testbed of EPUBs.[5] In the rest of this section we discuss all features that were evaluated, an overview is depicted in Figure 4.3.

JavaScript execution

Because most reading systems do not disclose whether JavaScript is supported, which is indeed an optional feature in the EPUB specification, this information needs to be obtained empirically. JavaScript support might be an important trait to the user, e.g. to support interactive EPUBs, but even more so to a potential attacker, considering the substantially increased threat surface. That is, through JavaScript a multitude of different APIs become available, which could be used to request local or remote resources, or even access user media devices (MediaDevices API [Mozj]).

We test three different ways of how JavaScript can be included in an EPUB: (i) directly embedding code with a `<script>` tag in an XHTML file (inline), (ii) reference a separate JavaScript file within the EPUB by setting the `src` attribute of `<script>` tags (external), and (iii) reference a JavaScript file hosted on an external web server (remote). All three approaches were evaluated by dynamically changing the content of a visible HTML element through inline, external or remote JavaScript code. When the content of such an element assumed the dynamically assigned value over the original value, we could safely assume that JavaScript was executed.

[5] `https://github.com/DistriNet/evil-epubs`

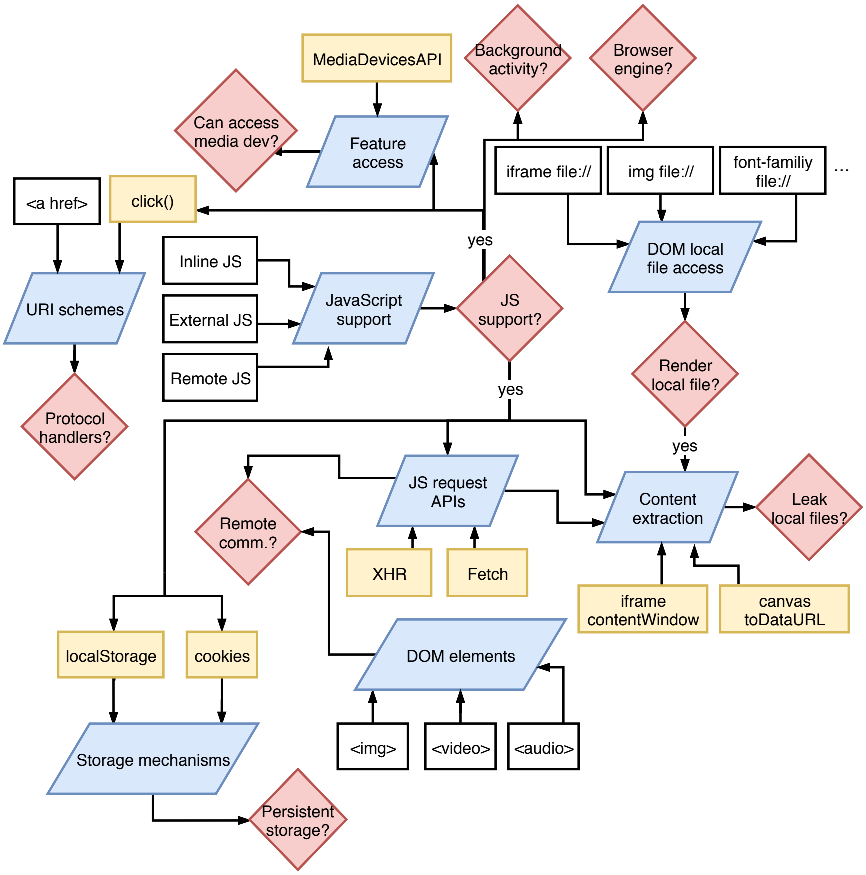


Figure 4.3: Overview of the different EPUB experiments. In order to assess certain features (red) of the reading system, we used several experiments (rectangular), both with (yellow) and without (white) JavaScript; these experiments are grouped by category (blue).

Local file system access

The EPUB specification allows reading systems to support references to certain types of resources on the local file system, explicitly mentioning audio, video and fonts, but also any resource retrievable by a script [GC19]. JavaScript-supporting reading systems that implement this optional feature may implicitly grant every EPUB the ability to retrieve files from the user's operating system. Even when the SOP is enforced to prevent content leaking, as is recommended by the specification [CG19], a malicious EPUB might still be able to gather sensitive information such as the presence of certain files, or even the user's account name.

For this evaluation, we performed three sets of experiments in which the EPUB attempts to access five types of resources: textual files (.html, .txt, .log, .bogus), images (.png, .jpg), audio (.mp3), video (.mp4) and fonts (.ttf). In an attempt to bypass the potentially restrictive direct access to the local file system, we do not only refer to the resource through its absolute path (`file://` protocol), but also leverage relative symbolic links. For UNIX systems, we were able to enclose correctly functioning symbolic links pointing to a file and folder in the ZIP file, which is essential for embedding them in an EPUB. We did not find a way to reproduce this on Windows. These experiments were considered successful only when the obtained information could be leaked to a remote server (see Experiment 4.4.1).

In the first set of experiments, the EPUB attempts to render the local user files by simply including them by means of an `iframe`, `img`, `audio` or `video` element, or by assigning a CSS `font-family` to include a font. When such a resource is rendered, it is trivial to confirm its existence on the local file system. For iframes and images, an observable `load` event is fired when the subresource was successfully loaded. Similarly, on audio and video elements the `canplaythrough` event is fired. Although no such event exists for fonts, existence of font files can be inferred by leveraging `canvas` elements to check whether the referred font has been applied to a text box.

The second set of experiments aims to determine whether the content of local resources can be accessed through the XMLHttpRequest and Fetch API [Moz17b; Moz17f] or by leveraging content-specific methods. For textual resources, the EPUB tries to access the rendered content within an iframe through its `contentWindow` attribute. Images, on the other hand, can be encoded in the base64 format through the `toDataURL` functionality of the `canvas` element. However, when reading systems use a unique domain to host the EPUB's content, as is recommended by the specification [CG19], the SOP disallows access to the content of the referred resource. Again, here we can leverage symbolic linking to make it appear as if the referred content is hosted on the same domain.

Well-secured EPUB reading systems will prevent the EPUB from rendering local files and leaking their content, however, we might still be able to leak the existence of a particular file by leveraging timing attacks. This was evaluated in an additional experiment, by measuring the time between setting the `src` attribute of an image element and the firing of the `onerror` event, for both a URL of an existing file and a non-existing file. In every experiment, this measurement was performed 20,000 times, alternating the sequence order of the existing and non-existing file to reduce potential noise. When in a significant portion of these cases the measured time was larger for the existing file than for the non-existing file, or vice-versa, we consider a timing attack to be viable. For all such labeled reading systems, this calculated accuracy was at least 95%, except for two reading systems where we measured an accuracy of about 75%. However, in all cases the accuracy can be increased by performing multiple measurements. On MacOS and Linux, we used a filesystem in user space (FUSE) [nrc19] in advance to determine whether the local filesystem is accessed in an attempt to read out the file.

Remote communication

Similar to the local resource access discussed in the previous section, the EPUB specification allows reading systems to support references to online resources for certain resource types [GC19], implying that remote communication with a server is possible. However, the standard also acknowledges the security implications produced by this trait and advises reading system developers to explicitly notify users of network traffic, and ideally, even request user consent in advance [CG19]. Indeed, this capability is essential for relaying sensitive information to a tracker, or for receiving instructions from an attacker.

In this experiment, we investigate whether an EPUB is able to communicate with remote servers while it is opened by the reading system, and whether the user is notified of the occurring network traffic. Various HTML tags can be used to initiate HTTP requests, and in an attempt to be exhaustive we leveraged the comprehensive collection on the HTTPLeaks GitHub repository [Cur19], in combination with the XMLHttpRequest and Fetch API [Moz17b; Moz17f]. When any of these requests reaches the remote server, we label the EPUB reading system as supporting remote communication. As we manually load the crafted EPUBs in the readers, we also take note of any request for consent that was presented to the user.

Persistent storage

In modern browsers, websites have access to various mechanisms to store data locally, such as cookies and the LocalStorage API [Hic16]. Again, EPUB reading systems might inherit this functionality to provide storage capabilities to EPUBs. The EPUB specification rightfully recommends reading system developers to treat all stored data as sensitive, preventing other documents from accessing.

In these experiments, we first determine whether the EPUB reading system supports persistent storage through one of the two mechanisms. Since reading systems might merely provide the API, neglecting the persistence trait, we evaluate whether the stored information persists after closing the EPUB reading system. To adequately validate inter-session persistence, we start an initial session by opening the crafted EPUB. After rendering is complete, we close the reading system, thereby ending the first session. Finally, by reopening the same EPUB file and starting a second session, we inquire whether any cookies or LocalStorage entries have remained.

In an additional experiment, we check compliance with the recommendation to isolate this data from other documents. For this, we use different EPUBs in subsequent sessions, validating whether a modification by the first EPUB is detectable by the second.

Feature access

Modern browsers allow websites to request access to features, such as the user's geolocation, microphone and webcam [Bos$^+$19; Pop16]. When such access is requested, the browser will ask the user for consent to allow the website to access the indicated resource. We did not find any occurrence of these mechanisms in the EPUB specification, however, since most EPUB reading systems rely on browser engines, it is possible that this functionality is inherited. Because access to these media devices could allow an EPUB to record the user's surroundings or determine the user's location, it proves a tempting target for a potential attacker.

In this experiment, we evaluate whether the GeoLocation and MediaDevices API are made available in EPUB reading systems, and if so, whether user consent is required.

URI schemes

On the Internet, resources are referenced through URI, of which most rely on the HTTP or HTTPS protocol. However, by using custom URI schemes, websites can also instruct the browser to open applications upon activation of the URI (e.g. by clicking a hyperlink), even passing on arguments in the URI. For instance, the `mailto:` scheme is often employed to refer to an e-mail address, and when activated, will open the operating system's default mail application [DMZ10]. Whereas the `mailto:` scheme is one of the official URI schemes issued by the Internet Assigned Numbers Authority (IANA) [IAN], there are also many non-registered schemes used in practice.

To prevent misuse, modern browsers generally request confirmation from the user to initiate another application. This precaution is considered critical as URI links can be activated without any user interaction, e.g. through the `click()` function in JavaScript. Depending on the security considerations of a referred application, leveraging the arguments of such an activation could initiate a phone call, send a mail or download a file, facilitating various attacks by respectively exposing a user's phone number or e-mail address, or downloading a malicious payload.

In this experiment, we investigate whether EPUB reading systems support initiation of applications through URI schemes, and if so, whether the reader requested permission from the user for this action.

Browser engine evaluation

Considering that browser engines require regular patching to fix security bugs, disclosed vulnerabilities could be abused to target reading systems with an outdated browser engine.

In this experiment, we explore browser engine use in EPUB reading systems by evaluating whether the embedded browser engine is outdated and insecure. While at first sight consulting the user-agent string poses a straightforward solution, this information might not correctly represent the underlying browser engine. For instance, reading systems could have modified it, and WebKit has stopped updating the user-agent string altogether [Webc]. Therefore, we identify the embedded browser engine version by fingerprinting browser engines based on supported features, leveraging MDN's browser compatibility dataset [Mozi]. Such a fingerprint is constructed by evaluating support for each HTML element, attribute and JavaScript API present in the MDN dataset. This way, we collected almost 100 distinct fingerprints from applications whose embedded browser engine is known, and subsequently used those to determine the embedded browser engine of the reading systems. A browser engine is marked insecure if its age has surpassed at least three years, and if any vulnerabilities are publicly disclosed.

Background activity

To facilitate multi-tasking, mobile applications retain operation for a short time after focus is lost (e.g. when the user switches to another app), depending on the application's configuration. However, to improve battery life and memory consumption, mobile platforms impose restrictions on background tasks. We did not find any official documentation on how much time an application is allowed to run in the background on iOS, yet an Apple staff member has stated on the official Apple Developer Forums that this is around three minutes after losing focus [App17]. Similarly for Android, this exact time limit is undocumented, but is said to be around ten minutes before the application is forced into idle mode [Anda].

Likewise, EPUB reading systems can invoke this functionality to remain running when switched to the background, increasing the time window of a potential attack. By embedding a counter inside the EPUB, we can detect whether a switch to the background paused the embedded JavaScript execution.

4.4.2 Evaluated EPUB reading systems

This testbed was used to evaluate a set of 92 free EPUB reading systems, available for desktop platforms (Windows 10, macOS 10.14.6 and Linux Ubuntu 18.04), mobile platforms (iOS 12 and Android 9) or as browser extensions (Chrome 78 and Firefox 70). We used the iOS App Store, Google Play Store, Chrome Web Store and Firefox Add-on Store for selecting and installing reading systems on iOS, Android, Chrome and Firefox. The selection was based on the store's search functionality, using the terms "epub reader" and "ebook reader", scanning the first 100 results each time. For Android, we limited our selection to applications that were downloaded by at least 5.000 users. For the desktop platforms, we used a web search engine to make up the selection of EPUB reading systems, also leveraging curated lists. Here, we installed all encountered applications by downloading them from a website or installing them using the respective application store of the platform. By converting the evaluation EPUBs to AZW e-books, we also evaluated Kindle applications if available on the platform. For a complete overview of all evaluated EPUB reading systems, we refer to Appendix C.

Additionally, we evaluated the five most popular physical e-reader devices (Kindle Paperwhite 4, PocketBook Touch HD 3, Kobo Clara HD, Onyx Nova Pro, Tolino Shine 3). Their pre-installed EPUB rendering applications were tested out-of-the-box.

4.5 Results

This section will cover the results obtained by performing the semi-automated evaluation described in the previous section. Some reading systems were not able to render a perfectly compliant EPUB 3.2 e-book and were therefore excluded from our evaluation (see Appendix C).

4.5.1 Desktop

For desktop-based reading systems, experiments were run on Windows 10 (17763), macOS (10.14.6) and Ubuntu (18.04).

Windows

Table 4.1 shows the results of our evaluation on the Windows platform, which consisted of 15 reading systems. Of these reading systems, five execute embedded JavaScript, which can be escalated to leak at least the existence of certain files, and two of them can even be abused to leak file contents. Calibre 3 and MS Edge grant EPUBs the ability to open third-party applications installed on the user's operating system. Interestingly, only the latter asks for the user's consent.

Interestingly, Adobe Digital Editions's rendering behavior differs between files residing on the local file system and files residing on a network share. Although

Application	JavaScript		Local Resources			Remote communication	Persistent storage		Features	URI handles	Insecure engine
	Local	Remote	Existence	Render	Leak		Cookies	LocalStorage			
Adobe Digital Editions (4.5.10)	●	●	🖼	🖼 ‡	-	●	○	●	-	○	○
Bibliovore (2.0.2.0)	○	○	-	-	-	○	○	○	-	○	-
BookReader (1.6.0.0)	○	○	-	-	-	○	○	○	-	○	-
Bookviser Reader (6.8.1.0)	○	○	-	-	-	○	○	○	-	○	-
Calibre (3.40.1)	●	●	</> 🖼 🗋 🗎 ■ 🔊 🎥 x	</> 🖼 🗋 🗎 🔊 x	-	●	○	○	-	●	●
(4.3.0)*	●	○	-	-	-	●	○	○	-	○§	○
CoolReader (n/a)	○	○	-	-	-	○	○	○	-	○	-
EPUB File Reader (1.5)	●	●	</> 🖼 🗋 🔊 🎥	</> 🖼 🗋 🔊 🎥	</> 🖼 🗋	●	○	○	-	○	○
FBReader (0.12.10)	○	○	-	-	-	○	○	○	-	○	-
Freda (4.21)	○	○	-	-	-	○	○	○	-	○	-
Icecream Ebook Reader (5.19)*	●	○	</> 🖼 🗋 🗎 x	</> 🖼 🗋 🗎 x	</> 🖼 🗋 🗎	●	○	○	-	○	●
Liberty (1.0.0.13)	○	○	-	-	-	●	○	○	-	○	-
MS Edge (44.17763.1.0)	●	●	🔊 🎥	🔊 🎥	-	●	○	○	-	●†	○
Nook (1.10.1.15)	○	○	-	-	-	○	○	○	-	○	-
Overdrive (3.8.0)	○	○	-	-	-	●	○	○	-	○	-
SumatraPDF (3.1.2)	○	○	-	-	-	○	○	○	-	○	-

</> .html file | 🖼 .png/.jpg file | 🗋 .txt file | 🗎 .log file | ■ .bogus file | 🔊 .mp3 file | 🎥 .mp4 file | x .ttf

[*] Only executes inline JavaScript.
[†] Requires user consent.
[‡] Additionally renders textual files (.html, .txt), images (.png, .jpg), audio and video residing on a connected network share.
[§] Allows EPUB to open URL in default browser.

Table 4.1: Evaluation results for EPUB reading systems on Windows.

the means of access are identical; through an absolute file path, it only allows an EPUB to render images located on the former, whereas textual files (.html and .txt), images, audio and video can be rendered if located on the latter. This can be exploited to enumerate both local files and files residing on a network share. This vulnerability was assigned CVE-2020-3798, and has been resolved since Adobe's 4.5.11.187303 release of the application [Ado20].

Our tests identified WebKit 538.1 as the underlying browser for both Calibre 3 and Icecream Ebook Reader, which was released in 2014. This engine is considered insecure, since several vulnerabilities are publicly disclosed. For instance, by leveraging such a vulnerability [Lee17], we were able to leak arbitrary file contents in Calibre 3. Fortunately, Calibre started using an updated engine since its major update to version 4, effectively mitigating this vulnerability.

macOS

As shown in Table 4.2, except for FBReader and Amazon's Kindle application, all reading systems evaluated on macOS support JavaScript execution. All reading systems that support JavaScript can communicate with a remote server without informing the user. Moreover, half of the ten tested readers can leak the presence of certain resources on the local file system by rendering them. Three of those even allow an attacker to leak arbitrary files to a remote server.

Furthermore, four reading systems allow EPUBs to open installed applications on macOS, leveraging specific URI schemes, without requiring user consent. It is considered good practice for these referred applications to require user interaction before irreversible actions are undertaken, however, this is not always the case. For instance, when Skype for Business is configured as the default app to handle `tel:` scheme URIs, activation of such a URI will immediately lead to calling the included phone number. Although the results of this action are

Application	JavaScript		Local Resources			Remote communication	Persistent storage		Features	URI handles	Insecure engine
	Local	Remote	Existence	Render	Leak		Cookies	LocalStorage			
Adobe Digital Editions (4.5.10)	●	○	-	-	-	●	○	○	-	●	○
Apple Books (1.17)	●	●	-	-	-	●	○	●	-	○	○
Azardi (43.1)	●	○	⟨/⟩🖼🗋🖹■🔉🎞⊠	⟨/⟩🖼🗋🖹■🔉🎞⊠	⟨/⟩🖼🗋🖹■	●	○	●	-	○	●
BookReader (5.14)	●	○	⟨/⟩🖼🗋🖹 🔉🎞⊠	⟨/⟩🖼🗋🖹 🔉🎞⊠	⟨/⟩🖼🗋🖹■	●	●	●†	-	●	○
Calibre (3.40.1)	●	●	⟨/⟩🖼🗋🖹■🔉🎞⊠	⟨/⟩🖼🗋🖹 🔉🎞⊠	-	●	○	○	-	●	●
(4.3.0)	●	○	-	-	-	●	○	○	-	○*	○
FBReader (0.9.0)	○	○	-	-	-	○	○	○	-	○	-
Kindle (1.25.2)	○	○	-	-	-	○	○	○	-	○*	-
Kitabu (1.2)	●	●	⊠	⊠‡	-	●	○	○	-	●	○
Murasaki (1.0.2)	●	●	⟨/⟩🖼🗋🖹■🔉🎞⊠	⟨/⟩🖼🗋🖹 🔉🎞⊠	⟨/⟩🖼🗋🖹■	●	○	○	-	○	○

⟨/⟩.html file | 🖼.png/.jpg file | 🗋.txt file | 🖹.log file | ■.bogus file | 🔉.mp3 file | 🎞.mp4 file | ⊠.ttf

* Allows EPUB to open URL in default browser.
† Allows access to LocalStorage of other EPUBs.
‡ Attempts to load all resources in the default application without consent (except fonts).

Table 4.2: Evaluation results for EPUB reading systems on macOS.

very noticeable, since both a visual and auditory cue are given when the call is initiated, it does not require any user interaction. Correspondingly, an attacker could make the user initiate calls to their premium-rate telephone number, e.g. when there has been no user activity for a certain time.

Linux Ubuntu

For the evaluation on the Linux platform, we only found three functioning reading systems, as shown in Table 4.3. Here, Calibre (version 3 and 4) is the only reading system that provides scripting support. Similar to the installations on the other desktop platforms, Calibre 3 uses an outdated browser engine for which a disclosed vulnerability can be exploited to leak arbitrary file contents.

Application	JavaScript		Local Resources			Remote communication	Persistent storage		Features	URI handles	Insecure engine
	Local	Remote	Existence	Render	Leak		Cookies	LocalStorage			
Calibre (3.46)	●	●	</> 🖼 🗋 🗎 ■ 🔊 🎥 ☒	</> 🖼 🗋 🗎 ☒	-	●	○	○	-	●	●
(4.3.0)	●	○	-	-	-	●	○	○	-	○*	○
FBReader (0.12.10)	○	○	-	-	-	○	○	○	-	○	-
Okular (1.7.2)	○	○	-	-	-	○	○	○	-	○	-

</> .html file | 🖼 .png/.jpg file | 🗋 .txt file | 🗎 .log file | ■ .bogus file | 🔊 .mp3 file | 🎥 .mp4 file | ☒ .ttf

* Allows EPUB to open URL in default browser.

Table 4.3: Evaluation results for EPUB reading systems on Linux Ubuntu.

4.5.2 Mobile

In this section, we discuss the results for EPUB reading systems on iOS 12 and Android 9. Note that, to improve legibility, we omitted the browser identification column in the mobile platform tables, since all relied on the engine framework provided by the OS. Consequently, these browser engines are implicitly updated with every system update, thus are considered up-to-date.

iOS

Out of the 20 evaluated iOS reading systems, 11 support JavaScript and allow EPUBs to communicate with servers over the Internet without user consent, as shown in Table 4.4. Only Apple Books requires explicit user interaction to permit the EPUB to communicate remotely, which is then remembered between sessions. Furthermore, two reading systems allow EPUBs to share LocalStorage data, while four reading systems allow it to access the GeoLocation API or enable it to open other applications, which in most cases requires user consent.

Application	JavaScript		Local Resources			Remote communication	Persistent storage		Features	URI handles	Runs in background
	Local	Remote	Existence	Render	Leak		Cookies	LocalStorage			
Aldiko Book Reader (1.1.6)	○	○	-	-	-	○	○	○	-	○	○
Apple Books (4.2.3)	●	●	-	-	-	●*	○	●	-	●†	●
Bluefire Reader (2.9)	○	○	-	-	-	○	○	○	-	○	○
CHMate (6.9.1)	●	●	-	-	-	●	○	●	-	○	●
Ebook Reader (4.0.8)	●	●	-	-	-	●	●	●	-	○	○
Eboox (1.60.1)	○	○	-	-	-	○	○	○	-	○	○
Epub Reader (1.1)	○	○	-	-	-	○	○	○	-	○	○
EPUB Reader (5.1.55)	○	○	-	-	-	○	○	○	-	○	○
FBReader (1.0.10)	○	○	-	-	-	○	○	○	-	○	○
Gerty (1.1.5)	●	●	-	-	-	●	○	●§	-	○	○
Kobo Books (9.14)	●	○	-	-	-	●	○	○	-	○	●
Kybook 3 (0.7.8)	○	○	-	-	-	○	○	○	-	○	○
Marvin (3.1.2)	●	○	-	-	-	●	○	●	Location†	●‡	○
Play Books (5.3.0)	○	○	-	-	-	○	○	○	-	○	○
PocketBook (3.2)	●	●	-	-	-	●	○	○	-	○	○
Power Reader (6.10)	●	●	-	-	-	●	●	●	-	○	○
R2 Reader (2.0.1)	●	○	-	-	-	●	○	○	-	○	○
TotalReader (5.1.61)	●	●	-	-	-	●	○	●	-	●	○
YiBook (1.8.6)	○	○	-	-	-	○	○	○	-	○	○
Yomu (2.3.0)	●	●	-	-	-	●	○	●§	Location†	○	○

[*] Requires user interaction, only the first time.
[†] Requires user consent.
[‡] Requires user constent, except for Mail app.
[§] Allows access to LocalStorage of other EPUBs.

Table 4.4: Evaluation results for EPUB reading systems on iOS.

As a result of the iOS application platform design, the EPUB is isolated from the rest of the file system: the user has to select the EPUB that will be loaded, and the application can only access this particular file. Consequently, access to local resources is blocked by design as these are not available within the application.

Android

In contrast to the iOS reading systems, almost every reading system requests the permission “Photos, media and files on your device”, either upon installation or when attempting to import an EPUB. Interestingly, most also make use of Android’s Storage Access Framework (SAF) [Andb], an API to access user-selected files, which is more constrained but subsequently does not need explicit permissions to facilitate importing EPUBs. However, when SAF is used in combination with the file permissions, which was the case for all but two applications, this does not prevent attacks that leak arbitrary file contents. In fact, for three Android applications we could successfully leak arbitrary file contents to a remote server, as shown in Table 4.5.

Again, the results show that JavaScript support provides additional capabilities that often can be abused. Seven reading systems grant the ability to open other applications, of which only one asks for user permission when this referred application is the browser (i.e. by using the `http` or `https` scheme). Furthermore, out of six applications that support access to the LocalStorage API, five do not provide sufficient isolation, thus allowing access to content saved by other EPUBs.

Application	JavaScript		Local Resources			Remote communication	Persistent storage		Features	URI handles	Runs in background
	Local	Remote	Existence	Render	Leak		Cookies	LocalStorage			
4shared Reader	○	○	-	-	-	○	○	○	-	○	○
AlReader (1.911805270)	○	○	-	-	-	○	○	○	-	○	○
Aldiko Book Reader (3.1)	○	○	-	-	-	○	○	○	-	○	○
Aldiko Classic (3.1.3)	○	○	-	-	-	○	○	○	-	○	○
Bookari Free (4.2.5)	○	○	-	-	-	○	○	○	-	○	○
Book Reader (1.12.12)	○	○	-	-	-	○	○	○	-	○	○
Cool Reader (3.2.32)	○	○	-	-	-	○	○	○	-	○	○
Ebook Reader (1.0)	○	○	-	-	-	○	○	○	-	○	○
Ebook Reader (5.0.8.2)[*]	●	○	-	-	-	●	○	●[§]	-	●	●
EBook Reader (3.5.0)	○	○	-	-	-	○	○	○	-	○	○
eBoox (2.22)	○	○	-	-	-	○	○	○	-	○	○
ePub Reader (2.1.2)[†]	●	●	🌐🖼🗋🗒■🔊🎞🗚	🌐🖼🗋🗒■🔊🎞🗚	-	●	○	○	-	●	●
Epub reader (4.0)	○	○	-	-	-	○	○	○	-	○	○
Epub Reader (librera) (8.0.39)	○	○	-	-	-	○	○	○	-	○	○
EPUBReader (1.0.32)	●	○	🌐🖼🗋🗒■🔊🎞🗚	🔊🎞🗚	-	○	○	●	-	●	○
eReader Prestigio (6.0.0.9)	○	○	-	-	-	○	○	○	-	○	○
FBReader (3.0.15)	○	○	-	-	-	○	○	○	-	○	○
Freda (4.31)	○	○	-	-	-	○	○	○	-	○	○
FullReader (4.1.4)	○	○	-	-	-	○	○	○	-	○	○
Gitden Reader (4.5.3)	●	●	🌐🖼🗋🗒■🔊🎞🗚	🌐🖼🗋🗒■🔊 🗚	🌐🖼🗋🗒■🔊🎞🗚	●	○	●[§]	-	○	●
Google Play Books (5.2.7)	○	○	-	-	-	○	○	○	-	○	○
Infinity Reader (1.7.57)	●	●	-	-	-	●	○	○	-	○	●
iReader (1.1.4)	●	○	🌐🖼🗋🗒■🔊🎞🗚	🖼 🔊🎞🗚	-	●	○	●[§]	-	●	●
Kindle (3.2.0.35)	○	○	-	-	-	○	○	○	-	○	○
Librera (8.1.242)	○	○	-	-	-	○	○	○	-	○	○
Lit Pub (3.5.3)	●	○	🌐🖼🗋🗒■🔊🎞🗚	🌐🖼🗋🗒■🔊🎞	🌐🖼🗋🗒■🔊🎞	●	○	●[§]	-	○	○
Lithium (0.21.1)	●	●	-	-	-	●	○	○	-	●[‖]	●
Moon+ Reader (5.1)	○	○	-	-	-	○	○	○	-	○	○
PocketBook (3.21)[‡]	●	○	-	-	-	○	○	○	-	○	○
Reader FB2 (1.20)	○	○	-	-	-	○	○	○	-	○	○
ReadEra (19.07.28)	○	○	-	-	-	○	○	○	-	○	○
Reasily (1907d)	●	○	🌐🖼🗋🗒■🔊🎞🗚	🌐🖼🗋🗒■🔊🎞🗚	🌐🖼🗋🗒■🔊🎞	●	○	●[§]	-	●	●
Solati Reader (2.5.1)	○	○	-	-	-	○	○	○	-	○	○
Supreader (3.2.30)	●	○	-	-	-	●	○	○	-	●	●
Tolino (4.10.2)	○	○	-	-	-	○	○	○	-	○	○

🌐.html file | 🖼.png/.jpg file | 🗋.txt file | 🗒.log file | ■.bogus file | 🔊.mp3 file | 🎞.mp4 file | 🗚.ttf

[*] Crashes when attempting to render resource on local file system.
[†] Opens all referenced resources in a new frame.
[‡] Only executes inline JavaScript.
[§] Allows access to LocalStorage of other EPUBs.
[‖] Requests user consent only when browser is launched.

Table 4.5: Evaluation results for EPUB reading systems on Android.

4.5.3 Browser extensions

For both Chrome and Firefox, we evaluated five extensions that are advertised as EPUB reading systems. As shown in Table 4.6, three Chrome extensions allow JavaScript execution, giving them access to persistent storage, and even to the microphone, camera or location (provided that the user consents). Because all EPUBs opened by these applications shared the same origin (chrome-extension://[extension_id]), EPUBs can access the persistent storage of e-books that were opened previously.

Application	JavaScript		Local Resources			Remote communication	Persistent storage		Features	URI handles
	Local	Remote	Existence	Render	Leak		Cookies	LocalStorage		
Chrome extensions (3)	●[*]	○	-	-	-	●	●[†]	●[†]	Microphone, Camera, Location[‡]	●[‡]
Chrome extensions (2)	○	○	-	-	-	●	○	○	-	●[‖]
Firefox extensions (5)	○	○	-	-	-	●	○	○	-	●[‖]

[*] Only external JavaScript.
[†] Can access cookies and LocalStorage of other EPUBs.
[‡] Requires user consent.
[‖] Requires user click, and only works for `mailto:`.

Table 4.6: Evaluation results for EPUB reading extensions for Chrome

The remaining seven extensions do not allow JavaScript execution as a result of the imposed CSP, prohibiting all inline JavaScript and only allowing resources local to the extension [Moze]. Although remote resources are blocked by the CSP, the extensions still provide functionality to fetch these (hence the ability to perform remote communication). In Section 4.6 we show how this functionality lead to a universal XSS in EPUBReader (available on both Chrome and Firefox). This was also the only extension that automatically rendered an EPUB when a referring link is clicked (achievable through JavaScript) in Chrome.

4.5.4 Physical e-reader devices

Only for the Kobo e-reader, our testbed confirmed limited JavaScript support, as is documented by Kobo [Kob]. Note that the e-reader only executes embedded scripts for KEPUB files (Kobo's custom e-book format), these are created by simply changing the `.epub` file extension to `.kepub.epub`. Furthermore, we could use the e-reader's internet connection to contact remote servers without user consent. Finally, the embedded browser engine framework was identified as QT 5.2.1 (released in 2014) for which several vulnerabilities have been reported [CVE].

Amazon's publishing guidelines affirm that scripting is not supported, and that all scripts are stripped from the source during conversion [Ama19]. However, as part of a manual evaluation of the Kindle, we found this to be inaccurate: the browser engine supports JavaScript execution, although it is disabled by default (in Section 4.6.3 we show how this can be circumvented).

4.6 Case studies

To complement our semi-automated evaluation, we manually analyzed a select number of applications for implementational flaws. This selection was based on different characteristics: Apple Books on macOS (popular pre-installed application that supports JavaScript but prevents rendering local files), EPUBReader (the most widely used browser extension on both Chrome and Firefox), and Kindle (the most widely used physical e-reader, with an 83.6% market share in the US [DG19]).

4.6.1 Apple Books

As we discussed in Section 4.5.1, the capabilities detected by our semi-automated evaluation did not lead to direct local file system access in Apple Books for macOS, even when leveraging symbolic links. However, through manual evaluation, we identified a user information disclosure vulnerability and persistent denial of service vulnerability.

The *user information disclosure* vulnerability allows an attacker to infer whether a specific EPUB is present in the user's library. When an EPUB is opened by Apple Books, it is unpacked and stored in a folder (`Books`) alongside other previously unpacked EPUBs. The contents of each EPUB are stored in a separate folder named after the EPUB's deterministically assigned 32-character serial ID. While we could not infer how this ID is generated exactly, we have verified that an EPUB is assigned the same ID across multiple devices or accounts. Although embedded symbolic links referring outside of this directory are denied, these links would still remain functional when pointing to a location within the EPUB folder, or even within the `Books` directory. As a result, to gain information about the contents of the user library, an EPUB could include a series of symbolic links, referring to potential locations of unpacked EPUBs. By verifying whether an arbitrary file in such a folder can be rendered, the EPUB can disclose whether any of the selected EPUBs is present in the user's library. We found that the iOS applications Gerty and Marvin could be exploited in a similar way.

The *persistent denial of service* attack is achieved by simply including a symbolic link that refers to the `Books` folder in which the EPUBs are unpacked. This will cause Apple Books to crash, reporting that it cannot access the user's library, for every subsequent reboot. Because Apple Books is an integral part of the operating system, it cannot be reinstalled without reinstalling macOS.

In response, Apple issued a CVE for both vulnerabilities (CVE-2019-8789 and CVE-2019-8774, respectively), and distributed a fix through operating system updates [App20a; App20b; App20c; App20d].

4.6.2 EPUBReader extension

The results of our semi-automated evaluation in Section 4.5.3 show that all Firefox extensions and two Chrome extensions block JavaScript execution, due to the imposed CSP. Even more interesting; upon installation, three extensions request permission to read and change all data of visited websites, using the `<all_urls>` permission indicator [Chra]. This allows the extensions to send

HTTP requests to any visited website and read out the response. Moreover, if the user is logged in on such a website, the request will implicitly include session cookies, and thus authenticating the user. By bypassing the CSP restrictions for embedded JavaScript in EPUBReader for both Chrome and Firefox, we were able to abuse this permission to steal user account information of any website on which the user is logged in, effectively leading to a universal XSS.

Although including remote resources directly is prevented by EPUBReader's CSP, it still tries to provide this functionality by first fetching the included images and referred media files, and then making their content available through a `blob://` URL (which is allowed by the default CSP). We leveraged this functionality to trick EPUBReader to make a JavaScript file available as a `blob://` URL (by simply including this file as an image). Because these URLs contain an unguessable UUID, we first used a CSS-based data exfiltration technique to leak this to the attacker server. Finally, the adversary can dynamically generate another EPUB that refers to this `blob://` URL, and then tricks the reader to open this generated EPUB (EPUBReader will automatically try to read EPUBs based on the URL pattern). Finally, the JavaScript payload will be executed, giving the attacker the same privileges as the browser extension (access to all authenticated content on all websites). A proof-of-concept implementation of our attack requires a single user interaction, such as a click, from the victim on Chrome in order to open a new window; on Firefox the attack can be performed without any user interaction, and is unnoticeable. Collectively, this affects almost 300,000 users.

4.6.3 Kindle

Our semi-automated evaluation indicated that Kindle does not support JavaScript execution, as confirmed by the publishing guidelines [Ama19]. By reverse engineering the application that renders the converted AZW3 files (`webreader`), we found that WebKit 534.26 is used to render e-books, but that in the browser engine's settings, the `enable-scripts` property is set to `false`. However, the application itself uses JavaScript to extract information from the DOM or to change styles.

Before each JS execution, the `enable-scripts` property is set to `true`, and immediately after it is set back to `false`. Consequently, JavaScript code contained in the EPUB will never be executed. Nevertheless, we found that some of the code that was being executed by the application contained dynamic values, some of which could be controlled by an attacker. For instance, a script that is executed on every page refresh includes the font that is used. This value can be controlled by the attacker by changing the font, which is done by sending a GET request to the webreader's SOAP server. Although the rendering engine also blocks web requests, this can be circumvented by leveraging SVGs. Presumably these are rendered outside of the browser engine, and thus an `<image>` element with the `xlink:href` attribute will trigger a request. Consequently, it is possible to set the font to an arbitrary value. When the font contains a single quote, it can escape the string context of the dynamic code and execute arbitrary code.[6]

Once the attacker can execute JavaScript, it is possible to extract the contents of arbitrary files on the system by leveraging the built-in image viewer. This viewer can be automatically triggered by sending a GET request to the webreader's SOAP server, containing the URL of the image that should be displayed. This viewer uses the native `file_get_contents()` function to get the URL's content and then base64-encodes it to embed it in an `<img>` element's `data:` URL. As such, the content of any arbitrary file (not just images) can be be extracted, and subsequently leaked to the adversary by initiating a WebSocket connection, which also is not blocked by the rendering engine.

4.7 Real-world analysis

In this section, we analyze the EPUB ecosystem by assessing the presence of malicious and tracking EPUBs in the wild, and the feasibility of distributing them through a self-publishing service.

4.7.1 Malicious and tracking EPUBs in the wild

In order to investigate whether any of the discussed techniques are currently being used in the wild to either attack or track users, we performed an additional analysis of EPUBs available in a real-world setting. To this end, we downloaded several free EPUBs from five popular online e-book stores (eBooks.com, Google Play Books, Project Gutenberg, Kobo, Amazon). After a manual inspection of the e-books, we did not find any indication of abuse. It should be noted however that this evaluation is limited, and only considers abuse by the EPUB stores; abuse by individual publishers would be infeasible to evaluate from an external perspective, as this would require purchasing a very large number of e-books.

To further evaluate other types of abuse, we obtained a large number of EPUBs from file sharing platforms. More precisely, we downloaded the 1,000 most

[6] Because the length of the font name is limited, the malicious payload has to use `eval()` on the `textContent` of a (hidden) DOM element.

popular and most recent EPUB torrents from The Pirate Bay and the same amount from 4shared (these are marked as the most widely used sources to illegally obtain an e-book according to a study by Digimarc [Dig17]). In total, we obtained 7,238 EPUB files from torrents (in several cases, a single torrent contained multiple EPUBs), and 1,807 from 4shared. Next, we unpacked all EPUBs and parsed all documents, looking for possible types of abuse (references to files on the local file system, symbolic links, connections to a remote server, and JavaScript inclusions). We did not find any evidence of abuse, either in terms of tracking or attempts to compromise the EPUB reading system. Interestingly, we found that only 65 e-books, less than one percent of all 7,238 considered EPUBs, made use of JavaScript. In most cases, the code was minimal, and was used to change the background color or font size. All e-books were completely functional without executing the JavaScript code.

4.7.2 Malicious EPUB distribution through self-publishing

To explore the feasibility of publishing malicious EPUBs through official e-book vendors, we submitted manuscripts to the six most popular free self-publishing services. For each service, we bought the published version of our manuscript to check whether any of the embedded scripts were still present. The following is a list of these services, along with their associated vendor and its e-book market share according to the 2017 AuthorEarnings report [Aut17]: Kindle Direct Publishing (Amazon, 80%), iBooks Author (Apple Books, 10%), Barnes & Noble Press (Barnes & Noble, 3%), Kobo Writing Life (Kobo, 2%), and Google Books Partner Centre (Google Play Books, 1.4%). No exact figures were available for the sixth tested service, Smashwords, however they also distribute self-published titles through Apple Books, Barnes & Noble, and Kobo in addition to their own website [Sma19].

Of the six vendors, only Google Play Books rejected our submitted manuscript. Although Amazon succeeded in removing most scripts, we were still able to publish the exploit discussed in Section 4.6.3, which could target millions of Kindle devices. The remaining four vendors appeared to take no vetting measures at all; embedding scripts in a published EPUB was trivial. Of these, only Smashwords provides downloadable EPUB files upon purchase, hence, any EPUB reading system can be used to open them. The other three vendors deliver e-books directly to their associated reading systems (however, this can be circumvented). For Apple Books (application) and Kobo (application and physical e-reader), the embedded scripts were executed, and even allowed remote communication. However, Barnes & Noble's application crashed upon rendering embedded scripts, curbing potential abuse.

In conclusion, our experiment shows that four out of six evaluated self-publishing services can be abused to distribute malicious EPUBs through official vendors. These vendors account for approximately 94% of all EPUB sales, of which at least 33% is attributed to self-published EPUBs according to the 2017 AuthorEarnings report [Aut17]. We notified all five self-publishing services of which the vetting process was deemed inadequate.

4.8 Discussion

Our semi-automated evaluation shows that many of the JavaScript-supporting EPUB reading systems do not correctly enforce the specification's security recommendations, and thus can be abused in several ways. Furthermore, a significant part of these reading systems does not prevent EPUBs from accessing the local file system and even provide JavaScript APIs that are not included in the EPUB specification. In this section, we elaborate on the underlying issues and make suggestions on what can be improved to remedy the various issues.

4.8.1 EPUB reading system implementations

In contrast to mobile reading systems, we identified a high variety of rendering engines for desktop reading systems. Moreover, we find that five of the evaluated desktop applications employ an outdated engine, and consequently, a publicly disclosed vulnerability could be leveraged to exploit the application. Even applications that employ a more up-to-date engine may still be affected by so-called n-day vulnerabilities [Cui18; Wan^{+}19], security issues that have been patched in the upstream component (and thus known publicly), but that still affect software that did not yet update this vulnerable component. As it may take days, or even years (e.g. in the case of Calibre 3) to update a known-vulnerable browser engine, we believe this forms a significant threat for EPUB reading systems.

For both mobile platforms, we found that applications relied on the built-in renderer, and thus all share the same version. Another key difference with desktop applications is that mobile reading systems operate from a more sandboxed environment by default. For instance, on iOS, none of the applications requested permission to access other files on the system, and consequently could not be abused to render or leak files on the local system. Although a similar functionality is available on Android, through the Storage Access Framework [Andb], most applications still required file permissions, and as a result, we managed to detect the existence of files in six applications, and leak their contents in half of those. This highlights that developers should try to use the minimal amount of privileges to reduce the potential consequences of an attack. By further analyzing the Android applications, we found that for two the file-leak vulnerability was caused by configuring the WebView component to allow access to the local file system, using `setAllowFileAccessFromFileURLs` [Andc].

Although several applications would render files on the local filesystem, not all of them lead to extraction of their contents. Our manual analysis of these cases showed a direct relation to SOP enforcement: the EPUB content was served from a custom, non-existent domain, preventing access to `file://` resources. Yet, not all reading systems implementing this practice were able to achieve complete isolation of the local file system. For instance, Adobe Digital Editions on Windows employs a dedicated domain, but EPUBs are still allowed to render local images or even HTML files on network shares. The latter is especially dangerous, as it gives access to the `file://` from where the local filesystem can be accessed.

4.8.2 EPUB specification

Although a valued effort has been made to include effective security recommendations, we argue that the EPUB specification does not impose sufficiently strict requirements for EPUB reading systems. Of course, the responsibility to actually conform their reading system to the specification's security requirements remains that of the developers, however, hardened requirements could eventually be consolidated into a quantified compliance checker application.

Probably even more effective would be attenuating the capabilities that are to be granted according to the EPUB specification. For instance, an EPUB is allowed to only refer to audio, video and fonts through static XHTML and CSS, but any resource is allowed to be retrieved by embedded scripts [CG19]. This can be useful for keeping the size of an EPUB small, since the more sizable audio and video files can be fetched from an online service. However, access to resources from the local filesystem, which in the current version of the specification is allowed, introduces a significant threat, which does not outweigh its limited benefits. Furthermore, the ability to render local resources implies the ability to determine their existence, information that can be gained for various purposes among which file system fingerprinting. For this reason, we argue to completely prohibit EPUBs from referring to resources that reside on the user's operating system. Moreover, as reference to remote resources is very rare in EPUBs, we strongly believe that this should require consent from the user, in order to prevent any form of tracking.

Interestingly, our semi-automated evaluation revealed that more than half of the JavaScript supporting reading systems also support GeoLocation and UserMedia APIs, or opening applications through URI handles, functionalities that are not mentioned in the EPUB specification. These functionalities originate from the underlying browser engine, and are likely not considered by the developer. Assuming the EPUB specification does not aspire to incorporate such browser functionalities, we argue that the specification should include a whitelist of APIs that can be enabled.

Based on our real-world analysis of 9,000 EPUBs, we argue that the discussed restrictions for the EPUB specification would have a minimal impact; none of the analyzed EPUBs required local or remote resources to render correctly, and even the few that embedded JavaScript remained functional when execution was prevented. In that regard, we also propose to reconsider the capability of unrestricted JavaScript execution in EPUB reading systems, perhaps requiring user consent when a script is about to be executed.

4.8.3 Responsible disclosure

All vulnerabilities, either identified through our semi-automated testbed or our case-studies, were responsibly disclosed to the involved parties. In addition, we sent out an early warning to all vendors whose reading system did not satisfy the specification's security recommendations. In total, we reached out to 33 vendors, responsible for 37 reading systems, each time using the most appropriate private channel that was available. Although we received a generic or no response from the majority, vendors of very popular reading systems such as Apple and Adobe were eager to solve the reported issues, for which three CVEs were issued.

4.9 Related work

We did not encounter any prior studies evaluating the implications of web technology use in non-browser applications. However, our work shares several similarities with the following research.

4.9.1 Portable Document Format

Today, PDF is one of the most popular file formats used for operating system independent document exchange. Its capabilities bear close resemblance to those of the EPUB format, including support for scripting and network connectivity. Unfortunately, previous research has demonstrated that these traits may to lead to security, privacy and content integrity vulnerabilities [BS12; CSS10; Mla+18]. In that regard, we hope that by expressing our concerns about EPUB capabilities at an early stage, the specification can be adapted to avoid similar consequences.

Various research efforts focus on the use of machine learning to distinguish between benign and malicious PDF files. Research by Smutz et al. and Srndic et al. argue that PDF file metadata and structure are valuable features that can be used by a static, machine learning based detection system [SL13; SS12]. Maiorca et al. demonstrated a new evasion technique for PDF file analysis based on logical structure; they also present a framework to solve this problem [MCG13]. Nissim et al. performed an extensive study reviewing and comparing state-of-the-art techniques for detecting malicious PDF files [Nis+15].

4.9.2 Comprehensive policy evaluations

Various studies have exposed vulnerabilities and inconsistencies in browser policy implementations through comprehensive evaluations. By combining manual and automated analysis in four popular browsers, Aggarwal et al. uncovered several implementational weaknesses for private browsing modes [Agg+10]. Furthermore, they show that some of these weaknesses can be exploited by an attacker to bypass the imposed privacy policy. Schwenk at al., on the other hand, performed a comprehensive evaluation of the same-origin policy in 10 browsers, leveraging an extensive set of 544 different test cases [SNM17]. Their results exposed various vulnerabilities and inconsistencies among browsers, pleading for a formal definition of the same-origin policy. In another study, Franken et al. performed an evaluation of third-party cookie policy implementations in 7 browsers and 46 browser extensions, leveraging their automated framework [FVJ18]. According to their results, all imposed third-party cookie policies of all major browsers as well as evaluated extensions can be bypassed. Finally, a longitudinal study by Luo et al., comprising of 20 different mobile browser families, analyzed support for eight different security mechanisms over the course of seven years [Luo+19]. Their findings expose various issues such as lacking support and multi-year vulnerability issues, even for several popular mobile browsers.

4.10 Conclusion

In this chapter we report on a semi-automated evaluation to measure the security and privacy practices of 92 free EPUB reading systems and five physical reading devices. Our results show that almost none of the systems that support JavaScript execution adequately adhere to the security considerations of the EPUB specification. For eight reading systems, a malicious EPUB can even extract arbitrary files from the local system.

We are the first to comprehensively evaluate the security and privacy practices of EPUB reading systems, and hope to increase awareness of the associated threat surface among users and developers. Furthermore, we propose that the current security recommendations of the EPUB standard should be refined into binding requirements. To further assist developers, additional documentation could be provided in more specific terms how existing browser engine frameworks can be correctly incorporated, pointing out critical configuration elements.

In addition to this large-scale evaluation, we also performed a more elaborate manual analysis of a select number of EPUB reading systems. This manual analysis exposed two severe security issues: first, as soon as a malicious EPUB would be opened in Kindle, it could leak arbitrary files from the local system; second, the entire browsing session of users with the EPUBReader browser extension can be compromised upon visiting a malicious website. The results also highlight that the outcome of our semi-automated evaluation should be considered a lower bound, and that several security and privacy issues rely on application-specific behavior.

As part of our assessment of the EPUB ecosystem, we performed an analysis of more than 9,000 EPUBs, obtained “in the wild” from five online e-book stores and two popular file sharing platforms, and evaluated the vetting process of six popular self-publishing services. We did not find evidence of any ongoing abuse, indicating that the issues identified through our evaluations are indeed novel. However, this and the fact that four of the evaluated self-publishing services allowed JavaScript inclusion which could lead to publication of malicious EPUBs, make this study timely: we urge developers to further mitigate the identified issues and adopt additional security measures before their users are exploited.

Finally, we demonstrated that the consolidation of established web technologies in non-browser applications does not necessarily imply a proper translation of the web security and privacy primitives. With this study, we hope to have motivated the need for more comprehensive and in-depth evaluations in this largely unexplored research domain.

5

Conclusion

The right understanding of any matter and a misunderstanding of the same matter do not wholly exclude each other.

– Franz Kafka [Kaf25]
(The Trial, 1925)

Browser policies serve as the final barrier against potential security and privacy breaches on the Web. Unfortunately, even long-standing policies that have been in place for years exhibit critical vulnerabilities that make them fall short of their intended purpose, with adverse effects on user security and privacy. In this dissertation, we have identified numerous implementation shortcomings by leveraging automated dynamic testing. In doing so, we have unveiled their root causes and various handling flaws, supported by empirical data. Moreover, our research demonstrates that these issues also impact non-browser applications that incorporate web technology.

In this concluding chapter, we will summarize the contributions presented in this dissertation, outline potential areas for future research and explore strategies for enhancing current browser engine development and deployment practices. Ultimately, we will share our final thoughts on the future of browser engine security and privacy.

5.1 Summary of contributions

In Chapter 2, we presented the first systematic and comprehensive study of policies governing the most fundamental pillars of the Web: HTTP cookies and requests. Through the development of a novel framework for dynamic policy implementation analysis, we identified a multitude of issues for the employment of cross-site countermeasures and anti-tracking policies in popular web browsers. Initially introduced as a safeguard against CSRF attacks, same-site cookies were discovered to be implemented inconsistently which allowed a bypass in various web browsers. Remarkably, even the built-in setting to simply block all third-party cookies proved to be ineffective in Chromium-based browsers, as it could be circumvented by embedding JavaScript in a loaded PDF file. Moreover, virtually none of the assessed privacy measures succeeded in fulfilling their intended purpose of providing comprehensive protection against tracking. Our real-world investigation found no evidence of active exploitation of these bypasses in the wild, allowing developers to address the issues before any potential exploitation could occur. Although our follow-up study recognized efforts to tackle the these issues, developers remained unable to fully eliminate all previously identified workarounds against cross-site countermeasures and anti-tracking policies, thereby leaving these protective measures ineffective. This serves as a clear exhibition of the challenges inherent in achieving a comprehensive policy implementation.

Building upon the foundational work discussed in Chapter 2, we directed our focus toward identifying the underlying causes of implementation flaws within browser policies in Chapter 3. To achieve this, we conducted an extensive analysis of the entire lifecycle of nearly all known CSP bugs at the time of our study, with CSP being one of the oldest and most important browser security policies. We created the `BugHog` framework to facilitate this analysis, which is capable of conducting dynamic evaluations on Chromium and Firefox revision binaries spanning over a decade, encompassing CSP's entire development history. As such, `BugHog` proves to be an invaluable tool in the process of identifying code revisions responsible for introducing or resolving policy bugs by automating the majority of the steps involved. Aided by `BugHog`, our examination of lifecycles, bug reports and code revisions exposed recurring issues that contributed to the origin of policy bugs. Based on our results, we emphasized the crucial role of browser architecture in preventing policy bugs, with the centralization of policy logic and enforcement as a key factor. However, our study revealed that the root causes of policy bugs extend beyond policy implementation as well, where human mistakes and ineffective protocols within the context of bug handling practices led to the public disclosure of unresolved security bugs. In that same light, we urged for a more effective communication channel between browser vendors for sharing security bugs, since we found several cases in which bugs were reported, fixed and publicly disclosed through the bug tracking platform of one vendor, but remained unresolved in browsers of other vendors, leaving users vulnerable for extended periods of time.

While many native applications across various platforms incorporate browser engines, this class of applications was never subjected to a comprehensive and in-depth security and privacy analysis. In Chapter 4, we addressed this gap by conducting an evaluation of the security and privacy implications of EPUB reading systems, revealing that they share numerous issues with web browsers. We pinpointed several vulnerability causes, including the use of outdated browser engines, the insecure configuration of browser engines and overly permissive capabilities for loaded EPUBs. Furthermore, we demonstrated that EPUBs exploiting these vulnerabilities can be distributed by adversaries using the self-publishing services of popular online bookstores, such as Amazon's Kindle Store, Apple Books and Rakuten Kobo. Several of the identified issues stemmed from shortcomings within the EPUB specification, giving rise to security and privacy concerns. As part of our effort to improve overall user security and privacy, we not only released our semi-automated testbed but also expanded the W3C test suite used by their official EPUB compliance checker. Furthermore, in the forthcoming iteration of the EPUB specification, significant improvements have been made. Among these enhancements, the most critical is the explicit requirement to prevent EPUBs from accessing the local file system, marking a step in the right direction.

The work presented in this dissertation uncovered numerous security and privacy bugs, which were all responsibly disclosed to the affected parties. All frameworks and testbeds have been published as open-source, allowing for further research and development.

5.2 Future work

In the house of research, every opened door ushers us into a vast corridor filled with yet unopened doors. The work presented in this dissertation is no exception, and as such, numerous exciting directions for future research remain to be explored.

5.2.1 Comprehensive implementation verification

Verifying the robustness of a security policy implementation through dynamic evaluation can be regarded as an optimization problem. The ultimate objective is to find the minimal set of test cases that covers all possible browser behaviors for which the policy should interfere. As previously discussed in Section 1.5.1, this comprehensive approach necessitates considering all possible policy configurations on one hand and all possible browser behaviors that can be triggered by a web page (e.g. through the use of HTML elements or JavaScript) on the other hand. Nevertheless, even when filtering out irrelevant combinations, this would result in an explosion of test cases, leaving the task of verifying the correctness of the policy's interference for each test case.

In previous research, testbeds have been constructed using a variety of approaches. These include repurposing existing unit tests [Agg$^+$10], employing template-based generation [HMN15; Luo$^+$19; RPS23; SNM17], leveraging real-world deployment [Sin$^+$10], manual curation [Luo$^+$17] and grammar-based generation [Wi$^+$23]. Methodologies for testbed creation are influenced by the specific type of policy being evaluated. As such, to ensure a comprehensive evaluation of unexplored policies, it might be necessary to develop novel methodologies tailored to those policies, as there is currently no established policy-agnostic approach available to address this need. Additionally, there is an opportunity to enhance existing methodologies, such as reducing the number of test cases by applying heuristics to target the most relevant test cases. Or, to expand the existing collection of test cases, automated methods involving targeted permutations can be employed. These methods could involve altering embedded browsing contexts or modifying attribute values of HTML elements.

Once all test cases are executed, all outcomes should be evaluated to determine whether the policy's interference is correct or not. The chosen methodology for generation is crucial here, as it determines the type of outcomes that are collected. For example, if only test cases in which a policy should interfere are generated or labeled, we would already know where the absence of interference is a bug [Agg$^+$10; HMN15]. This assessment can be conducted manually, provided the number of test cases is manageable. Another approach is to identify inconsistencies between different web browsers and consider these as indications of faulty behavior [SNM17; Wi$^+$23] or employ machine learning [RPS23]. Other directions could be taken such as considering inconsistencies observed over time in a longitude analysis to aid in the determination of the correctness of the policy's interference. This approach could enhance the efficiency and accuracy of the evaluation process, particularly when dealing with a large volume of test cases or evolving web environments.

Note that comprehensiveness could also be found in the form of formal verification [Akh+10; Ban+14; FKS16; FKS17; JTL12]. However, the manual effort required to create a formal model of browser behavior is substantial, especially when considering that the Web is constantly evolving.

5.2.2 Standardized language for bug reporting

Currently, the templates for reporting security bugs on browser vendor tracking platforms lack standardization. These platforms typically provide loosely defined templates that can be filled in by the reporter, including for example a freeform description on how to reproduce the bug. This leads to significant variability in the quality of bug reports, making it challenging at times to reproduce the reported issues. In some instances, reproduction is even rendered impossible due to link rot. This occurs when the bug is inadequately described in the report and the linked PoC residing on an external web page is no longer available. This situation poses problems both for research that relies on these bug reports and for browser vendors who may wish to revisit a bug report at a later stage.

Standardizing bug reporting templates and emphasizing the significance of comprehensive and resilient bug descriptions could serve as effective solutions to address these issues. The difficulty lies in finding a balance where the standard is flexible enough to accommodate different types of bugs while maintaining a level of strictness that ensures all relevant information is consistently included. This language could be either used by reporters or by triagers to describe the bug, or both. Moreover, the adoption of a standardized language could bring additional benefits, including improved bug categorization, which is currently futilely attempted through inconsistent labeling practices. It could also improve the exchange of bug information between vendors and researchers, and support the automated creation of PoCs. These PoCs could then be integrated into automated testing frameworks, enhancing the overall robustness of security evaluations and bug assessments.

5.2.3 BugHog

The `BugHog` framework is a powerful tool for analyzing the lifecycles of browser bugs, but there is always room for improvement. Firstly, it is important to note that the current version only supports the evaluation of Chromium and Firefox, and it would be valuable to expand its capabilities to include other popular browsers. Specifically, incorporating the open-source WebKit browser engine would be a significant enhancement, as WebKit serves as the foundation for Safari, enabling a comprehensive analysis of the three major browser families.

Since our study discussed in Chapter 3 revealed architectural differences that influenced the security of one browser over the other, this addition could provide us with more valuable insights. Furthermore, certain browsers introduce unique, browser-specific features that are not inherent to the embedded browser engine, and thus, evaluating browsers within the same browser engine family may be warranted. In a similar vein, mobile variants also present an interesting avenue for exploration, as they are known to display distinct behavior in comparison to their desktop counterparts [Luo+19]. Nevertheless, note that the transparency of browser vendors is crucial here, as we rely on the availability of source code and bug reports. For instance, while security bugs in Chromium and Firefox are publicly disclosed by default after a certain period of time, this is not the case for Safari unfortunately.

Secondly, the process of identifying bug lifecycles still involves a number of manual steps. One significant challenge stems from the reliance on the limited set of publicly available revision binaries, published by browser vendors. In cases where required binaries are not publicly available, researchers may have to manually sift through a limited range of revisions to locate the relevant revision. The current solution involves the reseacher building a select set of revision binaries to alleviate the manual workload, but this approach demands a substantial amount of storage space and computing power, and is not always straightforward due to the intricacies of the build process. Additionally, it is worth noting that browser vendors likely build more binaries than they make available online, so having access to these unpublished binaries would be highly advantageous. To reduce manual effort, alternative methods can be explored as well, such as the use of heuristics to pinpoint the most probable revision of interest within a specific range. This method allows for a more structured manual review process, where revisions are assessed in order of likelihood, ideally reducing the number of revisions that require manual inspection. Algorithms like SZZ [ŚZZ05] could be employed to implement such heuristics effectively.

The foregoing of this section has highlighted just two of the numerous potential enhancements that could be made to the `BugHog` framework. There are several other avenues for improvement, including the integration of existing test suites like WPT. This addition could expand the framework's capabilities, allowing it to consider a wider range of test outcomes and assess browser behavior more comprehensively. Furthermore, to expand its test surface, the framework could incorporate functionality for evaluating file system state to identify potential side-effects at OS level. Additionally, it could examine browser state from a broader perspective by also considering elements such as LocalStorage [Mozh] and IndexedDB [Mozg].

5.2.4 Browser engines in non-browser applications

Reusing browser engines in non-browser applications is a widespread practice, enabling developers to harness their tried and tested rendering capabilities. Nevertheless, as detailed in Chapter 4, this approach can give rise to security and privacy concerns. It is important to note that our focus in this chapter has been specifically on the incorporation of browser engines in EPUB reading systems, but this practice extends beyond this particular class of native applications. It would be valuable to explore the security and privacy implications in other categories of native applications that incorporate browser engines, such as messaging apps, office software and even games.

In particular, an interesting area of investigation lies in the analysis of the security and privacy ramifications of the deployment of outdated browser engines in non-browser software. For instance, consider the most recent release of Microsoft Teams, which, at the time of writing, is version 1.6 and embeds Electron 19.1.8 [Mic]. This release of Electron is severely outdated and has been without eligibility for security fixes for already a year [Git22].[1] Additionally, investigating the integration of web browsers by so-called “smart” devices like smart TVs, smartwatches, smart speakers and even smart fridges holds promise for further insights. For instance, managing timely browser engine updates on these devices may pose unique challenges, as prior research indicated that these devices are more prone to employ outdated software components compared to desktop and mobile applications [PXH22].

In numerous non-browser applications, a large set of unnecessary browser features is inherited from the embedded browser engine, often unbeknownst to the developer. This unnecessarily expands the attack surface of the application, potentially leaving room for exploitation by malicious actors. For instance, in the context of EPUB reading systems, features like the ability for a loaded EPUB to open external applications are completely unnecessary, yet they are often included through the embedded browser engine. Therefore, exploring the feasibility of developing a lightweight browser engine explicitly tailored for non-browser applications would be an interesting research direction. To make it adaptable for diverse applications, this engine could be designed with a modular architecture, allowing developers to cherry-pick and include only the specific functionalities relevant to their use cases. By default, it could feature a minimal renderer, akin to the simplicity of web browsers from the 1990s, thereby significantly reducing the attack surface.

[1] Electron backports security fixes to its latest three stable major versions [Elea]. At the time of writing, Electron’s latest stable major version is 27, and consequently, security updates only extend to versions 26 and 25.

5.3 Thoughts on development and deployment

In this section, we outline our thoughts on how browser engine development and deployment can be improved. While these recommendations may align with future research directions to some extent, our primary emphasis here lies with actionable steps that draw upon the expertise and resources of vendors.

5.3.1 Code base

One crucial takeaway of this dissertation is the intricate challenge of maintaining a code base comprising millions of code lines that has to keep up with the ever-evolving collection of web mechanisms and policies. As discussed in Chapter 1, the architecture of this code base was not originally designed with the current state of the Web in mind. Browser developers in the 1990s could not have predicted all features that would be incrementally incorporated into browsers. And while indeed the core of browsers might have changed over the years too, some artifacts of the past such as component architectures and design decisions still remain. In numerous instances, particularly in the context of policy enforcement, these remnants present a difficult challenge when it comes to aligning them with newly introduced functionality. It seems evident that indeed browser vendors favor the pragmatic approach of building upon existing code, rather than developing a browser entirely from scratch. However, does starting with a clean slate pose as a viable solution for a more durable browser?

When examining this issue from an economic standpoint, it becomes obvious that a substantial investment of resources is required. By using the Constructive Cost Model (COCOMO) we can gauge the cost of developing a browser without forking from an existing code base [Boe81].[2] COCOMO relies on the number of lines of code to estimate the development costs of a software project. In Table 5.1, we compare the estimated development costs of Chromium and Firefox with those of other major open-source projects. Although there is an expected margin of error in this estimation, it nonetheless underscores that the cost of building a browser from scratch is comparable to that of other significant open-source projects, necessitating a substantial allocation of resources.

Even if an organization would be prepared to commit resources of such a substantial magnitude, other challenges remain to overcome. Perhaps the most critical question is: How can we ensure that this new browser does not succumb

[2] COCOMO is considered outdated, but for the sake of simplicity, we employ it as a comparative baseline. Although numerous other models exist, they would require additional data that is not readily available. In this calculation, we applied a coefficient of $a = 2.4$ and an exponent of $b = 1.05$.

Project	Lines of code*	Person-years†	Cost‡
Apache HTTP Server	1,658,736	481	$178M
GNU Compiler Collection	9,991,490	3,116	$1,153M
Android	14,606,292	4,640	$1,717M
Firefox	**28,049,384**	**9,139**	**$3,381M**
Chromium	**28,528,607**	**9,275**	**$3,432M**
Linux Kernel	34,412,146	11,625	$4,391M

* All values were collected from Black Duck Open Hub: `https://openhub.net/`.

† Represents the amount of time an individual would need to write the specified number of lines of code during regular working hours.

‡ We employ an average salary of $370K, conform with the wages of senior software engineers working for Google and Mozilla [leva; levb]. Note that this is a conservative estimate, as it does not include additional costs such as taxes.

Table 5.1: Estimated development costs of major open-source projects.

to the same issues as the current ones? Here, a well-thought-out architecture and design of the new browser are paramount, and should be based on the lessons learned from current issues. But even if this new architecture and design could seamlessly address all the issues that have been identified and enable a more straightforward way of comprehensively incorporating existing features and policies, the ever-evolving nature of the Web presents a persistent challenge. Much like the developers of the 1990s, the developers of the new browser would not be able to predict all features that would be incrementally incorporated into the Web. Furthermore, a policy-centric approach might impede performance and usability, factors widely regarded as crucial for the success of a browser.

Perhaps solutions can be found in reshaping the very bedrock of the Web itself: the specifications and standards governing both mechanisms and policies. Again, however, these long-term solutions would require a significant amount of resources, in addition the development costs of a browser that is compatible with these new specifications and standards. On top of that, a collaboration between a multitude of stakeholders deeply entrenched in the Web's ecosystem is vital for this to succeed, spanning from browser vendors on one end to web services on the other end. This is unlikely to materialize in the near future, due to a lack of intrinsic motivation. Even with the anticipated transition to Web 3.0, the foundational user-facing elements currently in use, such as HTML, CSS, and JavaScript, are likely to persist.

The perspective presented above might seem to lean towards a pessimistic view, given the inability to discern a practical and durable solution to the identified issues. Indeed, it is probable that without a change of course, the incorporation of new features will persist as a complex challenge that is prone to errors. However, it is equally important to recognize the extensive efforts dedicated to enhancing the foundational architecture and design of web browsers. For instance, in Chapter 3, we discussed the centralization of policy logic and enforcement as a critical step in preventing policy-related bugs. Such a measure was successfully implemented in Chromium through the Policy Container, effectively addressing numerous issues related to policy inheritance [Chr20b]. In this context, adopting a pragmatic stance might be the most realistic approach, as it allows for the gradual improvement of the current state of affairs. Nevertheless, such efforts do come with associated costs as well, though less substantial than those depicted in Table 5.1. For instance, the redevelopment of Firefox for performance enhancement was estimated to raise development expenses by around 6% that year [Kei17].

5.3.2 Bug prevention

In both Chapters 2 and 3, we encountered numerous instances of straightforward bypasses where a single line of code, often involving the use of a specific HTML element, was sufficient to exploit a bug. In the latter chapter, it became apparent that one of the most common developer intentions when introducing a bug was to incorporate new policy functionality or to introduce a new feature. These oversights reveal that developers did not consistently consider the potential impact on security policies when introducing new features. This observation can be attributed, in part, to the limited resources available to development teams. It is common for security experts to participate primarily in the design phase of a new feature, while their involvement during the implementation phase is less consistent. Furthermore, even when a security expert is assigned to review a new feature, they may not possess comprehensive knowledge of all the intricacies and edge cases related to the relevant policies. For instance, a particular security expert could be highly knowledgeable about SOP, but their familiarity with CSP might be comparatively limited.

With every major browser version, tens to hundreds of new features are introduced [VJ22]. Consequently, it is impractical to subject each feature to an extensive security expert review. As such, we contend that, given the necessary evaluation required to scrutinize all features, investments in automation are key. In connection with this, in Chapter 3, we identified several bugs that could have been detected through a simple automated process. For instance, a straightforward test utilizing the CSP policy `default-src 'none'` could be applied to every newly introduced feature. In this context, neither requests should be sent nor scripts executed on the test page. Such a test would have detected bypasses involving newly introduced features such as `<base>`, `<a>`'s `ping` attribute, favicon retrieval, form submission and Workers.

In a more generalized approach, browser vendors could require security test cases to be developed for each newly introduced feature, that subject the feature to the strictest application of security policies. This process for test generation can be automated to make it accessible to non-security experts. For example, developers could simply specify various HTML pages with different feature configurations, which is likely already a part of their development process. The automated process could then generate test cases by imposing rigorous security policies on these pages, such as CSP, HTTP Strict Transport Security (HSTS), or by indicating that the test must be run in a browser with a strict third-party cookie policy enabled. While these tests may not be exhaustive, they would serve as a valuable indicator of whether the introduced feature enables a bypass and would have detected the previously identified straightforward bypasses. Even though this approach would likely produce false positives, adhering to the principle of "security by default" is a sound development practice.

5.3.3 Bug handling

As illustrated in Chapter 3, the existing methodology for sharing security bugs among browser vendors exhibits limitations that can impede a timely resolution. Despite the push and pull of regression tests to and from the shared WPT repository, oversights have already prevented either action from occurring. Moreover, even in the absence of oversights, this approach can result in a significant delay of several months before the bug is effectively shared.[3] To reduce oversights and expedite the sharing process, an alternative approach could involve sharing bugs immediately after triage. Additionally, it is worth noting that regression tests become public from the moment they are pushed to the WPT repository, which is not ideal for handling security vulnerabilities.

While the adoption of a standardized bug reporting language, as described in Section 5.2.2, could be a proper long-term solution, it is important to ensure that the process of translating a report into this language does not impede the swift sharing of security bugs. Another approach is the direct exchange of confirmed bug reports or PoCs between browser vendors. This method could involve granting limited access within the existing bug tracking platforms to security experts from other vendors. However, this approach relies on a high level of trust between vendors and may face technical challenges related to support for access granularity. A more suitable approach could be for browser vendors to collaborate on a privately shared bug tracking platform, exclusively

[3] This inefficiency primarily stems from the necessary steps preceding the creation of a regression test: First, the bug must undergo triage and be assigned to a developer for resolution. It is only after these steps that a regression test can be created, potentially as part of a fix.

accessible to their security experts. This approach aligns with discussions among security experts from various browsers [Wes23].

A standardized bug reporting language holds potential to enhance the accurate interpretation of bug reports as well, thereby adressing mislabeling to a certain degree. However, it falls short in preventing developers from erroneously classifying bugs as resolved, which can lead to premature public disclosure as evidenced in Chapter 3. To rectify this issue, vendors should intensify the enforcement of regression test creation before a bug is marked as fixed. This is particularly crucial when developers lack information about the revision that resolved the reported bug. Additionally, identifying the bug introducing revision would offer valuable insights for both developers and researchers. Tools like `BugHog` can play a valuable role in this process, aiding in the identification of the revisions of interest. Notably, the Chromium team recognizes the importance of this and offers a *bisect bonus* if a security bug reporter identifies the introducing revision [Gooa].

5.3.4 Deployment of browser engines in native applications

Many issues discussed in Chapter 4 arise from the misconfiguration of embedded browser engines of native applications. One way to mitigate these issues is to establish a default posture in which, in the absence of explicit configuration settings, the most secure and privacy-preserving settings are automatically applied, and sensitive features are disabled. This could involve the disabling of access to the local file system or the prevention of popup window creation. While there has been some progress in this direction, as evidenced by Electron's online documentation that highlights security best practices [Eleb], there are still areas where sensitive settings remain enabled by default. For example, the automatic approval of session permission requests from remotely loaded content can pose a security risk. This configuration allows web pages loaded by the renderer to leverage all the permissions associated with the embedded Chromium engine of Electron [Chre].

However, relying solely on strict default configuration settings may not be sufficient, as developers can still unintentionally misconfigure the browser engine to enable intended use cases. As such, in conjunction with a default security posture, transparency towards developers is essential as well. To achieve this, developers must be well-informed about the potential security and privacy implications associated with specific configurations. Additionally, in certain cases, more fine-grained permissions may be necessary to securely facilitate intended use cases. For instance, an attacker could attempt to exploit file system permissions in applications that genuinely require these for their intended purpose. In such scenarios, a more granular approach might involve requiring users to grant access to specific resources or features each time the application requests them, akin to the practice employed by many mobile OSs. This approach would give users more control over the permissions they grant to applications, reducing the risk of abuse.

Equally important, if not more, is the need for transparency towards users. It is imperative to inform users about the security and privacy implications of applications that embed browser engines, or even software in general. Users could be empowered by providing a list of applications with security and privacy benchmarks. This practice is already in place for privacy measures in web browsers, where users can access privacy test results to evaluate their browser's privacy rating [pri]. Likewise, for EPUB reading systems, tests and their corresponding results that gauge compliance with the W3C's specifications are publicly accessible [HL].

5.4 Concluding remarks

The extensive and growing body of research on the implementation flaws of security and privacy policies serves as compelling evidence that current practices surrounding browser development leave room for improvement. Since more durable solutions, such as changing browser architecture and design, demand substantial time and resources, pragmatic approaches pose more realistic in the short term. These should be employed to prevent oversights that we consider to be low-hanging fruit and for which failure to address them cannot be excused (e.g. one-liner bypasses caused by overlooked HTML elements). Given the limited resources, we argue that the automation of evaluation processes, for instance at feature introduction, is a crucial step in the right direction, where even the most basic measures could prevent numerous identified issues.

Conversely, research on the deployment of browser engines in native applications remains relatively limited and we hope that our work will encourage further exploration of this topic. The inclusion of browser engines within native applications can often be likened to acquiring a Swiss army knife for screwing in a light bulb, as often only a small subset of the engine's functionality is effectively utilized. In light of this, we believe that developing browser engines explicitly designed for non-browser applications (e.g. by featuring a more modular architecture) would result in a reduced attack surface, contributing to a more secure and privacy-preserving use of existing technology.

A

Third-party cookie evaluation

A.1 Test compositions

In this section, we explicate the various test compositions that we have integrated in our framework. These compositions are shown in Table A.1, together with the illustrated domains.

ID	Test composition								
1	🔵	$\xrightarrow[\text{includes}]{}$	🔴						
2	🔵	$\xrightarrow[\text{includes}]{}$	🟢	$\xrightarrow[\text{includes}]{}$	🔴				
3	🔵	$\xrightarrow[\text{includes}]{}$	🟢	$\xrightarrow[\text{includes}]{}$	🔴				
4	🔴	$\xrightarrow[\text{includes}]{}$	🔵	$\xrightarrow[\text{includes}]{}$	🔴				
5	🔵	$\xrightarrow[\text{includes}]{}$	🔴	$\xrightarrow[\text{includes}]{}$	🔵	$\xrightarrow[\text{includes}]{}$	🔴		
6	file://	$\xrightarrow[\text{includes}]{}$	🔴						
7	🔵	$\xrightarrow[\text{includes}]{}$ *	🔴						
8	🔴	$\xrightarrow[\text{includes}]{}$	🔵	$\xrightarrow[\text{includes}]{}$	🔴	$\xrightarrow[\text{includes}]{}$	🔵	$\xrightarrow[\text{includes}]{}$	🔴

First-party

Middle party

Third-party

* Iframe constructed through `data:text/html`.

Table A.1: Test compositions supported by our framework.

A.2 Extension set population

In this section, we present the extension set populations. For the ad tracking protection extensions, these are shown in Table A.2 and for the ad blocking extensions in Table A.3. All extensions for Chrome, Opera and Firefox were selected based on relevant search criteria and a minimum number of users or downloads (whichever was available). Due to the unavailability of both numbers for Edge extensions, we selected Edge extensions based on the popularity of their counterparts for the other browsers. The extension “AdBlocker Lite” takes up two entries in Table 2.2 and A.3 because we tested its two modes.

Set	Extension name	Version	Number of users/downloads
	Chrome Tracking Protection Extensions		
SET B1	Blur	7.7.2390	248,825 users
SET B2	ScriptSafe	1.0.9.1	286,512 users
SET B3	Ghostery	7.4.1.4	2,787,473 users
	Privacy Badger	2017.11.20	711,102 users
	Disconnect	5.18.23	918,877 users
SET B4	uMatrix	1.1.12	121,618 users
	Opera Tracking Protection Extensions		
SET B5	Blur: Protect your passwords, payments & privacy	7.7.2393	154,817 downloads
SET B6	Disconnect	5.17.5	564,628 downloads
	Privacy Badger	2017.11.20	140,381 downloads
SET B7	Ghostery	7.4.3.1	4,865,900 downloads
	Firefox Tracking Protection Extensions		
SET B8	DuckDuckGo Plus*	2017.11.30	419,351 users
SET B9	Privacy Badger	2017.11.20	411,406 users
SET B10	Ghostery – Privacy Ad Blocker	7.4.1.4	1,048,907 users
SET B11	Cliqz - Schnellsuche und Trackingschutz	2.21.3	94,361 users
	Edge Tracking Protection Extensions		
SET B12	Ghostery	7.5.0.0	N/A

* Recently changed its name to "DuckDuckGo Privacy Essentials".

Table A.2: Population of the tracking protection extension sets.

Set	Extension name	Version	Number of users/downloads
	Chrome Ad Blocking Extensions		
SET A1	AdRemover for Google Chrome™	1.1.1.0	9,463,986 users
	Windscribe - Free VPN and Ad Blocker	2.3.4	553,466 users
	uBlock	0.9.5.0	519,056 users
SET A2	AdBlocker Ultimate	2.26	628,321 users
	Ads Killer	0.99.70	2,262,911 users
	Hola ad blocker	1.21.624	143,790 users
SET A3	Fair AdBlocker	1.404	1,808,682 users
SET A4	AdGuard AdBlocker	2.7.2	4,650,713 users
SET A5	AdBlock Pro	4.3	2,134,631 users
SET A6	uBlock Adblocker Plus	2.3	332,645 users
	uBlock Origin	1.14.22	10,000,000+ users
	uBlock Plus Adblocker	1.5.2	521,915 users
SET A7	AdBlock	3.22.1	10,000,000+ users
	Adblock Plus	1.13.4	10,000,000+ users
	Opera Ad Blocking Extensions		
SET A8	AdBlocker Lite (Lite mode)	0.4.0	164,309 downloads
	AdBlock	2.57	11,199,416 downloads
SET A9	AdBlocker Ultimate	2.23	1,209,271 downloads
SET A10	Adblock Fast	1.2.0	465,483 downloads
	AdBlocker Lite (Full mode)	0.4.0	164,309 downloads
SET A11	Adguard	2.7.2	5,649,827 downloads
SET A12	ContentBlockHelper	10.2.0	371,330 downloads
SET A13	uBlock origin	1.14.16	3,738,666 downloads
SET A14	Adblock Plus	1.13.4	33,802,382 downloads
	Firefox Ad Blocking Extensions		
SET A15	AdBlock for Firefox	3.8.0	865,131 users
	AdBlocker Ultimate	2.28	448,458 users
SET A16	Adguard AdBlocker	2.7.3	299,462 users
SET A17	uBlock Origin	1.14.18	5,216,321 users
SET A18	Adblock Plus	3.0.1	13,574,386 users
	Edge Ad Blocking Extensions		
SET A19	AdBlock	2.4.0.0	N/A
SET A20	Adblock Plus	0.9.9.0	N/A
SET A21	Adguard Adblocker	2.8.4	N/A
SET A22	uBlock origin	1.14.24	N/A

Table A.3: Population of the ad blocking extension sets.

A.3 Bug reports and responses

In this section, we address the bug reports that we filed and their subsequent responses. Bugs were reported to both browsers (Section A.3.1) and extensions (Section A.3.2). In order to not inspire any attackers or trackers, we decided to only file private bug reports. Note that bug threads mights still be private when visiting the associated link.

A.3.1 Built-in browser protection

[bug1] The bug that can be leveraged to bypass Chrome's and Opera's third-party cookie policy has been confirmed and is scheduled to be fixed at the time of writing. [1]

[bug2] We reported that Safari 10 does not block all third-party cookies when this option is enabled. At the time of writing, this bug has not yet been confirmed.[2]

[bug3] The bug that nullifies Edge's option to block third-party cookies has been confirmed.[3]

[bug4] The bypasses for Opera's ad blocker have been reported, however, we were not given access to the bug thread. Instead, we were given an email address through which we can inquire about the process.

[bug5] In the bug thread that we have started for bypasses concerning Firefox' tracking protection, references have been made to previously reported similar bugs that are related to Firefox' Safe Browsing feature [Mozm].[4] For example, the AppCache API had already been reported to bypass the URL classifier used by Safe Browsing to signal websites known for phishing or malware. Although the bug has not yet been officially flagged as confirmed at the time of writing, there was an intention to fix.

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=836746

[2] https://bugs.webkit.org/show_bug.cgi?id=186589

[3] https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/16512847

[4] https://bugzilla.mozilla.org/show_bug.cgi?id=1447935

A.3.2 Extensions

[bug6] This bug permitted cross-site requests, initiated by JavaScript embedded in a PDF, to bypass the WebExtension API in Chromium-based browsers. This made it impossible for extensions (e.g. ad blockers and anti-tracking extensions) to implement a thorough third-party cookie and request policy. Unfortunately, our bug thread was closed as WontFix,[5] because this functionality was working as intended; requests initiated by an extension (PDFium) should not be interceptable by other extensions. Thread responses showed reluctance to treating PDFium differently because it would be costly and difficult to implement. We mentioned that Opera - a Chromium-based browser - actually managed to mitigate these requests with its built-in ad blocker, but also proposed an alternative solution like providing a setting to block execution of JavaScript embedded in PDFs. Response to our proposition was supportive, however we are not aware of any progress on the matter. In the same bug report, we also explained the difficulties for extensions to distinct between requests initiated through the AppCache or ServiceWorker API, and requests initiated by browser functionality. However, no responses have been made in regard to this.

[bug7] We reported that requests for fetching the favicons are not interceptable through Firefox' WebExtension API and that requests initiated through the AppCache API are not easily distinguishable in Firefox. The bug thread was closed as WontFix,[6] because the first issue had already been reported and no additional effort will be made to fix the deprecated AppCache API.

[bug8] In addition to the aforementioned bugs caused through the AppCache and WebSocket API, we identified a wide variety of bugs inherent to the implementation of ad blocking and privacy protection extensions. Because of the large number of affected extensions, many without a dedicated bug tracker, we only contacted a selection of them. This selection involved the 11 most popular and recently updated extensions, most of them supported by multiple browsers, to which we reached out through a private channel. Unfortunately, only 5 extension developers responded, of which only 2 pro-actively tried and succeeded to fix the issue.

[5] https://bugs.chromium.org/p/chromium/issues/detail?id=824705

[6] https://bugzilla.mozilla.org/show_bug.cgi?id=1447933

A.3.3 Same-site cookie

[bug9] The prerender bug that we found in Chrome and Opera has been filed through the Chromium project, where it was confirmed and scheduled to be fixed.[7]

[bug10] We have reported the several bypasses that we found for Edge's implementation of the same-site cookie policy. This bug report has been confirmed.[8]

[7] https://bugs.chromium.org/p/chromium/issues/detail?id=709946

[8] https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/18054323/

B

Bug report search criteria and intention labeling

B.1 Bug report search criteria

The search criteria described in the following sections were utilized to collect reports of bugs related to CSP or caused by CSP. These search criteria are intentionally overly broad as not to miss potentially relevant bug reports. False positives were removed manually. Additionally, whenever a discovered report was linked to another relevant report that was not originally part of our search results, it was included in our dataset as well. Note that all keywords used in the search criteria will be checked against the whole bug report, including the title, description, and comments.

B.1.1 Chromium

- label:
  `Security_Severity-Low` OR
  `Security_Severity-Medium` OR
  `Security_Severity-High` OR
  `Security_Severity-Critical`
- status: NOT `WontFix`
- (`CSP` OR `Content-Security-Policy`)

URL: `https://bugs.chromium.org/p/chromium/issues/list?q=%28label%3ASecurity_Severity-Low%20OR%20label%3ASecurity_Severity-Medium%20OR%20label%3ASecurity_Severity-High%20OR%20label%3ASecurity_Severity-Critical%29%20-status%3AWontFix%20%28CSP%20OR%20Content-Security-Policy%29&can=1`

B.1.2 Firefox

- Component: `DOM: Security`
- Resolution: `FIXED`
- Classification: `Client Software, Developer Infrastructure, Components, Server Software, Other`
- Type: `defect`
- Summary: `CSP`

URL: `https://bugzilla.mozilla.org/buglist.cgi?bug_type=defect&short_desc=CSP&classification=Client%20Software&classification=Developer%20Infrastructure&classification=Components&classification=Server%20Software&classification=Other&short_desc_type=allwordssubstr&query_format=advanced&component=DOM%3A%20Security&resolution=FIXED&list_id=16012294`

B.2 Bug report distribution

Table B.1 shows the exact number of valid or available CSP bug reports, how many of those were reproducible by our framework (with regard to its technical limitations), and finally how many we were able to effectively reproduce. Note that this table only elaborates on the number of bug reports found exclusively through our used search criteria. The total number of reproduced bug reports amounts to 86 when also taking into account reports found through cross-report links.

Valid CSP bugs In addition to false positives (e.g. reports that do not describe a CSP bug, or a bug caused by CSP), several reports missed a PoC due to an expired external link. These occurrences were regarded as unavailable if we could not construct a working PoC based on the available description and comments.

	Chromium		Firefox	
Valid CSP bugs	74	↓82%	29	↓86%
In-scope reports	61	↓95%	25	↓92%
Reproduced bugs	58		23	

Table B.1: Number of bug reports.

In-scope reports Not all relevant bugs were reproducible by our framework, due to technical limitations:

- The bug is reported for another OS-specific.[1]
- User interaction is essential for reproducing the bug.
- Console access is required to check exploit success.
- The bug is facilitated by an installed extension.

Reproduced bugs Finally, while all technical requirements were fulfilled, we were not able to effectively reproduce five in-scope bugs. We believe that this may be due to an inadequate PoC, an unclear bug description or a limited understanding of the bug.

B.3 Revision intention labels

In this section, we describe the labeling process and the interpretation of each revision intention label.

B.3.1 Labeling process

The labeling process followed an iterative approach where a researcher built the label list by reviewing all revision metadata and assigning the appropriate label to each revision. A second researcher then independently annotated the same revisions using the pre-constructed label list. The agreement between the two researchers was measured using the Cohen's Kappa coefficient and was found to be 0.81, indicating a good level of agreement. Any disagreements were resolved through discussion until all were resolved.

[1] We found several Chromium bugs reported for iOS. However, since Chromium's iOS version is based on the WebKit engine, we did not consider those as valid.

B.3.2 Label interpretation

Introduce CSP CSP is supported starting from this revision.

Fix affected CSP bug Intentionally fixes the reported bug.

Fix other CSP bug Fixes the reported CSP bug as an unintentional side effect of an intentional fix for another CSP bug.

Fix unrelated security bug Fixes the reported CSP bug as an unintentional side effect of an intentional fix for another non-CSP security bug.

Fix non-security bug Fixes the reported CSP bug as an unintentional side effect of an intentional fix for a non-security bug.

Enable affected CSP feature Enables the CSP feature that is affected by the reported CSP bug.

Enable CSP feature Enables a CSP feature that is not affected by the reported CSP bug.

Enable security feature Enables a non-CSP security feature.

Enable non-security feature Enables a non-security feature.

Update CSP feature Updates a CSP feature that is not affected by the reported CSP bug.

Update security feature Updates a non-CSP security feature.

Update non-security feature Updates a non-security feature.

Design revision of CSP Modifies the high-level design of the CSP implementation.

Design revision of other security policy Modifies the high-level design of another security policy implementation.

Non-security design revision Modifies the high-level design of any other feature implementation.

C

Additional reading system information

For measures of completeness and transparency, we provide an overview of all EPUB reading systems that were considered during our evaluation. We also included the number of reported users. Unfortunately, this metric is not available for iOS, so instead the number of ratings can be used as an estimator of the relative reach of an application. Additionally, we reported on the deduced embedded browser engine for all reading systems supporting JavaScript. Here, “OS” indicates that the reading system relies on the engine framework provided by the operating system, and thus is considered up-to-date. Finally, we also include the readers that were excluded from our analysis along with the reason. In total, we considered 92 reading applications on seven platforms (Windows, Ubuntu, macOS, iOS, Android, Firefox & Chrome extensions) and five standalone physical e-readers.

Reading system	Version	Rendering engine	Release date
Adobe Digital Editions	4.5.10	OS Trident	N/A
Bibliovore	2.0.2.0	-	-
BookReader	1.6.0.0	-	-
Bookviser Reader	6.8.1.0	-	-
Calibre	3.40.1	WebKit 538.1	Oct 2014
Calibre	4.3.0	Blink 77	Sep 2019
CoolReader	N/A	-	-
EPUB File Reader	1.5	OS Trident	N/A
FBReader	0.12.10	-	-
Freda	4.21	-	-
Icecream Ebook Reader	5.19	WebKit 538.1	Oct 2014
Liberty	1.0.0.13	-	-
MS Edge	44.17763.1.0	EdgeHTML 18.17763	Oct 2018
Nook	1.10.1.15	-	-
Overdrive	3.8.0	-	-
SumatraPDF	3.1.2	-	-

Table C.1: Evaluated EPUB reading systems for Windows

Reading system	Reason
Cover	Unable to open fully compliant EPUB file.
Epub3 Reader	Unable to correctly render fully compliant EPUB file.
FlyReader	Unable to open fully compliant EPUB file.
Perfect PDF Reader	Unable to correctly render fully compliant EPUB file.

Table C.2: Omitted EPUB reading systems for Windows

Reading system	Version	Rendering engine	Release date
Adobe Digital Editions	4.5.10	OS WebKit	N/A
Apple Books	1.17	OS WebKit	N/A
Azardi	43.1	Gecko 38	May 2015
BookReader	5.14	OS WebKit	N/A
Calibre	3.40.1	WebKit 538.1	Oct 2014
Calibre	4.3.0	Blink 77	Sep 2019
FBReader	0.9.0	-	-
Kindle	1.25.2	-	-
Kitabu	1.2	OS WebKit	N/A
Murasaki	1.0.2	OS WebKit	N/A

Table C.3: Evaluated EPUB reading systems for macOS

Reading system	Reason
Kobo for Desktop	Unable to side-load EPUBs.

Table C.4: Omitted EPUB reading systems for macOS

Reading system	Version	Rendering engine	Release date
Calibre	3.46	WebKit 538.1	Oct 2014
Calibre	4.3.0	Blink 77	Sep 2019
FBReader	0.12.10	-	-
Okular	1.7.2	-	-

Table C.5: Evaluated EPUB reading systems for Linux Ubuntu

Reading system	Reason
Bookworm	Unable to install.
Buka	Unable to start.
Cool Reader	Unable to start.
Easy eBook Viewer	Unable to open fully compliant EPUBs.
GNOME Books	Unable to open fully compliant EPUBs.
Lucidor	Unable to start.

Table C.6: Omitted EPUB reading systems for Linux Ubuntu

Reading system	Version	Last updated	Number of ratings	Rendering engine	Release date
Aldiko Book Reader	1.1.6	Aug 10, 2017	51	-	-
Apple Books	4.2.3	Jun 3, 2019	N/A	OS WebKit	N/A
Bluefire Reader	2.9	Jan 6, 2018	2.5K	-	-
CHMate	6.9.1	May 22, 2018	13	OS Webkit	N/A
Ebook Reader	4.0.9	Dec 22, 2017	345	OS Webkit	N/A
eBoox	1.60.1	Jul 18, 2019	114	-	-
EPUB Reader	5.1.55	Mar 14, 2017	647	-	-
FBReader	1.0.10	Aug 31, 2019	12	-	-
Gerty	1.1.5	Aug 8, 2015	65	OS Webkit	N/A
Kobo Books	9.14	Jun 11, 2019	8.4K	OS Webkit	N/A
Kybook 3	0.7.8	Feb 23, 2019	579	-	-
Marvin	3.1.2	Oct 11, 2017	239	OS Webkit	N/A
Play Books	5.3.0	Aug 26, 2019	8.3K	-	-
PocketBook	3.2	Aug 12, 2019	637	OS Webkit	N/A
Power Reader	6.10	Aug 28, 2017	4	OS Webkit	N/A
R2 Reader	2.0.1	Jun 21, 2019	5	OS Webkit	N/A
TotalReader	5.1.61	Jul 4, 2017	329	OS Webkit	N/A
YiBook	1.8.5	May 13, 2018	4	-	-
Yomu	2.3.0	Jul 17, 2019	59	OS Webkit	N/A

Table C.7: Evaluated EPUB reading systems for iOS

Reading system	Version	Reason
Kindle	6.24	Unable to side-load EPUBs.
PureReader	1.4.2	Unable to open fully compliant EPUBs.

Table C.8: Omitted EPUB reading systems for iOS

Reading system	Version	Number of downloads	Last updated	Rendering engine	Release date
4shared Reader	1.20.0	1M+	May 2019	-	-
Aldiko Book Reader	3.1.3	10M+	Oct 2018	-	-
Aldiko Classic	3.1.3	500K+	Oct 2018	-	-
AlReader	1.911805270	5M+	May 2018	-	-
Bookari Free	4.2.5	1M+	Feb 2018	-	-
Book Reader	1.12.12	1M+	Jun 2019	-	-
Cool Reader	3.2.32	10M+	Aug 2019	-	-
Ebook Reader	1.0	10K+	Aug 2019	-	-
Ebook Reader	5.0.8.2	5M+	Jun 2019	OS Blink	N/A
EBook Reader	3.5.0	5M+	Jul 2019	-	-
eBoox	2.22	1M+	Aug 2019	-	-
ePub Reader	2.1.2	1M+	May 2015	OS Blink	N/A
Epub reader	4.0	10K+	Apr 2019	-	-
Epub Reader	8.0.39	100K+	Feb 2019	-	-
EPUBReader	1.0.32	100K+	Nov 2014	OS Blink	N/A
eReader Prestigio	6.0.0.9	10M+	May 2019	-	-
FBReader	3.0.15	10M+	Jul 2019	-	-
Freda	4.31	10K+	Mar 2019	-	-
FullReader	4.1.4	1M+	Aug 2019	-	-
Gitden Reader	4.5.3	100K+	Jan 2018	OS Blink	N/A
Google Play Books	5.2.7	1B+	Aug 2019	-	-
Infinity Reader	1.7.57	5K+	May 2017	OS Blink	N/A
iReader	1.1.4	5K+	Aug 2019	OS Blink	N/A
Kindle	3.2.0.35	100M+	Jul 2019	-	-
Librera	8.1.242	10M+	Aug 2019	-	-
Lit Pub	3.5.3	100K+	Jun 2017	OS Blink	N/A
Lithium	0.21.1	1M+	Jan 2019	OS Blink	N/A
Moon+ Reader	5.1	10M+	Aug 2019	-	-
PocketBook	3.21	1M+	Aug 2019	OS Blink	N/A
Reader FB2	1.20	50K+	Jun 2019	-	-
ReadEra	19.07.28	5M+	Jun 2019	-	-
Reasily	1907d	100K+	Jun 2019	OS Blink	N/A
Solati Reader	2.5.1	10K+	Jun 2015	-	-
Supreader	3.2.30	1M+	Dec 2018	OS Blink	N/A
Tolino	4.10.2	100K+	Feb 2019	-	-

Table C.9: Evaluated EPUB reading systems for Android

Reading system	Reason
Adobe Digital Editions	Unable to open local EPUBs.
Cloudshelf	Unable to open local EPUBs.
Kobo	Unable to side-load EPUBs.
SKY Reader	Unable to open local EPUBs.

Table C.10: Omitted EPUB reading systems for Android

Reading system name	Version	Store ID	Number of users
Chrome			
Ebook Reader for Google Drive	1.0.7	mfpbhmcmakfaeajfpehaoijecamlehpl	1,513
EPUBReader	2.0.8	jhhclmfgfllimlhabjkgkeebkbiadflb	137,917
EPUB READER	1.0.1	mbcgbbpomkkndfbpiepjimakkbocjgkh	816
ePUB Reader	0.1.1	dgkibcakfnhcnijakeobjifghganmojn	10,574
ePUB Reader	1.1.0	fnplkbhndemgbopkkpmpnfklkhphpneg	1,893
Firefox			
EPUBReader	2.0.9	-	158,480
ePUB Reader	0.1.1	-	5,097
ePUB Reader	0.0.1	-	36
myBook	0.4.0	-	434
QiuReader	0.1.5	-	1,512

Table C.11: Evaluated EPUB reading systems for Chrome and Firefox

Bibliography

[Aca+14] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. “The Web Never Forgets: Persistent Tracking Mechanisms in the Wild”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 674–689. ISBN: 9781450329576. DOI: `10.1145/2660267.2660347`. URL: `https://doi.org/10.1145/2660267.2660347`.

[Ado20] Adobe. *Security Updates Available for Adobe Digital Editions | APSB20-23*. Apr. 2020. URL: `https://helpx.adobe.com/security/products/Digital-Editions/apsb20-23.html`.

[Agg+10] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. “An Analysis of Private Browsing Modes in Modern Browsers”. In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security’10. Washington, DC: USENIX Association, 2010, pp. 6–6. ISBN: 888-7-6666-5555-4. URL: `http://dl.acm.org/citation.cfm?id=1929820.1929828`.

[AH99] L. Adamic and B. A. Huberman. “The nature of markets in the World Wide Web”. In: (May 1999). URL: `https://dx.doi.org/10.2139/ssrn.166108`.

[Akh+10] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. “Towards a Formal Foundation of Web Security”. In: *2010 23rd IEEE Computer Security Foundations Symposium*. 2010, pp. 290–304. DOI: `10.1109/CSF.2010.27`.

[Ale+20] N. Alexopoulos, S. M. Habib, S. Schulz, and M. Mühlhäuser. “The Tip of the Iceberg: On the Merits of Finding Security Bugs”. In: *ACM Trans. Priv. Secur.* 24.1 (Sept. 2020). ISSN: 2471-2566. DOI: `10.1145/3406112`. URL: `https://doi.org/10.1145/3406112`.

[Ale+22] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser. "How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 359–376. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos.

[Alt12] A. Alter. "Your E-Book Is Reading You". In: *The Wall Street Journal* (July 19, 2012). URL: https://www.wsj.com/articles/SB10001424052702304870304577490950051438304 (visited on 11/06/2019).

[Ama19] *Amazon Kindle Publishing Guidelines*. Amazon, 2019. URL: http://kindlegen.s3.amazonaws.com/AmazonKindlePublishingGuidelines.pdf.

[Anda] Android. *Background Execution Limits*. URL: https://developer.android.com/about/versions/oreo/background.

[Andb] Android. *Open files using storage access framework*. URL: https://developer.android.com/guide/topics/providers/document-provider.

[Andc] Android. *WebSettings*. URL: https://developer.android.com/reference/android/webkit/WebSettings.html.

[App03] Apple. *Apple Unveils Safari*. Jan. 2003. URL: https://www.apple.com/newsroom/2003/01/07Apple-Unveils-Safari/.

[App17] Apple. *UIApplication Background Task Notes*. 2017. URL: https://forums.developer.apple.com/thread/85066.

[App20a] Apple. *About the security content of iOS 13.1 and iPadOS 13.1*. Feb. 2020. URL: https://support.apple.com/en-us/HT210603.

[App20b] Apple. *About the security content of iOS 13.2 and iPadOS 13.2*. Apr. 2020. URL: https://support.apple.com/en-gb/HT210721.

[App20c] Apple. *About the security content of macOS Catalina 10.15*. Feb. 2020. URL: https://support.apple.com/en-us/HT210634.

[App20d] Apple. *About the security content of macOS Catalina 10.15.1, Security Update 2019-001, and Security Update 2019-006*. Apr. 2020. URL: https://support.apple.com/en-us/HT210722.

[Asa+12] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider. "Bug introducing changes: A case study with Android". In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 2012, pp. 116–119. DOI: 10.1109/MSR.2012.6224267.

[Aut17] AuthorEarnings. *February 2017 Big, Bad, Wide & International Report: covering Amazon, Apple, B&N, and Kobo ebook sales in the US, UK, Canada, Australia, and New Zealand.* 2017. URL: https://web.archive.org/web/20190218084936/http:/authorearnings.com/report/february-2017/.

[Aye+11] M. Ayenson, D. Wambach, A. Soltani, N. Good, and C. Hoofnagle. "Flash cookies and privacy II: Now with HTML5 and ETag respawning". In: (2011).

[Bal12a] Baldur Bjarnason. *EPUB javascript security.* July 2012. URL: https://www.baldurbjarnason.com/notes/epub-javascript-security/.

[Bal12b] Baldur Bjarnason. *Javascript in ebooks.* Feb. 2012. URL: https://www.baldurbjarnason.com/notes/javascript-in-ebooks/.

[Ban+14] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. "Discovering concrete attacks on website authorization by formal analysis". In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

[Ban20] D. Bannister. "Facebook pays out $25k bug bounty for chained DOM-based XSS". In: (Nov. 2020). URL: https://portswigger.net/daily-swig/facebook-pays-out-25k-bug-bounty-for-chained-dom-based-xss.

[Bao+22] L. Bao, X. Xia, A. E. Hassan, and X. Yang. "V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities". In: *Proceedings of the 44th International Conference on Software Engineering.* ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2352–2364. ISBN: 9781450392211. DOI: 10.1145/3510003.3510113. URL: https://doi.org/10.1145/3510003.3510113.

[Bar11a] A. Barth. *HTTP State Management Mechanism.* RFC 6265. RFC Editor, Apr. 2011, pp. 1–37. URL: https://tools.ietf.org/html/rfc6265.

[Bar11b] A. Barth. *The Web Origin Concept.* Dec. 2011. DOI: 10.17487/RFC6454. URL: https://www.rfc-editor.org/info/rfc6454.

[Bar13] A. Barth. *Blink: A rendering engine for the Chromium project.* Apr. 2013. URL: https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html.

[Bas+18] M. A. Bashir, S. Arshad, E. Kirda, W. Robertson, and C. Wilson. "How Tracking Companies Circumvented Ad Blockers Using WebSockets". In: *Proceedings of the Internet Measurement Conference 2018*. IMC '18. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 471–477. ISBN: 9781450356190. DOI: 10.1145/3278532.3278573. URL: https://doi.org/10.1145/3278532.3278573.

[BB07] A. Bortz and D. Boneh. "Exposing Private Information by Timing Web Applications". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 621–628. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242656. URL: http://doi.acm.org/10.1145/1242572.1242656.

[BBB16] M. di Biase, M. Bruntink, and A. Bacchelli. "A Security Perspective on Code Review: The Case of Chromium". In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2016, pp. 21–30. DOI: 10.1109/SCAM.2016.30.

[Ber92] T. Berners-Lee. "The world-wide web". In: *Computer Networks and ISDN Systems* 25.4 (1992), pp. 454–459. ISSN: 0169-7552. DOI: https://doi.org/10.1016/0169-7552(92)90039-S. URL: https://www.sciencedirect.com/science/article/pii/016975529290039S.

[Ber96] T. Berners-Lee. "WWW: past, present, and future". In: *Computer* 29.10 (Oct. 1996), pp. 69–77. DOI: 10.1109/2.539724. URL: https://doi.org/10.1109/2.539724.

[BJM08] A. Barth, C. Jackson, and J. C. Mitchell. "Robust Defenses for Cross-site Request Forgery". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: ACM, 2008, pp. 75–88. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455782. URL: http://doi.acm.org/10.1145/1455770.1455782.

[Boe81] B. W. Boehm. *Software engineering economics*. Prentice-Hall, Inc., 1981.

[Boo15] A. Boodman. *How Chromium Works*. Sept. 2015. URL: https://aboodman.medium.com/in-march-2011-i-drafted-an-article-explaining-how-the-team-responsible-for-google-chrome-ships-c479ba623a1b.

[Bos+19] H. Boström, C. Jennings, A. Narayanan, J.-I. Bruaroey, D. Burnett, A. Bergkvist, and B. Aboba. *Media Capture and Streams.* Candidate Recommendation. W3C, July 2019. URL: `https://www.w3.org/TR/2019/CR-mediacapture-streams-20190702/`.

[Bra+22a] L. Braz, C. Aeberhard, G. Calikli, and A. Bacchelli. "Less is More: Supporting Developers in Vulnerability Detection during Code Review". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE).* Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1317–1329. DOI: `10.1145/3510003.3511560`. URL: `https://doi.ieeecomputersociety.org/10.1145/3510003.3511560`.

[Bra+22b] L. Braz, E. Fregnan, V. Arora, and A. Bacchelli. *An Exploratory Study on Regression Vulnerabilities.* 2022. DOI: `10.48550/ARXIV.2207.01942`. URL: `https://arxiv.org/abs/2207.01942`.

[BS12] R. Brandis and L. Steller. *Threat Modelling Adobe PDF.* Tech. rep. Defence Science and Technology Organisation, Aug. 2012. URL: `https://www.dst.defence.gov.au/sites/default/files/publications/documents/DSTO-TR-2730.pdf`.

[BSA95] A. Berg (Writer), J. Schaffer (Writer), and A. Ackerman (Director). *Seinfeld. Season 6, Episode 20: The Doodle.* West-Shapiro Productions and Castle Rock Entertainment, Apr. 1995.

[Bug13] Bugzilla. *945222 - web-platform-tests: Create a test runner for web-platform-tests suite.* Dec. 2013. URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=945222`.

[Bug16] BugReplay. *Pornhub Bypasses Ad Blockers With WebSockets.* 2016. URL: `https://medium.com/thebugreport/pornhub-bypasses-ad-blockers-with-websockets-cedab35a8323`.

[Bug22] Bugzilla. *Put sameSite=lax, sameSite noneRequiresSecure, and sameSite schemeful behind the early beta flag.* Jan. 2022. URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=1751435`.

[BWW23] S. Bingler, M. West, and J. Wilander. *Cookies: HTTP State Management Mechanism.* Internet-Draft draft-ietf-httpbis-rfc6265bis-12. Work in Progress. Internet Engineering Task Force, May 2023. 66 pp. URL: `https://datatracker.ietf.org/doc/draft-ietf-httpbis-rfc6265bis/12/`.

[Cal+20] S. Calzavara, S. Roth, A. Rabitti, M. Backes, and B. Stock. "A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web". In: *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, Aug. 2020, pp. 683–

697. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/calzavara.

[CERa] CERN. *Restoring the first website*. URL: https://first-website.web.cern.ch/first-website/.

[CERb] CERN. *The birth of the Web*. URL: https://home.web.cern.ch/science/computing/birth-web/short-history-web.

[CG19] D. Cramer and M. Garrish. *EPUB Content Documents 3.2*. Standard. W3C, May 2019. URL: https://www.w3.org/publishing/epub3/epub-contentdocs.html.

[Cha22] A. Chavez. *Expanding testing for the Privacy Sandbox for the Web*. July 2022. URL: https://blog.google/products/chrome/update-testing-privacy-sandbox-web/.

[Chra] Chrome. *Declare Permissions and Warn Users*. URL: https://developer.chrome.com/extensions/permission_warnings.

[Chrb] Chromium. *bisect-builds.py*. URL: https://www.chromium.org/developers/bisect-builds-py/.

[Chrc] Chromium. *Reporting Security Bugs*. URL: https://www.chromium.org/Home/chromium-security/reporting-security-bugs/.

[Chrd] Chromium. *Testing and infrastructure*. URL: https://www.chromium.org/developers/testing/.

[Chre] Chromium for Developers. *Permissions list*. URL: https://developer.chrome.com/docs/extensions/mv3/declare_permissions/#permissions.

[Chrf] Chromium Projects. *Developer FAQ - Why Blink?* URL: https://www.chromium.org/blink/developer-faq/.

[Chr11] Chromium. *Chrome Prerendering*. 2011. URL: https://www.chromium.org/developers/design-documents/prerender.

[Chr12a] Chromium. *chrome.webRequest.onBeforeRequest doesn't intercept WebSocket requests*. 2012. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=129353.

[Chr12b] Chromium. *Revision 165317: introduction of CSP 1.0*. Nov. 2012. URL: https://chromium.googlesource.com/chromium/src/+/46dd3610caa75097ba521f7f74e5f5c0d7c23b79.

[Chr14] Chromium. *Issue 413454: We should be able to import the w3c test suites directly into blink (checking them in)*. Sept. 2014. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=413454.

[Chr20a] Chromium. *Issue 1115628: Security: Full CSP bypass through blob: URIs*. Aug. 2020. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1115628.

[Chr20b] Chromium. *Issue 1149272: Add Content Security Policies to the Policy Container*. Nov. 2020. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1149272.

[CM07] S. Christey and R. A. Martin. "Vulnerability Type Distributions in CVE". In: (May 2007). URL: https://cwe.mitre.org/documents/vuln-trends/index.html.

[CMN15] F. Camilo, A. Meneely, and M. Nagappan. "Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project". In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, 2015, pp. 269–279. ISBN: 9780769555942.

[com11] comScore. *The Impact of Cookie Deletion on Site-Server and Ad-Server Metrics in Australia*. Jan. 2011.

[Con20] M. Conca. *Changes to SameSite Cookie Behavior – A Call to Action for Web Developers*. Aug. 2020. URL: https://hacks.mozilla.org/2020/08/changes-to-samesite-cookie-behavior/.

[CRB16] S. Calzavara, A. Rabitti, and M. Bugliesi. "Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1365–1375. ISBN: 9781450341394. DOI: 10.1145/2976749.2978338. URL: https://doi.org/10.1145/2976749.2978338.

[CRB18] S. Calzavara, A. Rabitti, and M. Bugliesi. "Semantics-Based Analysis of Content Security Policy Deployment". In: *ACM Trans. Web* 12.2 (Jan. 2018). ISSN: 1559-1131. DOI: 10.1145/3149408. URL: https://doi.org/10.1145/3149408.

[CSS10] A. Castiglione, A. D. Santis, and C. Soriente. "Security and privacy issues in the Portable Document Format". In: *Journal of Systems and Software* 83.10 (2010), pp. 1813–1822. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2010.04.062. URL: http://www.sciencedirect.com/science/article/pii/S0164121210001287.

[Cui18] A. Cui. *The Overlooked Problem of 'N-Day' Vulnerabilities*. Dark Reading, Mar. 2018. URL: https://www.darkreading.com/vulnerabilities---threats/the-overlooked-problem-of-n-day-vulnerabilities/a/d-id/1331348.

[Cur19] Cure53. *HTTPLeaks*. GitHub, 2019. URL: https://github.com/cure53/HTTPLeaks.

[CVE] CVE Details. *QT 5.2.1 Security Vulnerabilities*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-12593/product_id-24410/version_id-164958/Digia-QT-5.2.1.html.

[Dam$^+$22] S. Dambra, I. Sanchez-Rola, L. Bilge, and D. Balzarotti. "When Sally Met Trackers: Web Tracking From the Users' Perspective". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2189–2206. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/dambra.

[Dem00] L. Dembart. "U.S. Removes an Encryption Barrier". In: (Jan. 2000). URL: https://www.nytimes.com/2000/01/31/business/worldbusiness/IHT-us-removes-an-encryption-barrier.html.

[DG19] M. Day and J. Gu. *The Enormous Numbers Behind Amazon's Market Reach*. Mar. 2019. URL: https://www.bloomberg.com/graphics/2019-amazon-reach-across-markets/.

[Dig17] Digimarc. *Inside the Mind of a Book Pirate*. 2017. URL: https://www.digimarc.com/docs/default-source/default-document-library/inside-the-mind-of-a-book-pirate.

[Dim$^+$21] Y. Dimova, G. Acar, L. Olejnik, W. Joosen, and T. Van Goethem. "The cname of the game: Large-scale analysis of dns-based tracking evasion". In: 2021, pp. 394–412. URL: https://doi.org/10.2478/popets-2021-0053.

[Dim$^+$22] Y. Dimova, G. Franken, V. Le Pochat, W. Joosen, and L. Desmet. "Tracking the Evolution of Cookie-Based Tracking on Facebook". In: WPES'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 181–196. ISBN: 9781450398732. DOI: 10.1145/3559613.3563200. URL: https://doi.org/10.1145/3559613.3563200.

[Din$^+$21] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, et al. "Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases." In: *NDSS*. 2021.

[DL07] W. Diffie and S. Landau. "26 - The export of cryptography in the 20th and the 21st centuries". In: *The History of Information Security*. Ed. by K. D. Leeuw and J. Bergstra. Amsterdam: Elsevier Science B.V., 2007, pp. 725–736. ISBN: 978-0-444-51608-4. DOI: https://doi.org/10.1016/B978-044451608-4/50027-4. URL: https://www.sciencedirect.com/science/article/pii/B9780444516084500274.

[DMZ10] M. Duerst, L. Masinter, and J. Zawinski. *The 'mailto' URI Scheme.* RFC 6068. RFC Editor, Oct. 2010. URL: `https://tools.ietf.org/html/rfc6068`.

[DS92] Dr. Dre and Snoop Dogg. *Nuthin' but a "G" Thang.* Death Row, Interscope, and Priority, Nov. 1992.

[EC12] N. Edwards and L. Chen. "An Historical Examination of Open Source Releases and Their Vulnerabilities". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security.* CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 183–194. ISBN: 9781450316514. DOI: `10.1145/2382196.2382218`. URL: `https://doi.org/10.1145/2382196.2382218`.

[Eck10] P. Eckersley. "How Unique is Your Web Browser?" In: *Proceedings of the 10th International Conference on Privacy Enhancing Technologies.* PETS'10. Berlin, Germany: Springer-Verlag, 2010, pp. 1–18. ISBN: 3-642-14526-4, 978-3-642-14526-1. URL: `http://dl.acm.org/citation.cfm?id=1881151.1881152`.

[Elea] Electron. *Electron Releases.* URL: `https://www.electronjs.org/docs/latest/tutorial/electron-timelines`.

[Eleb] Electron. *Security.* URL: `https://www.electronjs.org/docs/latest/tutorial/security`.

[EN16] S. Englehardt and A. Narayanan. "Online Tracking: A 1-million-site Measurement and Analysis". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Vienna, Austria: ACM, 2016, pp. 1388–1401. ISBN: 978-1-4503-4139-4. DOI: `10.1145/2976749.2978313`. URL: `http://doi.acm.org/10.1145/2976749.2978313`.

[Eri13] Eric Hellman. *Publishing Hackathon Pretty Much Ignores eBooks.* 2013. URL: `https://go-to-hellman.blogspot.com/2013/05/publishing-hackathon-pretty-much.html`.

[FAW13] M. Finifter, D. Akhawe, and D. Wagner. "An Empirical Study of Vulnerability Rewards Programs". In: *22nd USENIX Security Symposium (USENIX Security 13).* Washington, D.C.: USENIX Association, Aug. 2013, pp. 273–288. ISBN: 978-1-931971-03-4. URL: `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/finifter`.

[Fie+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. RFC Editor, June 1999, pp. 1–37. URL: `https://tools.ietf.org/html/rfc2616`.

[Fira] Firefox Source Docs. *How To Contribute Code To Firefox.* URL: https://firefox-source-docs.mozilla.org/setup/contributing_code.html.

[Firb] Firefox Source Docs. *Mochitest.* URL: https://firefox-source-docs.mozilla.org/testing/mochitest-plain/index.html.

[Fir13] Firefox. *Revision 144546: introduction of CSP 1.0.* Sept. 2013. URL: https://hg.mozilla.org/releases/mozilla-release/rev/6b181afc9fadbd4bb9d04648aa24a34bd9731e82.

[FKS16] D. Fett, R. Küsters, and G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1204–1215. ISBN: 9781450341394. DOI: 10.1145/2976749.2978385. URL: https://doi.org/10.1145/2976749.2978385.

[FKS17] D. Fett, R. Küsters, and G. Schmitz. "The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF).* 2017, pp. 189–202. DOI: 10.1109/CSF.2017.20.

[Flo14] A. Flood. "Ebooks can tell which novels you didn't finish". In: *The Guardian* (Dec. 10, 2014). URL: https://www.theguardian.com/books/2014/dec/10/kobo-survey-books-readers-finish-donna-tartt (visited on 11/06/2019).

[Fra+23] G. Franken, T. Van Goethem, L. Desmet, and W. Joosen. "A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs". In: *32nd USENIX Security Symposium (USENIX Security 23).* Anaheim, CA: USENIX Association, Aug. 2023, pp. 3673–3690. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/franken.

[Fre+22] E. Fregnan, L. Braz, M. D'Ambros, G. Çalikli, and A. Bacchelli. "First Come First Served: The Impact of File Position on Code Review". In: *arXiv preprint arXiv:2208.04259* (2022).

[FVJ18] G. Franken, T. Van Goethem, and W. Joosen. "Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies". In: *27th USENIX Security Symposium (USENIX Security 18).* Baltimore, MD: USENIX Association, Aug. 2018, pp. 151–168. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/franken.

[FVJ19] G. Franken, T. Van Goethem, and W. Joosen. "Exposing Cookie Policy Flaws Through an Extensive Evaluation of Browsers and Their Extensions". In: *IEEE Security & Privacy* 17.4 (2019), pp. 25–34. DOI: 10.1109/MSEC.2019.2909710.

[FVJ21] G. Franken, T. Van Goethem, and W. Joosen. "Reading Between the Lines: An Extensive Evaluation of the Security and Privacy Implications of EPUB Reading Systems". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1730–1747. DOI: 10.1109/SP40001.2021.00015.

[GC19] M. Garrish and D. Cramer. *EPUB 3.2*. Standard. W3C, May 2019. URL: https://www.w3.org/publishing/epub32/epub-spec.html.

[GCH23] M. Garrish, D. Cramer, and I. Herman. *EPUB 3.3*. W3C Recommendation. W3C, May 2023. URL: https://www.w3.org/TR/2023/REC-epub-33-20230525/.

[GH15] N. Gelernter and A. Herzberg. "Cross-Site Search Attacks". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 1394–1405. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813688. URL: http://doi.acm.org/10.1145/2810103.2813688.

[Git] GitHub. *PDF.js*. URL: https://mozilla.github.io/pdf.js/.

[Git22] GitHub. *electron 19.1.9*. Nov. 2022. URL: https://github.com/electron/electron/releases/tag/v19.1.9.

[GKK21] D. Geradin, D. Katsifis, and T. Karanikioti. "Google as a de facto privacy regulator: analysing the privacy sandbox from an antitrust perspective". In: *European Competition Journal* 17.3 (2021), pp. 617–681.

[Gooa] Google. *Chrome Vulnerability Reward Program Rules*. URL: https://bughunters.google.com/about/rules/5745167867576320/chrome-vulnerability-reward-program-rules.

[Goob] Google. *Web Tests (formerly known as "Layout Tests" or "LayoutTests")*. URL: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/testing/web_tests.md%5C#bisecting-regressions.

[Gooc] Google Source. *PDFium*. URL: https://pdfium.googlesource.com/pdfium/.

[Goo22] Google Cloud. *DevOps tech: Trunk-based development*. 2022. URL: https://cloud.google.com/architecture/devops/devops-tech-trunk-based-development.

[Gro+07] J. Grossman, R. Hansen, P. D. Petkov, A. Rager, and S. Fogie. *XSS Attacks: Cross Site Scripting Exploits and Defense.* Syngress Publishing, 2007. ISBN: 9780080553405.

[Gro06] J. Grossman. "CSRF, the sleeping giant". In: (Sept. 2006). URL: `https://blog.jeremiahgrossman.com/2006/09/csrf-sleeping-giant.html`.

[GS02] S. Garfinkel and G. Spafford. *Web security, privacy & commerce.* O'Reilly Media, Inc., 2002.

[GW17] I. Grigorik and M. West. *Reporting API.* Tech. rep. Nov. 2017. URL: `https://wicg.github.io/reporting/`.

[GW96] I. Goldberg and D. Wagner. "Randomness and the Netscape Browser". In: *Dr. Dobb's Journal.* Jan. 1996. URL: `https://people.eecs.berkeley.edu/~daw/papers/ddj-netscape.html`.

[Hei+12] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. "Scriptless Attacks: Stealing the Pie without Touching the Sill". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security.* CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 760–771. ISBN: 9781450316514. DOI: `10.1145/2382196.2382276`. URL: `https://doi.org/10.1145/2382196.2382276`.

[Hic16] I. Hickson. *Web Storage (Second Edition).* W3C Recommendation. http://www.w3.org/TR/2016/REC-webstorage-20160419/. W3C, Apr. 2016.

[HL] I. Herman and D. Lazin. *EPUB 3.3 Test Results.* URL: `https://w3c.github.io/epub-tests/results`.

[HMN15] C. Hothersall-Thomas, S. Maffeis, and C. Novakovic. "BrowserAudit: Automated Testing of Browser Security Features". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 37–47. ISBN: 9781450336208. DOI: `10.1145/2771783.2771789`. URL: `https://doi.org/10.1145/2771783.2771789`.

[How21] E. Howcroft. "World Wide Web source code NFT sells for 5.4 million at Sotheby's". In: (July 2021). URL: `https://www.reuters.com/technology/world-wide-web-source-code-nft-sells-54-million-sothebys-2021-06-30/`.

[IAN] Internet Assigned Numbers Authority. *Uniform Resource Identifier (URI) Schemes.* URL: `https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml`.

[Ian+23] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia, and F. Palomba. "The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study". In: *IEEE Transactions on Software Engineering* 49.1 (2023), pp. 44–63. DOI: 10.1109/TSE.2022.3140868.

[IAN14] Internet Assigned Numbers Authority. *Media type assignment: epub+zip.* Nov. 2014. URL: https://www.iana.org/assignments/media-types/application/epub+zip.

[Int17] Intellectual Property Office. *Online Copyright Infringement Tracker: Latest wave of research.* 2017. URL: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/628704/OCI_-tracker-7th-wave.pdf.

[ISQ17] U. Iqbal, Z. Shafiq, and Z. Qian. "The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists". In: Nov. 2017, pp. 171–183.

[IV10] H. J. d. V. Ilan Oshri and H. de Vries. "The rise of Firefox in the web browser industry: The role of open source in setting standards". In: *Business History* 52.5 (2010), pp. 834–856. DOI: 10.1080/00076791.2010.499431. eprint: https://doi.org/10.1080/00076791.2010.499431. URL: https://doi.org/10.1080/00076791.2010.499431.

[Jac96] T. Jackson. "This bug in your PC is a smart cookie". In: *Financial Times* (Feb. 1996).

[JB08] C. Jackson and A. Barth. "Beware of finer-grained origins". In: Web. 2008. URL: https://seclab.stanford.edu/websec/origins/fgo.pdf.

[JLS13] M. Johns, S. Lekies, and B. Stock. "Eradicating DNS Rebinding with the Extended Same-origin Policy". In: *22nd USENIX Security Symposium (USENIX Security 13).* Washington, D.C.: USENIX Association, Aug. 2013, pp. 621–636. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/johns.

[JTL12] D. Jang, Z. Tatlock, and S. Lerner. "Establishing Browser Security Guarantees through Formal Shim Verification". In: *21st USENIX Security Symposium (USENIX Security 12).* Bellevue, WA: USENIX Association, Aug. 2012, pp. 113–128. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang.

[Jun17] Jun Kokatsu. *Is your ePub reader secure enough?* May 2017. URL: https://shhnjk.blogspot.com/2017/05/is-your-epub-reader-secure-enough.html.

[Kaf25] F. Kafka. *The Trial.* Verlag Die Schmiede, Apr. 1925.

[Kas10] M. Kaste. *Is Your E-Book Reading Up On You?* Dec. 14, 2010. URL: https://www.npr.org/2010/12/15/132058735/is-your-e-book-reading-up-on-you (visited on 11/06/2019).

[KC15] G. Kontaxis and M. Chew. “Tracking Protection in Firefox For Privacy and Performance”. In: *In IEEE Web 2.0 Security & Privacy* (2015). URL: https://arxiv.org/abs/1506.04104.

[Kei17] G. Keizer. *Mozilla’s record 2016 revenue funded its Firefox Quantum browser.* Dec. 2017. URL: https://www.computerworld.com/article/3240008/mozillas-record-2016-revenue-funded-its-firefox-quantum-browser.html.

[Kes22] A. van Kesteren. *The Topics API.* Dec. 2022. URL: https://github.com/WebKit/standards-positions/issues/111.

[Kle10] L. Kleinrock. “An early history of the internet”. In: *IEEE Communications Magazine* 48.8 (Aug. 2010), pp. 26–36.

[KM97] D. Kristol and L. Montulli. *HTTP State Management Mechanism.* RFC 2109. RFC Editor, Feb. 1997, pp. 1–21. URL: https://tools.ietf.org/html/rfc2109.

[KMG18] C. Kerschbaumer, F. Marier, and M. Goodwin. *Supporting Same-Site Cookies in Firefox 60.* Apr. 2018. URL: https://blog.mozilla.org/security/2018/04/24/same-site-cookies-in-firefox-60/.

[Kni+21] L. Knittel, C. Mainka, M. Niemietz, D. T. Noß, and J. Schwenk. “XSinator.Com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1771–1788. ISBN: 9781450384544. DOI: 10.1145/3460120.3484739. URL: https://doi.org/10.1145/3460120.3484739.

[Kob] Kobo Labs. *Kobo EPUB Guidelines.* URL: https://github.com/kobolabs/epub-spec/blob/master/README.md.

[KPW06] S. Kim, K. Pan, and E. E. J. Whitehead. “Memories of Bug Fixes”. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* SIGSOFT ’06/FSE-14. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 35–45. ISBN: 1595934685. DOI: 10.1145/1181775.1181781. URL: https://doi.org/10.1145/1181775.1181781.

[Kri01] D. M. Kristol. “HTTP Cookies: Standards, Privacy, and Politics”. In: *ACM Trans. Internet Technol.* 1.2 (Nov. 2001), pp. 151–198. ISSN: 1533-5399. DOI: 10.1145/502152.502153. URL: https://doi.org/10.1145/502152.502153.

[Lap+20] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine. “Browser Fingerprinting: A Survey”. In: *ACM Trans. Web* 14.2 (Apr. 2020). ISSN: 1559-1131. DOI: 10.1145/3386040. URL: https://doi.org/10.1145/3386040.

[Lee17] J. H. Lee. *Issue 1134: WebKit: UXSS via ContainerNode :: parserRemoveChild (2)*. 2017. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1134.

[Lei+97] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. S. Wolff. “The Past and Future History of the Internet”. In: *Commun. ACM* 40.2 (Feb. 1997), pp. 102–108. ISSN: 0001-0782. DOI: 10.1145/253671.253741. URL: https://doi.org/10.1145/253671.253741.

[Lek+15] S. Lekies, B. Stock, M. Wentzel, and M. Johns. “The Unexpected Dangers of Dynamic JavaScript”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 723–735. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies.

[Ler+13] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. “Verifying web browser extensions’ compliance with private-browsing mode”. In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 57–74.

[leva] levels.fyi. *Google L5 Software Engineer Salary*. URL: https://www.levels.fyi/companies/google/salaries/software-engineer/levels/l5.

[levb] levels.fyi. *Mozilla P5 - Senior Staff Sofware Engineer Salary*. URL: https://www.levels.fyi/companies/mozilla/salaries/software-engineer/levels/p5.

[Lin+09] X. Lin, P. Zavarsky, R. Ruhl, and D. Lindskog. “Threat Modeling for CSRF Attacks”. In: *2009 International Conference on Computational Science and Engineering*. Vol. 3. 2009, pp. 486–491. DOI: 10.1109/CSE.2009.372.

[Liu+19] M. Liu, B. Zhang, W. Chen, and X. Zhang. “A Survey of Exploitation and Detection Methods of XSS Vulnerabilities”. In: *IEEE Access* 7 (2019), pp. 182004–182016. DOI: 10.1109/ACCESS.2019.2960449.

[LRB16] P. Laperdrix, W. Rudametkin, and B. Baudry. "Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 878–894. DOI: 10.1109/SP.2016.57.

[Luo$^+$17] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis. "Hindsight: Understanding the Evolution of UI Vulnerabilities in Mobile Browsers". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 149–162. ISBN: 9781450349468. DOI: 10.1145/3133956.3133987. URL: https://doi.org/10.1145/3133956.3133987.

[Luo$^+$19] M. Luo, P. Laperdrix, N. Honarmand, and N. Nikiforakis. "Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers". In: *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. 2019. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_01A-4_Luo_paper.pdf.

[MCG13] D. Maiorca, I. Corona, and G. Giacinto. "Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection". In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. Hangzhou, China: ACM, 2013, pp. 119–130. ISBN: 978-1-4503-1767-2. DOI: 10.1145/2484313.2484327. URL: http://doi.acm.org/10.1145/2484313.2484327.

[Mer23] R. Merewood. *Preparing for the end of third-party cookies*. Oct. 2023. URL: https://developer.chrome.com/blog/cookie-countdown-2023oct/.

[Mic] Microsoft. *Version update history for the new and classic Microsoft Teams app*. URL: https://learn.microsoft.com/en-us/officeupdates/teams-app-versioning.

[Mic09] Microsoft. *Happy 10th birthday Cross-Site Scripting!* Dec. 2009. URL: https://learn.microsoft.com/en-ca/archive/blogs/dross/happy-10th-birthday-cross-site-scripting.

[Mic18] Microsoft. *Platform status*. 2018. URL: https://developer.microsoft.com/en-us/microsoft-edge/platform/status/samesitecookies/.

[Miy01] H. Miyazaki (Writer and Director). *Spirited Away*. Studio Ghibli, July 2001.

[Mla+18] V. Mladenov, C. Mainka, K. M. zu Selhausen, M. Grothe, and J. Schwenk. "Vulnerability Report: Attacks bypassing the signature validation in PDF". In: (Nov. 2018). https://www.nds.ruhr-uni-bochum.de/media/ei/veroeffentlichungen/2019/02/12/report.pdf.

[MM12] J. R. Mayer and J. C. Mitchell. "Third-Party Web Tracking: Policy and Technology". In: *2012 IEEE Symposium on Security and Privacy*. May 2012, pp. 413–427. DOI: 10.1109/SP.2012.47.

[Moza] Mozilla. *Handling Mozilla Security Bugs*. URL: https://www.mozilla.org/en-US/about/governance/policies/security-group/bugs/.

[Mozb] Mozilla. *Patch uplifting rules*. URL: https://wiki.mozilla.org/Release_Management/Uplift_rules.

[Mozc] Mozilla. *Release Management/Feature Uplift*. URL: https://wiki.mozilla.org/Release_Management/Feature_Uplift.

[Mozd] Mozilla. *Understanding Web Security Checks in Firefox (Part 1)*. URL: https://blog.mozilla.org/attack-and-defense/2020/06/10/understanding-web-security-checks-in-firefox-part-1/.

[Moze] Mozilla Developer Network. *Content Security Policy*. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_Security_Policy.

[Mozf] Mozilla Developer Network. *Content Security Policy (CSP)*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[Mozg] Mozilla Developer Network. *IndexedDB API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[Mozh] Mozilla Developer Network. *LocalStorage*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage.

[Mozi] Mozilla Developer Network. *mdn-browser-compat-data*. URL: https://github.com/mdn/browser-compat-data.

[Mozj] Mozilla Developer Network. *MediaDevices*. URL: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices.

[Mozk] Mozilla Developer Network. *Same-origin policy*. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[Mozl] Mozilla Support. *Firefox Focus*. URL: https://support.mozilla.org/en-US/products/focus-firefox.

[Mozm] Mozilla Wiki. URL: https://wiki.mozilla.org/Security/Safe_Browsing.

[Mozn] MozillaSecurity. *autobisect*. URL: https://github.com/MozillaSecurity/autobisect.

[Moz15] Mozilla Blog. *Firefox Now Offers a More Private Browsing Experience*. 2015. URL: https://blog.mozilla.org/blog/2015/11/03/firefox-now-offers-a-more-private-browsing-experience/.

[Moz17a] Mozilla Developer Network. *Beacon API*. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/Beacon_API.

[Moz17b] Mozilla Developer Network. *Fetch API*. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[Moz17c] Mozilla Developer Network. *Service Worker API*. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[Moz17d] Mozilla Developer Network. *webRequest*. 2017. URL: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest.

[Moz17e] Mozilla Developer Network. *WebSocket*. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket.

[Moz17f] Mozilla Developer Network. *XMLHttpRequest*. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[Moz18a] Mozilla Developer Network. *EventSource*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/EventSource.

[Moz18b] Mozilla Developer Network. *Using the application cache*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache.

[Moz23] Mozilla. *Firefox rolls out Total Cookie Protection by default to more users worldwide*. Apr. 2023. URL: https://blog.mozilla.org/en/mozilla/firefox-rolls-out-total-cookie-protection-by-default-to-all-users-worldwide/.

[Mun+17] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan. “Do Bugs Foreshadow Vulnerabilities? An in-Depth Study of the Chromium Project”. In: *Empirical Softw. Engg.* 22.3 (June 2017), pp. 1305–1347. ISSN: 1382-3256. DOI: 10.1007/s10664-016-9447-3. URL: https://doi.org/10.1007/s10664-016-9447-3.

[Nat13] Nate Hoffelder. "An Epub3 eBook Could be Used to Hack Your Tablet, Steal Your Identity, and Cause the Downfall of Western Civilization". In: (2013). `https://the-digital-reader.com/2013/06/09/eric-hellmans-publishing-hackathon-entry-could-be-used-to-hack-your-tablet-steal-your-identity-and-cause-the-downfall-of-western-civilization/`.

[Nis+15] N. Nissim, A. Cohen, C. Glezer, and Y. Elovici. "Detection of malicious PDF files and directions for enhancements: A state-of-the art survey". In: *Computers & Security* 48 (2015), pp. 246–266. ISSN: 0167-4048. DOI: `https://doi.org/10.1016/j.cose.2014.10.014`. URL: `http://www.sciencedirect.com/science/article/pii/S0167404814001606`.

[Not10] M. Nottingham. *Web Linking*. RFC 5988. RFC Editor, Oct. 2010, pp. 1–23. URL: `https://tools.ietf.org/html/rfc5988`.

[nrc19] nrclark. *Pyfuse: A tool for simple FUSE Filesystems*. GitHub, 2019. URL: `https://github.com/nrclark/pyfuse`.

[Och11] J. G. Ochin. "Cross Browser Incompatibility: Reasons and Solutions". In: *International Journal of Software Engineering & Applications (IJSEA)* 2.3 (2011), pp. 66–77.

[OS06] A. Ozment and S. E. Schechter. "Milk or Wine: Does Software Security Improve with Age?" In: *15th USENIX Security Symposium (USENIX Security 06)*. Vancouver, B.C. Canada: USENIX Association, July 2006. URL: `https://www.usenix.org/conference/15th-usenix-security-symposium/milk-or-wine-does-software-security-improve-age`.

[OWA] OWASP. *Top Ten Project*. URL: `https://owasp.org/www-project-top-ten/`.

[Par+20] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. "Fuzzing JavaScript Engines with Aspect-preserving Mutation". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1629–1642. DOI: `10.1109/SP40000.2020.00067`.

[Phi98] B. Phillips. "Designers: the browser war casualties". In: *Computer* 31.10 (1998), pp. 14–16. DOI: `10.1109/2.722269`.

[Pie17] M. Pietraszak. *Browser Extensions*. Draft Community Group Report. https://browserext.github.io/browserext/. W3C, July 2017. URL: `Draft%20Community%20Group%20Report`.

[Pop16] A. Popescu. *Geolocation API Specification 2nd Edition*. W3C Recommendation. https://www.w3.org/TR/2016/REC-geolocation-API-20161108/. W3C, Nov. 2016.

[pri] privacytests.org. *PrivacyTests.org: Open-source tests of web browser privacy.* URL: `https://privacytests.org/`.

[Pro] Project Gutenberg. *Project Gutenberg Submission Guidelines.* URL: `https://web.archive.org/web/20181108181052/https://upload.pglaf.org/`.

[PwC10] PricewaterhouseCoopers. "Turning the Page: The Future of eBooks". In: (2010). https://www.pwc.co.uk/assets/pdf/ebooks-trends-and-developments.pdf. (Visited on 11/10/2019).

[PXH22] V. Prakash, S. Xie, and D. Y. Huang. "Inferring Software Update Practices on Smart Home IoT Devices Through User Agent Analysis". In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses.* SCORED'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 93–103. ISBN: 9781450398855. DOI: `10.1145/3560835.3564551`. URL: `https://doi.org/10.1145/3560835.3564551`.

[Res05] E. Rescorla. "Is Finding Security Holes a Good Idea?" In: *IEEE Security and Privacy* 3.1 (Jan. 2005), pp. 14–19. ISSN: 1540-7993. DOI: `10.1109/MSP.2005.17`. URL: `https://doi.org/10.1109/MSP.2005.17`.

[Res21] E. Rescorla. *Privacy analysis of FLoC.* June 2021. URL: `Privacy%20analysis%20of%20FLoC`.

[RKW12] F. Roesner, T. Kohno, and D. Wetherall. "Detecting and Defending Against Third-party Tracking on the Web". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation.* NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 12–12. URL: `http://dl.acm.org/citation.cfm?id=2228298.2228315`.

[RMJ15] K. Russell, Z. Mo, and B. Jones. "Continuous Testing of Chrome's WebGL Implementation". In: *WebGL Insights.* Ed. by P. Cozzi. http://www.webglinsights.com/. CRC Press, July 2015, pp. 31–46. ISBN: 978-1498716079.

[Rot$^+$] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock. "Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies". In: *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)* (). DOI: `10.14722/ndss.2020.23046`. URL: `https://par.nsf.gov/biblio/10173479`.

[RPS23] J. Rautenstrauch, G. Pellegrino, and B. Stock. "The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web". In: *44th IEEE Symposium on Security and Privacy.* May 2023. URL: `https://publications.cispa.saarland/3892/`.

[RSP17] S. Rathore, P. K. Sharma, and J. H. Park. "XSSClassifier: An Efficient XSS Attack Detection Approach Based on Machine Learning Classifier on SNSs." In: *Journal of Information Processing Systems* 13.4 (2017).

[Rya10] J. Ryan. *A History of the Internet and the Digital Future.* Reaktion Books, 2010.

[SB11] H. Saiedian and D. Broyle. "Security Vulnerabilities in the Same-Origin Policy: Implications and Alternatives". In: *Computer* 44.9 (2011), pp. 29–36. DOI: `10.1109/MC.2011.226`.

[SB12] B. Sterne and A. Barth. *Content Security Policy 1.0.* W3C Candidate Recommendation. https://www.w3.org/TR/2012/CR-CSP-20121115/. W3C, Nov. 2012.

[SBM95] D. Simon (Writer), E. Burns (Writer), and P. Medak (Director). *The Wire. Season 1, Episode 3: The Buys.* Blown Deadline Productions and HBO Entertainment, Apr. 1995.

[SBR17] D. F. Somè, N. Bielova, and T. Rezk. "On the Content Security Policy Violations Due to the Same-Origin Policy". In: *Proceedings of the 26th International Conference on World Wide Web.* WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 877–886. ISBN: 9781450349130. DOI: `10.1145/3038912.3052634`. URL: `https://doi.org/10.1145/3038912.3052634`.

[Sha17] R. Sharma. *Preventing cross-site attacks using same-site cookies.* 2017. URL: `https://blogs.dropbox.com/tech/2017/03/preventing-cross-site-attacks-using-same-site-cookies/`.

[Shi+23] Y. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang. "Precise (Un)Affected Version Analysis for Web Vulnerabilities". In: ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: `10.1145/3551349.3556933`. URL: `https://doi.org/10.1145/3551349.3556933`.

[Sie+22] H. Siewert, M. Kretschmer, M. Niemietz, and J. Somorovsky. "On the Security of Parsing Security-Relevant HTTP Headers in Modern Browsers". In: *2022 IEEE Security and Privacy Workshops (SPW).* 2022, pp. 342–352. DOI: `10.1109/SPW54247.2022.9833880`.

[Sin+10] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. "On the Incoherencies in Web Browser Access Control Policies". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy.* SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 463–478. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.35. URL: http://dx.doi.org/10.1109/SP.2010.35.

[SL13] N. Srndic and P. Laskov. "Detection of Malicious PDF Files Based on Hierarchical Document Structure". In: *NDSS.* 2013.

[Sma19] SmashWords. *Smashwords Distribution Network.* 2019. URL: https://www.smashwords.com/distribution.

[SNM17] J. Schwenk, M. Niemietz, and C. Mainka. "Same-Origin Policy: Evaluation in Modern Browsers". In: *26th USENIX Security Symposium (USENIX Security 17).* Vancouver, BC: USENIX Association, Aug. 2017, pp. 713–727. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk.

[Sny] Snyk. *Snyk Vulnerability DB: electron.* URL: https://security.snyk.io/package/npm/electron.

[Sol+09] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle. "Flash Cookies and Privacy". In: vol. 2010. Aug. 2009, pp. 158–163. DOI: 10.2139/ssrn.1446862.

[SRS22] P. Stolz, S. Roth, and B. Stock. "To hash or not to hash: A security assessment of CSP's unsafe-hashes expression". In: *2022 IEEE Security and Privacy Workshops (SPW).* 2022, pp. 1–12. DOI: 10.1109/SPW54247.2022.9833888.

[SS12] C. Smutz and A. Stavrou. "Malicious PDF Detection Using Metadata and Structural Features". In: *Proceedings of the 28th Annual Computer Security Applications Conference.* ACSAC '12. Orlando, Florida, USA: ACM, 2012, pp. 239–248. ISBN: 978-1-4503-1312-4. DOI: 10.1145/2420950.2420987. URL: http://doi.acm.org/10.1145/2420950.2420987.

[SSM10] S. Stamm, B. Sterne, and G. Markham. "Reining in the Web with Content Security Policy". In: *Proceedings of the 19th International Conference on World Wide Web.* WWW '10. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, pp. 921–930. ISBN: 9781605587998. DOI: 10.1145/1772690.1772784. URL: https://doi.org/10.1145/1772690.1772784.

[Sta17] C. P. Status. *'SameSite' cookie attribute.* 2017. URL: https://www.chromestatus.com/feature/4672634709082112.

[Sta19] C. P. Status. *Cookies default to SameSite=Lax.* 2019. URL: `https://chromestatus.com/feature/5088147346030592`.

[STK17] P. Snyder, C. Taylor, and C. Kanich. “Most websites don’t need to vibrate: A cost-benefit approach to improving browser security”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2017, pp. 179–194.

[Sto+17] G. Storey, D. Reisman, J. Mayer, and A. Narayanan. “The Future of Ad Blocking: An Analytical Framework and New Techniques”. In: (May 2017).

[Syn23a] Synopsys. *Chromium (Google Chrome).* 2023. URL: `https://openhub.net/p/chrome/analyses/latest/languages_summary`.

[Syn23b] Synopsys. *Mozilla Firefox.* 2023. URL: `https://openhub.net/p/firefox/analyses/latest/languages_summary`.

[ŚZZ05] J. Śliwerski, T. Zimmermann, and A. Zeller. “When Do Changes Induce Fixes?” In: *Proceedings of the 2005 International Workshop on Mining Software Repositories.* MSR ’05. St. Louis, Missouri: Association for Computing Machinery, 2005, pp. 1–5. ISBN: 1595931236. DOI: `10.1145/1083142.1083147`. URL: `https://doi.org/10.1145/1083142.1083147`.

[Tru] Trunk Based Development. *Introducion.* URL: `https://trunkbaseddevelopment.com/`.

[Van+14] T. Van Goethem, P. Chen, N. Nikiforakis, L. Desmet, and W. Joosen. “Large-scale security analysis of the web: Challenges and findings”. In: *International Conference on Trust and Trustworthy Computing.* Springer. 2014, pp. 110–126.

[Van+22] T. Van Goethem, G. Franken, I. Sanchez-Rola, D. Dworken, and W. Joosen. “SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model”. In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security.* ASIA CCS ’22. Nagasaki, Japan: Association for Computing Machinery, 2022, pp. 784–798. ISBN: 9781450391405. DOI: `10.1145/3488932.3517416`. URL: `https://doi.org/10.1145/3488932.3517416`.

[Vas+18] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. “FP-STALKER: Tracking Browser Fingerprint Evolutions”. In: *2018 IEEE Symposium on Security and Privacy (SP).* 2018, pp. 728–741. DOI: `10.1109/SP.2018.00008`.

[VHS16] S. Van Acker, D. Hausknecht, and A. Sabelfeld. "Data Exfiltration in the Face of CSP". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: Association for Computing Machinery, 2016, pp. 853–864. ISBN: 9781450342339. DOI: 10.1145/2897845.2897899. URL: https://doi.org/10.1145/2897845.2897899.

[VJ22] T. Van Goethem and W. Joosen. "Towards Improving the Deprecation Process of Web Features through Progressive Web Security". In: *2022 IEEE Security and Privacy Workshops (SPW)*. 2022, pp. 20–30. DOI: 10.1109/SPW54247.2022.9833872.

[VJN15] T. Van Goethem, W. Joosen, and N. Nikiforakis. "The Clock is Still Ticking: Timing Attacks in the Modern Web". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 1382–1393. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813632. URL: http://doi.acm.org/10.1145/2810103.2813632.

[W3C] W3C. *The history of the Web*. URL: https://www.w3.org/wiki/The_history_of_the_Web.

[W3C19] W3C. *EPUBCheck*. GitHub, 2019. URL: https://github.com/w3c/epubcheck.

[Wan+19] X. Wang, K. Sun, A. Batcheller, and S. Jajodia. "Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2019, pp. 485–492.

[WBV16] M. West, A. Barth, and D. Veditz. *Content Security Policy Level 2*. W3C Recommendation. https://www.w3.org/TR/CSP2/. W3C, Dec. 2016.

[WE20] A. Wirfs-Brock and B. Eich. "JavaScript: The First 20 Years". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386327. URL: https://doi.org/10.1145/3386327.

[Weba] WebKit. *Intelligent Tracking Prevention*. URL: https://webkit.org/blog/7675/intelligent-tracking-prevention/.

[Webb] WebKit. *Tracking Prevention in WebKit*. URL: https://webkit.org/tracking-prevention/.

[Webc] WebKit Bugzilla. *Limit user agent versioning to an upper bound*. URL: https://bugs.webkit.org/show_bug.cgi?id=180365.

[web] web-platform-tests. *web-platform-tests documentation*. URL: https://web-platform-tests.org.

[Wei$^+$16] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. "CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy". In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security.* Vienna, Austria, 2016. URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45542.pdf.

[Wes23] M. West. *Reporting security issues in specifications.* May 2023. URL: https://github.com/whatwg/meta/issues/281.

[WG16] M. West and M. Goodwin. *Same-Site Cookies.* Internet-Draft draft-ietf-httpbis-cookie-same-site-00. IETF Secretariat, June 2016. URL: https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00.

[Wi$^+$23] S. Wi, T. T. Nguyen, J. Kim, B. Stock, and S. Son. "DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing". In: *NDSS.* Feb. 2023. URL: https://publications.cispa.saarland/3891/.

[Wil17] J. Wilander. *Intelligent Tracking Prevention.* June 2017. URL: https://webkit.org/blog/7675/intelligent-tracking-prevention/.

[Wis13] R. Wischenbart. *The global eBook market: current conditions & future projections.* O'Reilly Media, Inc., 2013.

[WLR14] M. Weissbacher, T. Lauinger, and W. Robertson. "Why Is CSP Failing? Trends and Challenges in CSP Adoption". In: *Research in Attacks, Intrusions and Defenses.* Ed. by A. Stavrou, H. Bos, and G. Portokalidis. Cham: Springer International Publishing, 2014, pp. 212–233. ISBN: 978-3-319-11379-1.

[WS23] M. West and A. Sartori. *Content Security Policy Level 3.* W3C Working Draft. https://www.w3.org/TR/CSP3/. W3C, May 2023.

[WW07] J. Williams and D. Wichers. "OWASP Top 10". In: (2007). URL: https://owasp.org/www-pdf-archive//OWASP_Top_10_2007.pdf.

[WW10] J. Williams and D. Wichers. "OWASP Top 10 - 2010". In: (2010). URL: https://owasp.org/www-pdf-archive//OWASP_Top_10_-_2010.pdf.

[WW13] J. Williams and D. Wichers. "OWASP Top 10 - 2013". In: (2013). URL: https://owasp.org/www-pdf-archive//OWASP_Top_10_-_2013.pdf.

[WW17] J. Williams and D. Wichers. "OWASP Top 10 - 2017". In: (2017). URL: https://owasp.org/www-pdf-archive//OWASP_Top_10-2017_(en).pdf.pdf.

[Xia+20] G. Xiao, Z. Zheng, B. Jiang, and Y. Sui. “An Empirical Study of Regression Bug Chains in Linux”. In: *IEEE Transactions on Reliability* 69.2 (2020), pp. 558–570. DOI: 10.1109/TR.2019.2902171.

[Yen+12] T.-F. Yen, Y. Xie, F. Yu, R. (Yu, and M. Abadi. “Host Fingerprinting and Tracking on the Web:Privacy and Security Implications”. In: *The 19th Annual Network and Distributed System Security Symposium (NDSS) 2012.* Internet Society, Feb. 2012.

[Yin+11] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. “How Do Fixes Become Bugs?” In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ESEC/FSE ’11. Szeged, Hungary: Association for Computing Machinery, 2011, pp. 26–36. ISBN: 9781450304436. DOI: 10.1145/2025113.2025121. URL: https://doi.org/10.1145/2025113.2025121.

[Yu+16] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol. “Tracking the Trackers”. In: *Proceedings of the 25th International Conference on World Wide Web.* WWW ’16. Montr&#233;al, Qu&#233;bec, Canada: International World Wide Web Conferences Steering Committee, 2016, pp. 121–132. ISBN: 978-1-4503-4143-1. DOI: 10.1145/2872427.2883028. URL: https://doi.org/10.1145/2872427.2883028.

[ZAH11] S. Zaman, B. Adams, and A. E. Hassan. “Security versus Performance Bugs: A Case Study on Firefox”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories.* MSR ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 93–102. ISBN: 9781450305747. DOI: 10.1145/1985441.1985457. URL: https://doi.org/10.1145/1985441.1985457.

[ZF08] W. P. Zeller and E. W. Felten. “Cross-Site Request Forgeries: Exploitation and Prevention”. In: 2008.

[Zhe+15] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. “Cookies Lack Integrity: Real-World Implications”. In: *24th USENIX Security Symposium (USENIX Security 15).* Washington, D.C.: USENIX Association, 2015, pp. 707–721. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng.

List of publications

G. Franken, T. Van Goethem, and W. Joosen. "Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 151–168. ISBN: 978-1-939133-04-5

G. Franken, T. Van Goethem, and W. Joosen. "Exposing Cookie Policy Flaws Through an Extensive Evaluation of Browsers and Their Extensions". In: *IEEE Security & Privacy* 17.4 (2019), pp. 25–34. DOI: 10.1109/MSEC.2019.2909710

G. Franken, T. Van Goethem, and W. Joosen. "Reading Between the Lines: An Extensive Evaluation of the Security and Privacy Implications of EPUB Reading Systems". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1730–1747. DOI: 10.1109/SP40001.2021.00015

T. Van Goethem, G. Franken, I. Sanchez-Rola, D. Dworken, and W. Joosen. "SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security.* ASIA CCS '22. Nagasaki, Japan: Association for Computing Machinery, 2022, pp. 784–798. ISBN: 9781450391405. DOI: 10.1145/3488932.3517416

Y. Dimova, G. Franken, V. Le Pochat, W. Joosen, and L. Desmet. "Tracking the Evolution of Cookie-Based Tracking on Facebook". In: WPES'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 181–196. ISBN: 9781450398732. DOI: 10.1145/3559613.3563200

G. Franken, T. Van Goethem, L. Desmet, and W. Joosen. "A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3673–3690. ISBN: 978-1-939133-37-3

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
gertjan.franken@kuleuven.be
https://www.distrinet.cs.kuleuven.be/