

Computer-Aided Formal Security Analysis of the Web Platform

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dott.mag. Lorenzo Veronese

Registration Number 12007927

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Matteo Maffei

The dissertation has been reviewed by:

Prof. Musard Balliu

Dr.-Ing. Ben Stock

Vienna, September 9, 2024

Lorenzo Veronese

Erklärung zur Verfassung der Arbeit

Dott.mag. Lorenzo Veronese

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. September 2024

Lorenzo Veronese

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Matteo Maffei, for his guidance during these last four years. I would like to express my gratitude to Dr. Marco Squarcina for his support, feedback, and willingness to always make time for in-depth discussions and ad-hoc meetings. I also want to thank the reviewers of this thesis, Prof. Musard Balliu and Dr.-Ing. Ben Stock, for accepting this task and for the time spent reading and reviewing this manuscript. Thank you to Prof. Maria Christakis and Prof. Stefano Calzavara for agreeing to be part of my Proficiency Evaluation and Defense committee.

I additionally would like to thank all the researchers I collaborated with in the past few years. In particular, Pedro Bernardo, who, as a fellow PhD student, shared many of my same challenges and experiences. Thank you to Prof. Stefano Calzavara and Prof. Pedro Adão for their guidance and feedback.

Finally, I express my heartfelt thanks to all my colleagues in the Security and Privacy research unit at TU Wien. I am grateful for the countless engaging discussions and the friendly and supportive environment they created.

Kurzfassung

Im Laufe der letzten Jahrzehnte wurde eine Reihe von Standards, die von W3C, IETF und WHATWG veröffentlicht wurden, konsolidiert, um die Web-Plattform zu bilden, eine vollwertige Anwendungsplattform, im Gegensatz zum ursprünglichen Design des Webs als eine Reihe von Hyperlink-Dokumenten. Der aktuelle Stand der Standardisierung des Webs, fragmentiert in eine Vielzahl von Dokumenten, die von verschiedenen Organisationen verwaltet werden, erschwert die Überlegungen zur Sicherheit der Plattform insgesamt. Dies führte zur Einführung von Schwachstellen, die durch unvorhergesehene Interaktionen zwischen verschiedenen Webkomponenten verursacht wurden. Diese Situation ergibt sich aus der Tatsache, dass Webspezifikationen häufig informell definierte oder implizite Annahmen über die Sicherheit anderer Funktionen enthalten. In dieser Dissertation argumentieren wir für die Notwendigkeit einer strengen und formalen Definition der Web-Sicherheit im Hinblick auf Invarianten, deren Gültigkeit auf der gesamten Web-Plattform garantiert ist. In dieser Arbeit untersuchen wir insbesondere die Sicherheitsmechanismen des modernen Webs und formalisieren sie in Form von Web-Invarianten. Wir schlagen zwei Methodologien zur Validierung von Web-Invarianten in einem neuen Modell von Web-Spezifikationen (WebSpec) und außerdem in Browser-Implementierungen (Chromium, Firefox, Safari) vor, die es uns ermöglichen, neue Inkonsistenzen zu entdecken und fundierte Abhilfemaßnahmen vorzuschlagen. Anschließend konzentrieren wir uns auf die Anwendungssicherheit und untersuchen das weniger bekannte Web-Bedrohungsmodell des related-domain Angreifers und messen dessen Auswirkungen auf die Sicherheit der beliebtesten Websites im Web. Abschließend richten wir unsere Aufmerksamkeit auf Cookies und ihre lange Geschichte von Schwachstellen und diskutieren neue Verletzungen ihrer Integrität sowie neue Angriffe durch den related-domain Angreifer.

Abstract

Over the last two decades, a set of standards published by the W3C, IETF and WHATWG was consolidated to form the Web Platform, a full-fledged application platform as opposed to the initial design of the Web as a set of hyperlinked documents. The current state of the standardization of the Web, fragmented into a multitude of documents maintained by different organizations, complicates the process of reasoning about the security of the platform as a whole. This led to the introduction of vulnerabilities originating from unforeseen interactions between different Web components. This situation stems from the fact that Web specification often include informally-defined or implicit assumptions about the security of other features. In this thesis we argue for the need of a rigorous and formal definition of Web security in terms of invariants that are guaranteed to be valid across the Web platform. In particular, in this work we study the security mechanisms of the modern Web and formalize them in the form of Web invariants. We propose two methodologies for validating Web invariants on a new model of Web specifications (WebSpec) and on browser implementations (Chromium, Firefox, Safari) that allowed us to discover new inconsistencies and propose sound mitigations. We then focus on application security and study the lesser-known Web threat model of the related domain attacker, measuring its impact on the security of the most popular sites on the Web. Finally, we turn our attention to cookies and their long history of vulnerabilities, discussing new violations of their integrity protections and new attacks enabled by related-domain attackers.

List of Publications

- [VFB⁺23] Lorenzo Veronese, Benjamin Farinier, Pedro Bernardo, Mauro Tempesta, Marco Squarcina, and Matteo Maffei. WebSpec: Towards Machine-Checked Analysis of Browser Security Mechanisms. In 2023 IEEE Symposium on Security and Privacy (S&P). IEEE Computer Society, 2023.
- [BVDV⁺24] Pedro Bernardo¹, Lorenzo Veronese¹, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. Web Platform Threats: Automated Detection of Web Security Issues With WPT. In 33rd USENIX Security Symposium. USENIX Association, 2024.
- [STV⁺21] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web. In 30th USENIX Security Symposium. USENIX Association, 2021.
- [SAVM23] Marco Squarcina, Pedro Adão, Lorenzo Veronese, and Matteo Maffei. Cookie Crumbles: Breaking and Fixing Web Session Integrity. In 32nd USENIX Security Symposium. USENIX Association, 2023.
- [BSVL24] Philipp Beer, Marco Squarcina, Lorenzo Veronese, and Martina Lindorfer. Tabbed Out: Subverting the Android Custom Tab Security Model. In 2024 IEEE Symposium on Security and Privacy (S&P). IEEE Computer Society, 2024. **Not part of the thesis.**

¹Shared first authorship

Contents

| | |
|--|-------------|
| Kurzfassung | vii |
| Abstract | ix |
| List of Publications | xi |
| Contents | xiii |
| 1 Introduction | 1 |
| 1.1 Research Goals | 3 |
| 1.2 Methodology | 5 |
| 1.3 Contributions | 7 |
| 2 WebSpec: Towards Machine-Checked Analysis of Browser Security Mechanisms | 11 |
| 2.1 Introduction | 12 |
| 2.2 Browser Model | 14 |
| 2.3 WebSpec Toolchain | 19 |
| 2.4 Web Invariants and Attacks | 23 |
| 2.5 Verification of Security Properties | 31 |
| 2.6 Evaluation | 32 |
| 2.7 Related Work | 35 |
| 2.8 Conclusion | 37 |
| 3 Web Platform Threats: Automated Detection of Web Security Issues With WPT | 39 |
| 3.1 Introduction | 40 |
| 3.2 Background | 42 |
| 3.3 Web Invariants | 43 |
| 3.4 Trace Verification Pipeline | 52 |
| 3.5 Evaluation Results | 59 |
| 3.6 Related Work | 67 |
| 3.7 Conclusion | 69 |
| | xiii |

| | | |
|----------|--|------------|
| 4 | Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web | 71 |
| 4.1 | Introduction | 72 |
| 4.2 | Background | 74 |
| 4.3 | The Related-Domain Attacker | 75 |
| 4.4 | Analysis Methodology | 83 |
| 4.5 | Security Evaluation | 90 |
| 4.6 | Disclosure and Ethical Considerations | 96 |
| 4.7 | Case Studies | 98 |
| 4.8 | Related Work | 99 |
| 4.9 | Conclusion | 101 |
| 5 | Cookie Crumbles: Breaking and Fixing Web Session Integrity | 103 |
| 5.1 | Introduction | 104 |
| 5.2 | Background | 105 |
| 5.3 | Threat Model | 108 |
| 5.4 | Violating Cookie Integrity | 110 |
| 5.5 | CORF Token Fixation | 119 |
| 5.6 | Systematic Evaluation of Web Frameworks | 122 |
| 5.7 | Formal Verification of Web Frameworks | 128 |
| 5.8 | Related Work | 132 |
| 5.9 | Conclusion | 134 |
| 6 | Conclusion and Directions for Future Research | 135 |
| 6.1 | Conclusion | 135 |
| 6.2 | Directions for Future Work | 136 |
| | List of Figures | 139 |
| | List of Tables | 141 |
| | Bibliography | 143 |
| A | Appendix to Chapter 2 | 161 |
| A.1 | Web Invariants | 161 |
| A.2 | Verification of Security Properties: Preventing Cache API Access From Other Browsing Contexts | 171 |
| A.3 | Compiler | 173 |
| A.4 | Scalability | 175 |
| A.5 | Verifier Example | 176 |
| A.6 | Completeness | 177 |
| B | Appendix to Chapter 3 | 181 |
| B.1 | Encoding Known Web Invariants | 181 |
| B.2 | Test Selection | 183 |

| | |
|--------------------------------------|------------|
| C Appendix to Chapter 5 | 185 |
| C.1 Web Framework Analysis | 185 |

CHAPTER 1

Introduction

In the early days of the Web, the World Wide Web Consortium (W3C) was established to encourage the industry members to adopt standards that would allow for compatibility among Web browsers. Together with the RFC documents published in collaboration with the Internet Engineering Task Force (IETF), a set of W3C standards was consolidated over the years to form the foundation of the Web Platform. A full-fledged application platform, as opposed to its initial design composed of a web of hyperlinked documents. This paradigm-shift, however required, according to two of the more important industry players at the time (Mozilla and Opera), a corresponding shift in the development of standards, which was considered slow and susceptible to browsers independently developing incompatible solutions before jointly-developed specifications [FOS04]. This sentiment and the disagreement over the evolution of HTML between industry players and the W3C led to the establishment of the Web Hypertext Application Technology Working Group (WHATWG). The group, lead by Apple, Mozilla, Opera, and Microsoft, developed HTML5, which later became the accepted W3C standard for the next version of HTML (as opposed to XHTML), and currently maintains, among others, the *Living Standards* for HTML, DOM and Fetch. A Living Standard is continuously updated as it receives feedback from the community, with new features added over time at a rate intended to keep the standard ahead of implementations [Gro].

The current state of the standardization of the Web platform, where a multitude of documents are maintained and independently updated by different organizations, new features are added at a rapid pace to support more use cases, and new security mechanisms are introduced to mitigate potential vulnerabilities, results in an ever-growing complexity which is reflected in the complexity of browser implementations. Furthermore, the fragmented nature of the Web platform specification, where each feature is defined as a separate document, complicates the process of reasoning about the interactions between distinct Web components. Specifications are often very extensive, and make implicit assumptions about other components of the platform. Over the years, this resulted in the

introduction of vulnerabilities originating from unforeseen interactions between features both at the design level or in their implementation [SMWL10b, ABL⁺10, SBR17, TTS]. In response to these vulnerabilities new mitigations had to be introduced, in the form of explicit exceptions in existing standards or new security mechanisms (e.g., the introduction of *forbidden headers* in the Fetch standard in response to the attack presented in [SMWL10b]).

Flaws in the specifications or implementations of browsers affect the set of assumptions that developer can make while developing applications for the Web platform. For instance, the Same Origin Policy (SOP) [Bar11b] defines the primary security boundary between websites in the form of their *origin* (i.e., the scheme, host and port of the page URL). Thus, we can assume that all pages are unable to cross the origin boundary to, e.g., read the content of the user mailbox loaded in a different browser tab. The combination of all these assumptions have been abstracted in the form of three main *Web threat models*, defining the capabilities of an attacker depending on their position: a *network attacker* has access to the network where the victim is browsing the Web; a *Web attacker* controls a site on the Web; and the *related-domain attacker* controls a subdomain of a target site [CFST17].

Most of these security assumptions are, however, implicit or informally defined as part of specific security mechanism standards, and no single document defines the set of properties that should be seen as invariants in the Web. In this work, we argue that these security assumptions should be explicitly stated and rigorously specified to form the foundation of a formal definition of Web security. We name such security properties *Web invariants*, that is, *security properties of the Web platform that are expected to hold across its updates and independently on how its component interact with each other*. Web invariants are thus global, concerning the security of the platform as a whole, are proven correct (thus not assumed), and represent the set of the security assertions all applications should expect the platform to respect.

More specifically, in this thesis we study the security mechanisms of the modern Web and formalize them in the form of Web invariants. We propose two methodologies to verify such invariants, respectively, on a new model of the Web Platform specifications (WebSpec) and on the three major browser implementations (Chrome, Firefox, Safari). These verification pipelines allowed us to automatically discover inconsistencies that violate the security expectation of the Web platform and to propose sound modifications to the specifications that are guaranteed to maintain the Web security invariants. These formal guarantees are enabled by our WebSpec model, which is the most comprehensive to date for what concerns browser security mechanisms and supports machine-checked formal security proofs. Additionally, we study the lesser known Web threat model of the related domain attacker, measuring the security implications of subdomain takeover on the most popular sites on the Web. We then focus on cookies, showing how their integrity protections can be violated, enabling new classes of request forgery attacks (CORF token fixation). We finally propose a formally verified mitigation for CORF token fixation attacks enabled by a formalization of server-side Web frameworks.

1.1 Research Goals

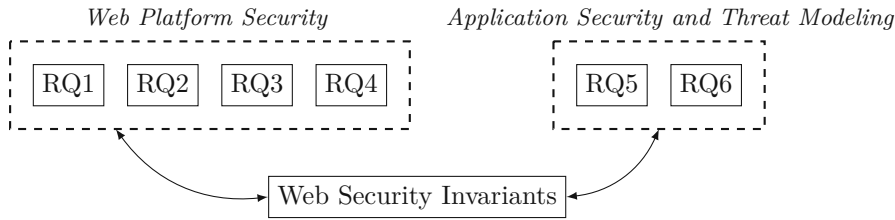


Figure 1.1: Research threads and questions

The study of the security of the Web platform cannot be performed in isolation, focusing only on browser mechanisms and their security properties, as any inconsistent behavior of the platform has an impact on the security of Web applications. Moreover, investigating how applications are developed in real-world scenarios and analyzing the threat models that are used by security experts for the analysis of websites offers insights into the security expectations and requirements of Web developers. For these reasons, we organize the work presented in this thesis to the two research threads of *Web Platform Security* and *Application security and Threat modeling* as depicted in Figure 1.1. Both research threads have the goal of defining or validating Web invariants, respectively focusing on the point of view of the browser or the applications. This highlights the role of Web invariants as a *security interface* between the platform and the consumers of Web APIs; that is, the browser offers certain security guarantees that are based on the Web applications requirements and that applications can expect to be valid.

1.1.1 Web Platform Security

Despite the efforts of the W3C consortium and browser vendors, the current standardization process for the Web platform leaves space for logical flaws to be overlooked during the manual reviewing process. Specifications of Web components are typically written informally and evaluated in isolation, thus, although their analysis is conducted by experts, corner cases may be overlooked. These flaws may be caused by implicit assumptions in the interplay between the proposed and existing Web components and security mechanisms. For this reason, it is unclear if composing the specifications that form the modern Web yields a consistent system that preserves the security properties defined in the standards. This leads to the following research question.

Research Question (RQ1). *Does the composition of the different components constituting the Web platform affect the security of the platform as a whole?*

The above research question requires a formal and rigorous definition of security, when applied to the Web platform. The security properties of the Web are either implicitly or informally defined as part of the specifications of the browser mechanisms enforcing them, and no single document defines the set of expected security properties of the platform.

It is thus essential, as emphasized in the seminal paper by Akhawe et al. [ABL⁺10], to define a set of formal properties that form the core of a more scientific understanding of Web security.

Research Question (RQ2). *Is it possible to define a set of invariants of the Web platform that hold across its updates and independently of how its components are composed?*

Once we define these invariants, we need a principled way of pinpointing logical flaws emerging from the composition of the Web components. This methodology should automatically detect inconsistencies in the standards defining Web security mechanisms, taking into account the interactions between the various security mechanisms and Web features. We thus define the following research question, which instantiates *RQ1* focusing on Web specifications.

Research Question (RQ3). *Can we design a verification pipeline to automatically detect security flaws in the specification of client-side security mechanisms?*

Assuming the specifications of the Web guarantee a specific set of security properties, additional logical flaws can be introduced by browser developer during the implementation phase. Specifications are usually written in natural language and refer to other existing Web components, thus correctly integrating them in browser code is challenging and error-prone. Additionally, given the constant evolution of the Web platform, the implementation of new updates often require non-trivial changes to existing browser components that were not originally developed with such modifications in mind. We thus define the following research question.

Research Question (RQ4). *Can we design a practical framework to formally and automatically detect security flaws in the implementation of client-side security mechanisms?*

1.1.2 Application Security and Threat Modeling

The traditional threat models for the security analysis of Web applications focused primarily on two classes of attackers. The *Web attacker*, which controls a malicious website and executes attacks by means of JavaScript (including content injection); and the *network attacker*, with the additional capability of monitoring and manipulating plain-text (unencrypted) network communication.

A lesser studied threat model is the *related-domain* (or same-site) attacker [BBC11]: a Web attacker controlling a sibling domain of their target Web application, making their domain same-site with the target. This privileged position allows related-domain attackers, for instance, to compromise the confidentiality and integrity of cookies, as cookies can be accessed and stored by domains belonging to the same site. The notion of site as a security boundary between websites has become progressively more relevant in recent years. For instance, browsers recently changed the default behavior of cookies so that they are only attached to same-site requests by default [The20, Con20]. This

highlights the importance of considering the related-domain attacker as a relevant threat in the modern Web. We thus define the following research question.

Research Question (RQ5). *How relevant is the threat posed by related domain (same-site) attackers on the Web? Can we quantify their impact on Web application security at scale?*

The current draft specification of cookies [CEWW22], which all major browsers now implement, as opposed to the 2011 standard [Bar11a], includes extended security features focused on strengthening cookie integrity (e.g. the SameSite attribute), also incorporating defenses against same-site attackers (i.e., the `__Host-` and `__Secure-` cookie name prefixes). These mechanisms have been proposed in response to the classes of attacks leveraging the weaknesses of cookie confidentiality and integrity, such as session hijacking, session fixation and cross-site request forgery (CSRF). Additional protections are implemented on the server side, where, for instance, the *synchronizer token pattern* is considered the most effective protection against CSRF [LKP21]. Given the progressively more central role of the same-site attacker threat and the long history of integrity issues affecting cookies, we question the effectiveness of these defenses, when used in combination, against related-domain attackers. Therefore, we define the following research question.

Research Question (RQ6). *How effective are the existing protections that enforce cookie integrity? What are the real-world security implications of cookie integrity issues?*

1.2 Methodology

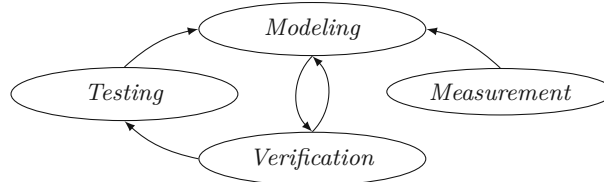


Figure 1.2: Overall Methodology

Figure 1.2 depicts the overall methodology that was employed for the work presented in this thesis. The *testing* of browser behavior and the *measurement* of the usage of Web APIs in applications inform the formal *modeling*, which includes the definition of Web invariants and the modeling of browser mechanisms or server-side applications. Models are used for *verification* of specifications or implementations and may result in security proofs, or counterexamples corresponding to attacks that can be validated by testing them on implementations.

In the following, we briefly outline the requirements and the practices that characterize each of the components of our methodology.

Modeling. The definition of Web invariants is supported by the thorough analysis of the specifications of the respective Web security mechanism and the browser code implementing it. In case the specifications provide ambiguous definitions of their security guarantees, we abstract as Web invariants the community security expectations that emerge from previous research or from our discussion with specification maintainers. Invariants are expressed in natural language and encoded in a language suitable for verification, for instance, we use the Coq logic (i.e., the Calculus of Inductive Constructions [CH86]) for model-checking against the WebSpec model and first-order logic for the verification of browser execution traces (see Section 1.3). Modeling the specifications that are included in WebSpec consists of characterizing the relevant Web component behavior at the right level of abstraction. Some specifications may, in fact, be written as a low-level step-by-step algorithm, whereas others may only present high-level requirements. For this reason, this step requires the conscious choice of abstracting low-level details when irrelevant for our model as well as making implementation decisions for underspecified behaviors.

Verification. The verification of Web invariants and the potential discovery of vulnerabilities is enabled by the use of automated theorem provers, such as the Z3 SMT solver [Res]. For the verification of server-side code, we use the WebSpi [BBM12] library for the the ProVerif protocol verifier [Bla01]. Although, automated proofs are preferred, in the case the automated solvers are unable to prove a specific property, a machine-checked proof is developed using the Coq proof assistant. The verification of invariants against the WebSpec model is supported by a compiler from the language of Coq into constrained horn clauses (CHC, a subset of first-order logic) expressed in the SMT-lib format. The ability to compile the model into an SMT formula allows us to efficiently check for counter examples or produce security proofs using Z3.

Testing. Expectations about the functionality of browser security mechanisms are tested against implementation during the modeling phase. This includes manual testing of corner cases and the validation of discovered attacks. The testing is enabled by the use of the Web Platforms Tests [WPTb] testing framework, which supports the execution of tests against the three major browsers (Chrome, Firefox, Safari). Additionally, for the purpose of guaranteeing that our WebSpec model is an accurate abstraction of the Web platform, each of the discovered invariant counterexamples is tested against multiple browser implementations.

Measurement. The study of a specific threat model or class of vulnerabilities is supported by the measurement of its prevalence on the Web. For the purpose of obtaining the most realistic picture of the state of the Web ecosystem, each measurement considers a large set of the top websites according to the Tranco [PGT⁺19] or Alexa rankings. Since these measurements may require testing for vulnerable domains, special care must be taken to not cause harm to the analyzed targets so that no site is affected by the testing. For this reason, only the precondition of the vulnerabilities are checked and no attack is executed against any public website. Moreover, when available, the use

of publicly available datasets of, e.g., open ports of all the IPv4 range [Rap20], or the HTTP archive [Arc], is preferred instead of active probing.

1.3 Contributions

Following the research threads defined in Section 1.1, we categorize the contributions of this thesis in two groups.

(Web Platform Security) First, we develop the most comprehensive model of the Web platform to date for what regards the supported security mechanisms, and a methodology for automated model checking of expected security properties (i.e., Web invariants) in the model. We then build on the developed invariants to propose a technique to check the validity of such properties on browser implementations, by validating the invariants for each concrete browser execution trace.

(Threat modeling and Application Security) Second, we study the lesser-known threat model of the *related-domain* attacker and present a measurement study of the impact of such attacker on Web security. Given the dangers posed by the related-domain attackers, we study cookie integrity issues and present a new class of vulnerabilities (CORF token fixation) that can be exploited by a related domain attacker to bypass token-based CSRF protections. We present a systematization of cookie integrity attacks and a formalization of Web frameworks to verify the correctness of our proposed mitigation against CORF token fixation attacks.

The following sections present a summary of the contributions for each chapter.

1.3.1 Model of the Web Platform

In Chapter 2, we present WebSpec, a framework for the formal analysis of browser security mechanisms. The framework includes an extensive model of the Web browser in Coq comprising a core set of Web platform components and security mechanisms (e.g., request handling, cookies, SOP, and CORS) as well as recently introduced features (e.g., CSP level 3, ServiceWorkers, and Trusted Types). WebSpec supports the definition of *Web invariants*, i.e., properties of the Web platform that are expected to be valid, and allows for machine checked proofs and model checking using the Z3 automated theorem prover. The WebSpec verification toolchain consists in a compiler and an attack trace verifier. The compiler translates the Coq browser model and the invariants into SMT-LIB queries for Z3. If a violation is found, the minimal sequence of actions leading to the attack is reconstructed and translated by the trace verifier into executable tests for cross-browser validation of the discovered inconsistency.

We define and formally encode in our model 10 Web invariants concerning the security properties of five Web components (cookies, CSP, Origin header, SOP, CORS). For five invariants WebSpec is able to discover counterexamples (i.e., the invariants do not hold

on the current Web platform) corresponding to (i) a new attack on cookies caused by the interaction with legacy APIs, (ii) a new inconsistency between CSP and a planned change to the HTML standard, and (iii) three previously reported attacks.

Using the verifier component, we validate and confirm all discovered inconsistencies against the latest versions of Chrome and Firefox, and propose new mitigations that we formally prove secure in our model.

1.3.2 Verification of Web invariants against implementations

In Chapter 3, we present a technique to formally and automatically detect security flaws in the implementation of browser security mechanisms. Specifically, we employ the Web Platform Tests (WPT [WPTb]) cross-browser testing suite to abstract browser test execution into sets of traces, that are then matched against Web invariants in order to identify the components that violate important security properties of the Web.

Our methodology consists in two main stages. First, the execution traces are collected by executing an instrumented version of Chromium, Firefox and Safari. Our instrumentation is based on combining browser extensions with network proxies for maximum visibility of browser events while maintaining cross-browser compatibility. Second, the obtained traces are translated to SMT-LIB and checked against Web invariants using an SMT solver. When the solver cannot prove the validity of an invariant for a specific browser execution trace a violation is discovered for that browser.

We define and formalize in first-order logic 9 Web invariants about Cookies and the Mixed Content policy and validate them against the 24k test included in the WPT suite. Our trace verification pipeline discovered violations in 104 tests corresponding to 10 attacks against Chromium, Firefox and Safari that were responsibly disclosed.

1.3.3 Definition and measurement of the Web threat model of the related-domain attacker

In Chapter 4, we define and measure the threat of the *related-domain* (or *same-site*) attacker to Web application security. In particular, we first systematize the attack vectors that can result in obtaining a same-site position and characterize the related-domain attacker by the capabilities they can obtain depending on the exploited attack vectors. We then develop a measurement methodology to evaluate the impact of the related-domain attacker on real-world Web applications.

Our measurement framework includes a module to detect subdomain takeover vulnerabilities deriving from expired domains, discontinued third-party services, and deprovisioned cloud VM instances. The output of this module is used to perform an analysis of the Web security implications that result from the discovered subdomains considering cookies, CSP, CORS, postMessage, and domain relaxation. Specifically, the Web security analysis aims at quantifying the number of sites that can be exploited by taking over the vulnerable

subdomains given the additional capabilities granted to a same-site attacker for each Web feature.

We perform a large-scale measurement study on the top 50K domains of the Tranco list, enumerating 26M total subdomains and discovering takeover vulnerabilities in 1520 subdomains distributed across 887 sites (most of which caused by discontinued third-party services). Our Web security analysis pictures a dire situation in which a related-domain attacker is able to violate cookie confidentiality and integrity in the majority of vulnerable sites (81%/99%), and CSP and CORS deployments offer additional attack surface compared to a traditional Web attacker.

1.3.4 Study of Cookie integrity issues on the client and server sides

In Chapter 5, we study the issues that affect the integrity of cookies and present new attacks that originate from undefined behavior in the specification, server-side parsing vulnerabilities and flaws in browser implementations. We then define the class of CORF (Cross-Origin Request Forgery) token fixation attacks, that allow related-domain attackers to bypass real-world CSRF protections, and we perform a security analysis of the top 13 Web frameworks. We finally propose a mitigation for CORF attacks that we formally verify for the affected frameworks.

We extend the threat model presented in Chapter 4 and define a taxonomy of threat models corresponding to different levels of attacker control and visibility over the network and sibling domains of a target website. This is done with the purpose of mapping attacker capabilities to concrete attacks to cookie integrity.

We perform a cross-browser evaluation of cookie integrity attacks, discovering and reporting new issues arising from (i) serialization collisions enabled by nameless cookies and the PHP cookie parsing implementation, (ii) the desynchronization between the cookies listed by the Document.cookie API and the browser cookie storage in Firefox, (iii) the chaining of standard-compliant parsers when using the AWS API gateway.

Our security analysis of the top Web frameworks uncovers CORF and session fixation vulnerabilities in 9 frameworks that were responsibly disclosed to the affected parties. We use the ProVerif protocol verifier to formally prove the correctness of our proposed mitigation of the synchronizer token pattern for each vulnerable framework.

WebSpec: Towards Machine-Checked Analysis of Browser Security Mechanisms

Abstract

The complexity of browsers has steadily increased over the years, driven by the continuous introduction and update of Web platform components, such as novel Web APIs and security mechanisms. Their specifications are manually reviewed by experts to identify potential security issues. However, this process has proved to be error-prone due to the extensiveness of modern browser specifications and the interplay between new and existing Web platform components. To tackle this problem, we developed WebSpec, the first formal security framework for the analysis of browser security mechanisms, which enables both the automatic discovery of logical flaws and the development of machine-checked security proofs. WebSpec, in particular, includes a comprehensive semantic model of the browser in the Coq proof assistant, a formalization in this model of ten Web security invariants, and a toolchain turning the Coq model and the Web invariants into SMT-lib formulas to enable model checking with the Z3 theorem prover. If a violation is found, the toolchain automatically generates executable tests corresponding to the discovered attack trace, which is validated across major browsers. We showcase the effectiveness of WebSpec by discovering two new logical flaws caused by the interaction of different browser mechanisms and by identifying three previously discovered logical flaws in the current Web platform, as well as five in old versions. Finally, we show how WebSpec can aid the verification of our proposed changes to amend the reported inconsistencies affecting the current Web platform.

This chapter presents the results of a collaboration with Benjamin Farinier, Pedro Bernardo, Mauro Tempesta, Marco Squarcina and Matteo Maffei and has been published

in the 44th IEEE Symposium on Security and Privacy in 2023 under the title “Web-Spec: Towards Machine-Checked Analysis of Browser Security Mechanisms” [VFB⁺23]. I developed the browser model and the encoding of the 10 Web invariants, performed the experimental evaluation and contributed to the development of the SMT-LIB compiler. Benjamin Farinier is responsible for the compiler, and Pedro Bernardo developed the verifier component which generates executable tests. Mauro Tempesta and Marco Squarcina contributed to the definition of the considered Web invariants and, with Matteo Maffei, were the general advisors of the work, contributing with continuous feedback.

2.1 Introduction

Web browsers are considered among the most complex software in use today, and the number of Web platform components, i.e., browser functionalities and security mechanisms, is constantly increasing. These are typically proposed by browser vendors in the form of a W3C *Editor’s Draft* and discussed within the community. If enough consensus is reached, the standardization process has to progress through several *maturity levels* before becoming a W3C recommendation.

While the implementation of new Web platform components is subject to extensive compliance testing (see, e.g., the *Web Platform Tests* project [WPTb]), their specifications undergo a manual expert review to identify potential issues: this is a continuous and extremely complex process that has to consider the interplay with legacy APIs and should, in principle, be revised whenever new components land on the Web platform.

Unfortunately, manual reviews tend to overlook logical flaws, eventually leading to critical security vulnerabilities. For example, the `HttpOnly` flag was introduced by Internet Explorer 6 [Com] as a way to protect the confidentiality of cookies with this attribute by not exposing them to scripts. Eight years after its launch, Singh et al. discovered that this property could be trivially violated by any scripts accessing the response headers of an AJAX request via the `getResponseHeader` function [SMWL10b]. Security vulnerabilities at the level of Web specifications have also affected CORS [ABL⁺10], CSP [SBR17], and Trusted Types [TTS], to name a few.

We argue that this dire situation stems from several concurring factors: (i) Web platform components are specified informally and therefore their analysis, albeit conducted by expert eyes, may easily overlook corner cases; (ii) there is no precise understanding of which security properties should be seen as invariants in the Web and, thus, be preserved by updates of the Web platform; (iii) Web platform components are typically evaluated in isolation, without considering their interactions, that is, the entangled nature of the Web platform.

Our Contributions. In this work, we advocate a paradigm shift, letting Web platform components and their interplay undergo a formal security analysis as opposed to a manual expert review. In particular, we introduce WebSpec, the first formal framework for the security analysis of browser security mechanisms that supports the automated detection

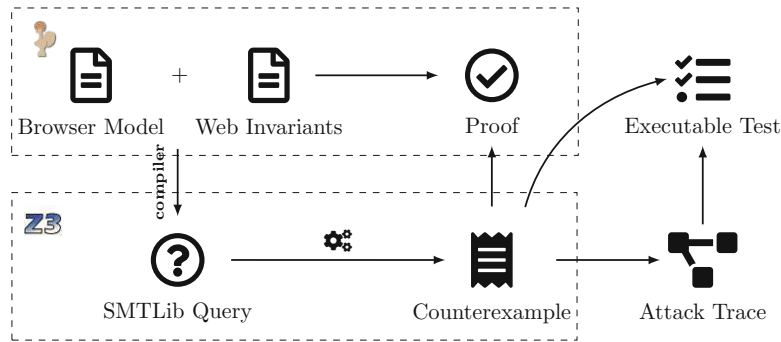


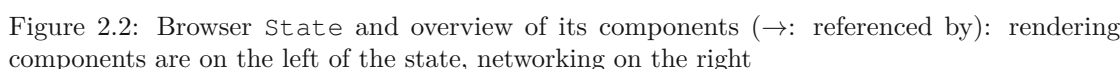
Figure 2.1: The WebSpec framework

of logical flaws as well as machine-checked security proofs. As outlined in Figure 2.1, WebSpec includes:

- a formal browser model in Coq (Section 2.2), which (i) formalizes a core set of Web platform components, including well-established (cookies, SOP, CORS, etc.) as well as recent ones (e.g., CSP level 3 and Trusted Types), and (ii) supports the definition of Web invariants, i.e., properties that are expected to hold in the Web;
- the WebSpec verification toolchain (Section 2.3), encompassing a compiler and a trace verifier. The compiler translates the browser model and the Web invariants into SMT-lib formulas to enable model checking by the Z3 automated theorem prover. A salient feature of WebSpec is the support for both bug finding and proof generation. If a violation is found, Z3 reconstructs the minimal sequence of actions leading to it, and the trace verifier displays the corresponding attack trace and maps it to executable tests in order to systematically validate the Web inconsistencies found in the model on major browsers. For Web invariants that instead hold, proofs can be directly derived by Z3 against the SMT-lib encoding or manually written and machine-checked in Coq.

We demonstrate the effectiveness of WebSpec by:

- defining ten Web invariants against which we identify (i) a new attack on cookies caused by the interaction with legacy APIs, (ii) a new inconsistency between CSP and a planned change to the HTML standard, as well as (iii) three previously reported logical flaws in the current Web platform (Section 2.4);
- validating all five Web inconsistencies against the latest versions of Chrome and Firefox;
- adjusting the model to reflect past states of the Web platform in order to identify five previously published attacks, with the goal of showing that automated security analysis would have prevented these vulnerabilities;



- ## 2.2 Browser Model

This section provides an overview of the main components of our browser model written in Coq. The model focuses on Web platform components, i.e., browser functionalities and security mechanisms, abstracting away from the network and Web servers. Our formalization enables reasoning about all possible sequences of events leading to an inconsistent state without necessarily having to model a specific Web application. We are indeed interested in proving and disproving Web invariants, i.e., *properties of the Web platform that are expected to hold across its updates and independently on how its components can interact with each other* [ABL⁺10]. Web invariants are supposed to hold for all Web applications, irrespectively of application-specific assumptions that attackers could violate. For instance, scripts in our model can, in principle, execute arbitrary sequences of any of the API calls we support, as this would be the case in presence of cross-site scripting attacks. The model also includes configuration flags that enable reasoning on former states of the Web platform or testing new proposals prior to their implementation.

The browser is modeled as a transition system in which a state evolves from an initial to a final configuration following a list of events and according to an inductive relation

named `Reachable` parameterized by a global environment:

```
Inductive Reachable : Global → list Event → State → Prop.
```

Intuitively, given a global environment `gb`, a list of events `evs`, and a state `st`, `Reachable gb evs st` means that, starting from a given initial state, `st` is reachable by executing sequentially the events in `evs` under environment `gb`.

The `Global` environment contains concrete values (e.g., the browser configuration) or symbolic variables (e.g., a set of pages) which are constant through the evolution of the browser state. An `Event` represents an atomic action that modifies the state, e.g., sending a network request or updating the DOM, which may originate from different sources, such as the browser itself, a script, or a service worker. A `State` (Figure 2.2) is a collection of datatypes used to model browser components.

Based on these ingredients, we formalize Web invariants within our model as follows, where hypothesis and conclusion are predicates that may refer to the global environment, past events, or the current state of the browser:

```
1 Parameter hypothesis : Global → list Event → State → Prop.
2 Parameter conclusion : Global → list Event → State → Prop.
3
4 Definition Invariant (gb: Global) (evs: list Event) (st: State) := Reachable gb evs st →
   hypothesis gb evs st → conclusion gb evs st.
```

In Section 2.4, we introduce ten Web invariants and discuss their security implications caused by the interplay of different Web platform components.

2.2.2 Page Rendering

The main component used to model the rendering functionality is the `Window` datatype. `Window` represents a window in terms of browsing context [WHAb, §7.1], i.e., an environment in which the browser displays a document. The field `wd_location` is the URL being visited and `wd_document` contains the displayed document. Since a `Window` can represent either a top-level window or a frame, `wd_parent` contains an optional index which, if empty, denotes a top-level window or points to the parent frame otherwise. Similarly, `wd_initiator` contains an optional index which is used to track the source browsing context of this window [WHAb, §7.11] by storing a reference to the window responsible for starting the navigation.

The `Document` datatype represents a Web page loaded and rendered in a browser window. When a page is loaded, `dc_html` represents the HTML code of the response, while `dc_dom` contains the rendered elements of the page. Static elements, e.g., forms and possibly other markup tags, are rendered immediately. Subresources of the page, such as frames and scripts, require an additional request to be included in `dc_dom`. For instance, the presence of a `HTMLFrame` in `dc_html` might cause three additional events to be executed in sequence: a request (`EvRequest`), a response (`EvResponse`), followed by the update of the DOM (`EvDOMUpdate`) resulting in `DOMFrame` being added to `dc_dom`. This approach

enables fine-grained modeling of the rendering process of the browser. In particular, our model captures the order in which resources are loaded and the presence or absence of specific elements.

WebSpec currently supports forms with the `method` and `action` attributes, images, scripts, and frames. Rendered frames in `dc_dom` contain a reference to the corresponding Window, reflecting the tree-like structure of the DOM. Finally, Document also includes the list of headers (`dc_headers`) in the HTTP response used to render the page and `dc_domain`, an optional field used to model domain relaxation via the `document.domain` API (Section 2.2.5).

2.2.3 Networking and Cookies

The main component used to model the networking functionality is the `FetchEngine`, which abstracts network access and is responsible for sending requests and receiving responses. `ft_request` contains the last emitted request, and `ft_emitter` maps to the originator of the request, i.e., whether the request is top-level or generated by the inclusion of subresources, issued by a script, a form, a worker, or it is a CORS preflight. `ft_response` is a field that either contains the corresponding response or is empty if the request is still pending. Finally, we store `Emitter * Request * Response` triples in `ft_history` to track previous network accesses.

The modeling of Request and Response is rather straightforward, as shown in Figure 2.2. We support requests and responses through HTTP and HTTPS protocols. For requests, we model the HTTP methods `GET`, `POST`, `PUT`, `DELETE`, and `OPTION`. Concerning responses, the following HTTP status codes are supported: (i) `200 OK`, successful response, (ii) `204 No Content`, successful response with an empty body, (iii) `302 Found`, redirection with no integrity guarantees in the redirected request over the HTTP method and the body of the original request [Mozd], (iv) `307 Temporary Redirect`, redirection enforcing that the method and body of the original request are preserved in the redirected one.

Supported headers are `Origin`, `Cookie` and `Referer` for requests, and `Content-Type`, `Set-Cookie`, `Location` and `Referrer-Policy` for responses. To support CSP and CORS, we include the `Content-Security-Policy` and `Access-Control-Allow-Origin` headers.

Cookies are stored in the `CookieJar` as a list of triples in the form `Domain * Cookie * CookieAttribute`, where `Domain` represents the host setting the cookie, `Cookie` is a pair corresponding to the name of the cookie and its value, and `CookieAttribute` is a record containing the attributes. Modeled cookie attributes are `Domain`, `Path`, `Secure`, `HttpOnly`, and `SameSite`. We also support cookie prefixes which enforce additional constraints [CEWW22]: (i) `__Secure-` cookies must be set with the `Secure` attribute and from a page served over HTTPS; (ii) `__Host-` cookies have all the constraints of the `__Secure-` attribute, plus the `Path` attribute must be set to the value `"/"` (ensuring that cookies will be attached to all requests) and must not contain a `Domain` attribute, thus restricting the scope to the host that set it.

2.2.4 Additional Features

Starting from the core functionalities discussed in the previous sections, our model can be extended to support other Web components, including novel security mechanisms that could benefit from the automated formal analysis enabled by WebSpec. We discuss in the following six additional Web components supported by our model.

Content Security Policy. The *Content Security Policy* (CSP) allows Web developers to tighten the security of Web applications by controlling which resources can be loaded and executed by the browser. Originally, the CSP was designed to mitigate content injection vulnerabilities. Subsequently, it was extended to restrict browser navigation (e.g., `form-action`, `frame-ancestors`) and protect DOM XSS sinks (via `trusted-types`). A CSP policy consists of a set of directives and source expressions specifying an allow-list of actions the page is allowed to perform. Currently modeled CSP directives are: `script-src`, which defines the allow-list for JavaScript sources, and `trusted-types` together with `require-trusted-types-for` for Trusted Types [KW] support, as explained later in this section.

Service Workers and Cache API. A service worker [W3C19] acts as client-side proxy between Web applications and the network. Web applications are required to register a service worker, binding it to a specific origin and a scope. When enabled, a service worker can intercept and modify HTTP requests and corresponding responses initiated by its origin. Furthermore, service workers are activated also by cross-origin requests towards their registering origin. Using the Cache API, a service worker can store HTTP responses and serve them even when the network is unreachable. We reflect these capabilities in our model by considering a specific kind of service worker that can perform fetch requests, serve synthetic responses, and cache pairs of requests and responses, regardless of the scope. To this end, we also model a lightweight Cache API and assume that service workers have arbitrary access to it.

Local Scheme URLs. We model requests to local scheme URLs [WHAa], i.e., URLs with a local scheme such as `data:` and `blob:`, as virtual requests that do not generate a response from the network. We partially support the File API [W3Cb] by enabling the creation of blob URLs via the `URL.createObjectURL` JavaScript method. We assume that local URLs are accepted interchangeably with remote URLs, meaning that they can be navigated by frames, or included as a script in a page. We thoroughly discuss the interaction between Content Security Policy and local scheme URLs in Section 2.4.3.

Local Storage. The Web Storage API [WHA_b, §12] enables JavaScript to store and retrieve key/value pairs in the browser. The API provides two mechanisms to store data: `sessionStorage`, an ephemeral storage that expires when the browser or the page is closed, and `localStorage`, which persists in the browser unless cleared explicitly. As we are interested in capturing single browser sessions, the difference between the two mechanisms is irrelevant. For this reason, we model only `localStorage`, providing methods to read and write data in the browser storage from any scripts.

Web Messaging. Cross-origin communication is enabled by the `postMessage` API [WHAb]. As we are interested in modeling messages that are sent and received—while we ignore messages that do not reach the destination—we encode the sending and receiving of a message as a single action. We also model the origin validation process performed in the receiving script. This way, we can capture potential security issues due to cross-origin messages processed without validating the sender’s origin [SS20].

Trusted Types. Trusted Types are an experimental security mechanism designed to prevent DOM XSS by restricting injection sinks to accepting only non-spoofable typed values in place of strings [KW]. These types can be created based on application-defined policies, allowing developers to specify via JavaScript a set of rules to protect injection sinks. Trusted Types are controlled by two CSP directives: `require-trusted-types-for 'script'`, which enables the enforcement of Trusted Types, i.e., instructs the browser to only accept Trusted Types for all DOM XSS injection sinks, and `trusted-types`, optionally followed by the name of one or more policies, which specifies the policies that are allowed to create Trusted Types objects. When no name is specified or when the special value `'none'` is used, no policy, and thus no Trusted Types, can be created, effectively disabling all DOM XSS sinks. We model the enforcement of Trusted Types on a page by mandating scripts to invoke the Trusted Type API to create a `TrustedHTML` object, and use it to modify the DOM via the `Element.innerHTML` property. Although we do not model the content of policies, we encode the ability to disallow the creation of any Trusted Types via the CSP directive `trusted-types 'none'`.

2.2.5 JavaScript

Contrary to previous works [Boh12], instead of modeling scripts with an internal state and precise small-step semantics, we model them in terms of actions that the browser can perform. Since we are not interested in application-specific behavior, our abstraction captures the execution of sequences of Web API calls and the evolution of the browser’s state. For example, we model the fact that a script can set a cookie, or add a request-response pair to a cache using the Cache API, but we do not model how cookie data, requests, or responses are built. Instead, we introduce symbolic variables with constraints following the API specification.

Scripts in our model can update the DOM, set and get cookies using `Document.cookie`, or navigate frames using the `Window.location` setter. They can use the Fetch API to perform network requests and read the corresponding responses up to SOP constraints, including support for CORS. We also model the legacy `Document.domain` API, which allows for cross-origin communication between windows in the same site by relaxing their `document.domain` property to a common ancestor. Although this API has been deprecated due to security concerns [Mozc], and Google announced that it will be disabled by default starting from Chrome 109 [Kit], it is still supported by all major browsers. Scripts can also use the APIs described in Section 2.2.4: they can update a cache from page

context using `Cache.put`, communicate with other windows using `Window.postMessage`, create blob URLs with `URL.createObjectURL`, and create Trusted Types.

2.3 WebSpec Toolchain

In the following, we present the WebSpec toolchain (Figure 2.1). This toolchain comprises (i) a compiler (Section 2.3.1) which translates the browser model and the Web invariants written in Coq into a query that can be automatically checked by an SMT solver – Z3 in the current implementation (Section 2.3.2) and (ii) a verifier which reconstructs from a SMT solution an attack trace enjoying correctness and minimality, and validates it against real Web browser (Section 2.3.3). Finally we discuss the advantages of our approach as compared to prior work (Section 2.3.4).

2.3.1 Compilation

To automatically verify the (in)validity of our invariants, we developed a compiler that translates the Coq model and the invariants into SMT-lib formulas, which are then fed to the Z3 solver. Technically, we compile Coq inductive types into CHC logic, i.e., first-order logic with fixed-points expressed in terms of Constrained Horn Clauses [HBdM11, HB12], in order to find inhabitants of the translated inductive types, i.e., terms of these types. In particular, the compiler translates `Reachable` and all the inductive types of kind `Prop` involved in the definitions of the query into relations expressed in terms of Horn clauses, while the remaining inductive types, including the browser state, the list of events, and the global environment, are instead translated into SMT datatypes. Note that existing tools for automation in Coq such as CoqHammer [CK18, Cza20] and SMTCoq [AFG⁺11] do not satisfy our needs: indeed, despite both relying on SMT solvers, they focus respectively on proof reconstruction and constraint solving, but not on inhabitant finding. We refer the interested reader to Appendix A.3 for a discussion of the fragment of the Coq logic supported by our compiler and for further details about the compilation pipeline.

For every Web invariant that we aim to verify in our model, we define a corresponding query as a Coq inductive type that is satisfied if a counterexample to the invariant is found in any of the states reachable from the initial browser state. For example, let us consider the following invariant:

Invariant. *Cookies with the Secure attribute can only be set (using the Set-Cookie header) over secure channels.*

We encode this invariant in WebSpec as follows:

```

1 Definition SecureCookiesInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ rp corr _evs cookie,
3     Reachable gb evs st →
4     evs = (EvResponse rp corr :: _evs) →
5     rp_hd_set_cookie (rp_headers rp) = Some cookie →
6     sc_secure cookie = true →
7     url_protocol (rp_url rp) = ProtocolHTTPS.
```

This definition says that for every reachable state where the browser handles a network response, i.e., the state is `Reachable` and the current event is `EvResponse` (lines 2-4), if the response contains a `Set-Cookie` header (line 5) with a cookie that has the `Secure` attribute (lines 6), then the protocol used to serve the response is `HTTPS` (line 7).

We encode a query for finding a counterexample to this invariant with the following Coq inductive type:

```
1 Inductive SecureCookiesQuery (gb: Global) (evs: list Event) (st: State) : Prop :=
2 | Query_state : ∀ rp corr _evs cookie,
3   Reachable gb evs st →
4   evs = (EvResponse rp corr :: _evs) →
5   rp_hd_set_cookie (rp_headers rp) = Some cookie →
6   sc_secure cookie = true →
7   url_protocol (rp_url rp) ≠ ProtocolHTTPS →
8   SecureCookiesQuery gb evs st.
```

This inductive type definition is essentially identical to `SecureCookieInvariant`, except the negation of the conclusion (line 7, we require the protocol to be \neq `HTTPS`). The following theorem formalizes that inhabitants of `SecureCookiesQuery` are indeed counterexamples of `SecureCookiesInvariant`:

```
1 Theorem secure_cookies_query_invalidates_invariant :
2   ∀ gb evs st, SecureCookiesQuery gb evs st →
3     not (SecureCookiesInvariant gb evs st).
```

2.3.2 SMT Solving and Trace Reconstruction

The compilation of the inductive `Reachable` relation results in a recursive CHC formula, which cannot be handled by standard SMT solvers. Therefore we use the μZ extension (satisfiability modulo least fixed-points [HBdM11]) of the Z3 theorem prover. More precisely, we run in parallel the Spacer engine of μZ , a generalized property-directed reachability (GPDR) model checker suitable for finding proofs [HB12], and the bounded model checking (BMC) engine of μZ , designed to find counterexamples. The four possible outcomes are:

SAT μZ finds a counterexample, hence the invariant does not hold. We discuss in Section 2.4 the security implications of violating an invariant.

UNKNOWN μZ fails to find a counterexample or to prove its absence. In such a case, which never happened in our case studies, we cannot draw any conclusion.

UNSAT μZ proves that there is no counterexample. Although this does not formally suffice to conclude that the invariant holds in our model since neither our compiler nor μZ are formally verified, this gives us strong confidence that this is the case. A formal proof in Coq can be manually produced, if stronger confidence is needed.

LOOP μZ does not terminate. Due to the way the BMC engine works, this means that μZ did not find a counterexample after exploring a certain number of steps. When this

number becomes high enough, though it is not a proof, it gives us a good intuition that the invariant is likely to hold, and hints it is worth starting a formal proof, as shown in Section 2.5.

When running WebSpec on `SecureCookiesQuery`, Spacer proves that there is no counterexample within 2min, while in the same time, BMC reaches a trace size of 50 events without detecting any attack. Moreover, we constructed a formal proof in Coq that the invariant indeed holds [Webb].

In the case μZ finds a counterexample (SAT), we first automatically extract an attack trace from it. It is worth noting that this attack trace enjoys the property of being minimal. This property is due to the decision procedure implemented in the BMC engine of μZ , which ensures that the list of events in the counterexample is the smallest one that leads to a contradiction of the invariant. Then, we verify its correctness by automatically translating it back into a Coq term and checking whether the trace produces an inhabitant of the query. We take this precaution because, as mentioned previously, neither our compiler nor μZ are formally verified. Since μZ instantiates all symbolic variables, the proof is straightforward and mostly automatic. WebSpec then renders this trace as a sequence diagram, making the representation of the counterexample accessible to users unfamiliar with formal verification. Examples of such diagrams are given in Section 2.4.

2.3.3 Trace Validation

In addition to rendering sequence diagrams, WebSpec includes a verifier to validate the discovered attack traces against real-world browser implementations (Chrome, Firefox). Our verifier consists of approximately 3500 lines of OCaml code and leverages the Web Platform Tests (WPT) [WPTb] cross-browser framework to map attack traces to tests, enabling verification against all major browsers. WPT is the standard test suite for browser and specification developers. It allows browser vendors to write tests modeling the expected behavior of Web standards and test their implementations for compliance. Using the common format specified by the WPT framework makes WebSpec tests compatible with the test suite. This allows for easy triage of the attack traces extracted from counterexamples and the inclusion of our cross-component tests into WPT.

The generated tests translate trace events to browser actions and server responses. These actions modify the browser state to match the model's state after a given event. To maintain the browser and model states consistent throughout test execution, the verifier ensures these actions execute in the correct order. The effects of these actions are collected (directly or indirectly) and verified via WPT assertions. If all assertions succeed, the test is *passing* and the trace is considered valid, whereas the test is *failing* if one assertion fails. We map each event to a $\langle \text{Setup}, \text{Action}, \text{Verification} \rangle$ (SAV) tuple:

Setup. A set of pre-conditions necessary to execute an event, e.g., for `EvWorkerCacheMatch` to happen, a service worker must be installed in the matched URL's scope.

Action. Any browser or server action, from top-level navigations and JavaScript API calls to server responses. Actions can be implicit or explicit: *Explicit* actions require an explicit API call or client-originated event, like window navigation; *Implicit* actions are actions triggered by other actions (implicit or explicit), like subresource loading.

Verification. A means to ensure that an *action* succeeded. *Explicit* verifications map to WPT assertions and are the primary mechanism for verifying attack traces. Asserting the value of a cookie after an `EvScriptSetCookie` event is an example of an explicit verification. *Implicit* verifications do not require a corresponding WPT assertion. Server responses are an example of actions with implicit verifications, as they are not browser-dependent behavior and are known before runtime.

Since events are state transitions in the model, corresponding *Actions* must change the concrete state of the browser accordingly. By enforcing that *Actions* occur in the correct order, we guarantee that the model and the concrete browser's state match throughout the test execution, reaching a final state that, e.g., violates a Web invariant. Precisely mapping trace events to SAV tuples allows us to generate executable tests that can perform and verify each event in a trace, thus ensuring the correspondence between test executions and traces. We refer to Appendix A.5 for a detailed example of how an attack trace presented in Section 2.4 is mapped to an executable test.

The following three paragraphs discuss some non-trivial implementation details necessary for trace verification:

Script Construction and Serialization. In our model, an `EvScript*` event is associated with a `DOMPath` (Figure 2.3) which identifies the `script` element performing the *action* in the current window's DOM. The verifier keeps an ordered list of `EvScript*` events for each script element in a trace. Since each `EvScript*` event maps to a specific sequence of JavaScript methods, these events can be serialized. This serialization is also performed in order, adding verification and synchronization code between event actions when necessary.

Event Synchronization. To keep tests as close as possible to the attack traces, the verifier must ensure that the actions corresponding to trace events are executed in the correct order. For instance, assume that a script in a page caches a response to an HTTP request, and such entry in the cache is subsequently processed by a service worker after matching on the corresponding request. Without synchronization, the service worker matching could occur before the cache update, leading to inconsistent test results. Ordering is implemented by enforcing explicit dependencies between actions, e.g., action a_j must be preceded by action a_i . This dependency relation is implemented by writing a nonce to a global key-value storage from a_i . Then, a_j is allowed to execute only after successfully reading the same nonce from the storage.

Content-Security-Policy Inference. Invariants regarding the CSP impose a challenge on verification as browsers do not allow direct access to the CSP via JavaScript. Therefore,

to verify the value of the CSP of a given browsing context, we must infer it. The verifier generates a set of URLs that the CSP should allow or block and calculates a signature representing this allow/block pattern. Pages updated via a `EvScriptDOMUpdate` then attempt to load scripts from these URLs. The verifier adds a `report-uri` field to the CSP for the server component to know both allowed and blocked requests. The server can then calculate the CSP signature and report it to the main page, which asserts its value. If it matches, we conclude that the CSP matches the one in the trace.

2.3.4 Discussion

The choice of Coq for the formalization of a Web browser brings two advantages: First, using a strictly-typed higher-order language as a specification language makes it possible to write expressive and parametrizable models which can easily and consistently be extended to new Web features. Second, having our model specified within a proof assistant allows us to write fully machine-checked proofs when the highest level of confidence is required. We developed such proof for the four changes we propose to fix the vulnerabilities that are currently affecting the Web platform.

In general, the main drawback of a model written in Coq is the lack of automation, which becomes particularly problematic when the model is constantly evolving to reflect the regular Web platform updates. Compiling our Web browser model from Coq to SMT queries circumvents the issue by providing not only automatic counterexample finding as in [ABL⁺10], but also automated proofs that counterexamples do not exist.

Finally, a common limitation of model-based security analysis is the lack of evidence of compliance between models and reality. In WebSpec, we avoid this pitfall by using of a verifier (Section 2.3.3) which allows us to execute and validate the discovered attack traces against real browsers. In particular, we are able to validate the five vulnerabilities found in our model of the current Web platform by running the attack traces produced by WebSpec against the latest versions of Chrome and Firefox. Additionally, the verifier enables automated testing of the semantic rules of the browser model, ensuring that our model matches the observable behavior of real browsers. For every Web component in our model, we query for a reachable state which makes use of the modeled feature. Then we validate, using the verifier, that the obtained state maps to a reachable concrete state across browser implementations. Although this does not correspond to a correctness proof, it provides empirical evidence that our modeling is consistent with browser behavior.

2.4 Web Invariants and Attacks

We define 10 Web invariants concerning the security properties of 5 core Web components: cookies, CSP, Origin header, SOP, and CORS. Table 2.1 presents an overview of the invariants that we formally encode in our model. For each invariant that does not hold in the current Web platform, WebSpec is able to find a counterexample that translates to a concrete attack. When the invariant holds, WebSpec can be configured to reflect a

2. WEBSPEC: TOWARDS MACHINE-CHECKED ANALYSIS OF BROWSER SECURITY MECHANISMS

Table 2.1: Web Invariants.

| Feature | Invariant | Description | Holds | References |
|---------|-----------|---|-------|------------------------------|
| Cookies | I.1 | Integrity of Secure cookies (network) | ● | [CEWW22] |
| | I.2 | Confidentiality of HttpOnly cookies (Web) | ● | [SMWL10b, Bar11a] |
| | I.3 | Integrity of <code>__Host-</code> cookies | ○ | [SMWL10b, Wes] |
| CSP | I.4 | Interaction with SOP | ○ | [W3Ca, SBR17] |
| | I.5 | Integrity of server-provided policies | ○ | [SCM21] |
| | I.6 | Access control on Trusted Types DOM sinks | ○ | [KW] |
| | I.7 | Safe policy inheritance | ● | [W3Ca] |
| Origin | I.8 | Authenticity of request initiator | ● | [ABL ⁺ 10, W3C14] |
| CORS | I.9 | Authorization of non-simple requests (i) | ● | [ABL ⁺ 10, W3C09] |
| | I.10 | Authorization of non-simple requests (ii) | ● | [ABL ⁺ 10, WHAb] |

● The invariant holds in the current version of the Web platform but a planned modification will invalidate it.)

past state of the Web that was affected by a vulnerability, confirming that our approach can identify previously reported attacks.

In this section, we focus on three invariants that do not hold in the current Web platform, showing how WebSpec is able to discover a new attack on the `__Host-` prefix for cookies as well as a new inconsistency between the Content Security Policy and a planned change in the HTML standard. We also present an attack against Trusted Types for which we propose a mitigation in Section 2.5. We illustrate the encoding of the full set of invariants in Appendix A.1.

2.4.1 Integrity of `__Host-` Cookies

The invariant stipulates that `__Host-` cookies ensure integrity against same-site attackers [STV⁺21]. When a cookie whose name starts with `__Host-` is set, the browser verifies that the Domain attribute is not present and discards the cookie otherwise, thus marking all `__Host-` cookie as host-only.

Invariant I.3. *A `__Host-` cookie set for the domain d can be set either by d (via HTTP headers) or by scripts included by the pages on d .*

We encode this invariant by splitting the two cases in which a host cookie can be set:

(i) via HTTP headers, and (ii) via JavaScript. We present case (ii) below and refer to Appendix A.1.1 for the full definition.

```

1 Definition HostInvariantSC (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ pt sc ctx c_idx cookie cname h _evs,
3     Reachable gb evs st →
4     (* A script is setting a cookie *)
5     is_script_in_dom_path gb (st_window st) pt sc ctx →
6     evs = (EvScriptSetCookie pt (DOMPath [] DOMTopLevel) c_idx cookie :: _evs) →
7     (* The cookie prefix is __Host *)
8     (sc_name cookie) = (Host cname) →
9     (* The cookie has been set in the script context *)
10    url_host (wd_location ctx) = Some h →
11    (sc_reg_domain cookie) = h.

```

For every reachable state in which a script `sc` is setting a cookie on the top-level window (lines 3-6), `ctx` is the window (browsing context) in which the script `sc` is running (line 5). If the cookie has the `__Host-` prefix (line 8), we require (line 11) the domain on which the cookie was registered to be equivalent to the domain of the `ctx` browsing context. This corresponds to stating that a script running on a page of domain d can set a host-prefix cookie only for the domain d .

Attack. When we run the query, our toolchain discovers a novel attack that breaks the invariant using domain relaxation. A script running on a page can modify at runtime the *effective* domain used for SOP checks through the `document.domain` API. Indeed, the value of `document.domain` is taken into account only for DOM access. All remaining access control policies implemented in the browser, e.g., for cookie jar access, XMLHttpRequests, and the origin information reported when performing a `postMessage`, use the original domain value [SMWL10b]. The mismatch between the access control policies in the DOM and the cookie jar allows a script running in an `iframe` to access the `document.cookie` property of the parent page when both pages set `document.domain` to the same value. Once the inner frame performs a set cookie of a host-prefix cookie through the parent page DOM, the browser uses the original domain value of the parent page to perform the host prefix checks, breaking the invariant.

The trace generated by WebSpec is shown in Figure 2.4 and detailed below. In the following, expressions of the form `DOMPath _` represent a unique path in the DOM. In particular, the first argument of `DOMPath` is the nesting level. For instance, we refer to the window loaded inside two nested `iframes` as `DOMPath [1,3] _`, where 1 and 3 are the indexes of the DOM elements representing the frames. The second argument is used to refer to a specific DOM object (`DOMIndex`) or to the whole document loaded in the frame (`DOMTopLevel`). An example is shown in Figure 2.3: the path to an image at index 3 loaded inside two nested `iframes` (respectively at index 2 and 1) is represented as `DOMPath [1,2] (DOMIndex 3)`, while the path of the window containing the image is `DOMPath [1,2] DOMTopLevel`.

The attack trace describes the following scenario: (steps 1-3) a page from `origin_1` is loaded in the top-level window of the browser. Note that `origin_1` is the subdomain 16162 of the host 13, loaded via HTTPS; (4-6) an `iframe` element is loaded from

2. WEBSPEC: TOWARDS MACHINE-CHECKED ANALYSIS OF BROWSER SECURITY MECHANISMS

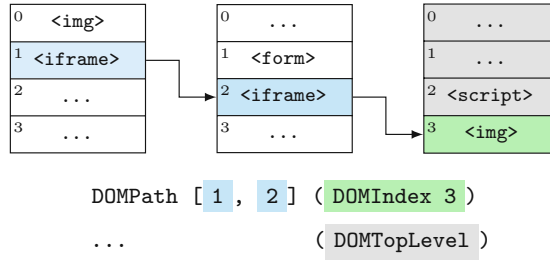


Figure 2.3: DOM Path Datatype

origin_4 at index 0 of the DOM in the main window (in the path `DOMPath [] (DOMIndex 0)`). origin_4 is another subdomain of the same host; (7-9) a script is loaded in the main window at index 1; (10-12) a script is loaded in the iframe at index 0 (`DOMPath [0] (DOMIndex 0)`); (13) the script in the main window sets its `document.domain` to its parent domain; (14) the script in the iframe sets its `document.domain` to its parent domain. The two pages are now effectively same origin, having performed domain relaxation to the same domain; (15) the script in the iframe (`DOMPath [0] (DOMIndex 0)`) sets a cookie using `document.cookie` of the top-level window (`DOMPath [] DOMTopLevel`). The cookie has the `__Host-` prefix and has been set by origin_1 for origin_4, breaking the invariant.

Although the current Web platform is still vulnerable to the attack, discontinuing the `document.domain` API will eventually make the invariant hold. WebSpec can reflect this change by specifying `c_domain_relaxation (config gb) = false`, allowing us to verify that the invariant holds.

2.4.2 Access control on Trusted Types DOM sinks

Trusted Types allow for *locking down* a document by disabling DOM XSS sinks entirely. This special setting corresponds to the following invariant.

Invariant I.6. *If a page has both `trusted-types`; and `require-trusted-types-for 'script'`; directives in the CSP then no script in the page can modify the DOM using a Trusted Types sink.*

We encode the invariant in our model as follows:

```
1 Definition TTInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ pt target_pt target_ctx ssrc ttypes,
3     Reachable gb evs st →
4     (* The target context has Trusted-Types enabled *)
5     url_protocol (wd_location target_ctx) = ProtocolHTTPS →
6     rp_hd_csp (dc_headers (wd_document target_ctx)) = Some
7       { | csp_script_src := ssrc; csp_trusted_types := Some ttypes | } →
8     tt_policy ttypes = Some None →
9     tt_require_for_script ttypes = true →
10    (* No script can update the dom using innerHTML *)
11    not (In (EvScriptUpdateHTML pt target_pt target_ctx) evs).
```

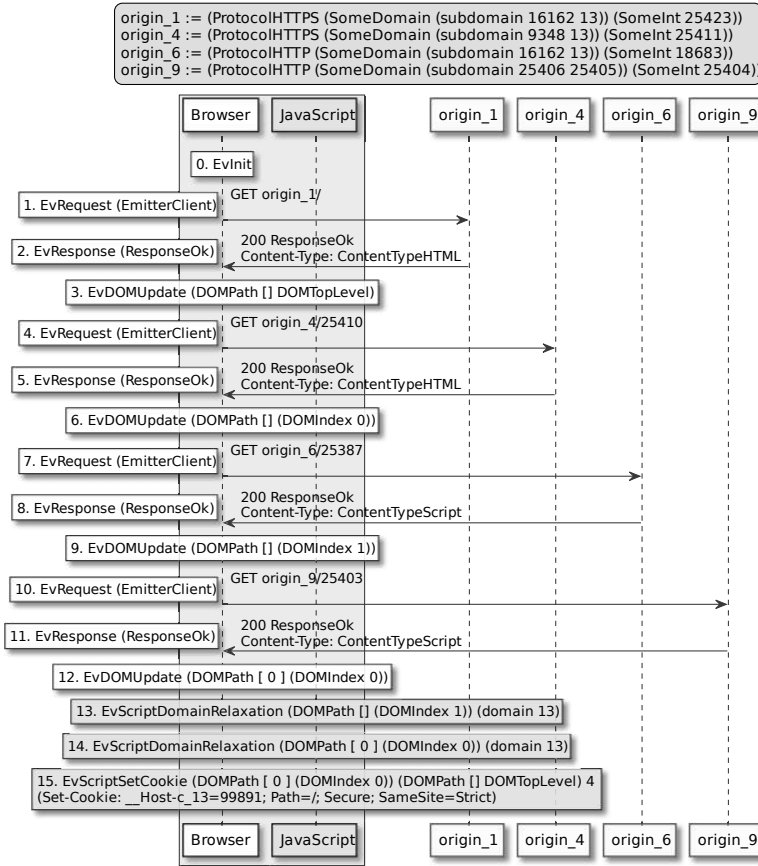


Figure 2.4: Host Cookies Inconsistency

Here we assert that there cannot be an html update event (using a DOM XSS sink, e.g., `innerHTML`) for the window `target_ctx`, if the aforementioned directives are used to define the policy for `target_ctx`.

Attack. An earlier version of the Trusted Types draft [KW, Editor’s Draft, Feb. 3, 2021] restricted Trusted Types to Secure Contexts only. This was part of an effort of browser vendors to restrict all new APIs to secure contexts to help advance the Web platform to default to the HTTPS protocol. The restriction, however, enabled attackers to bypass Trusted Types by framing the protected page from a non-secure context [TTS]. This silently disabled the DOM XSS protection despite the fact that the document was downloaded using a secure connection. When we enable the secure context restriction in our model, WebSpec is able to rediscover the bypass.

We can disable the secure context restriction with the `c_restrict_tt_to_secure_contexts` (config gb)= `false` configuration option. However, when we rerun the solver with this configuration, our toolchain can still find a counterexample for which the invariant does not hold. The trace is shown in Figure 2.5: (steps 1-3) a page protected with Trusted

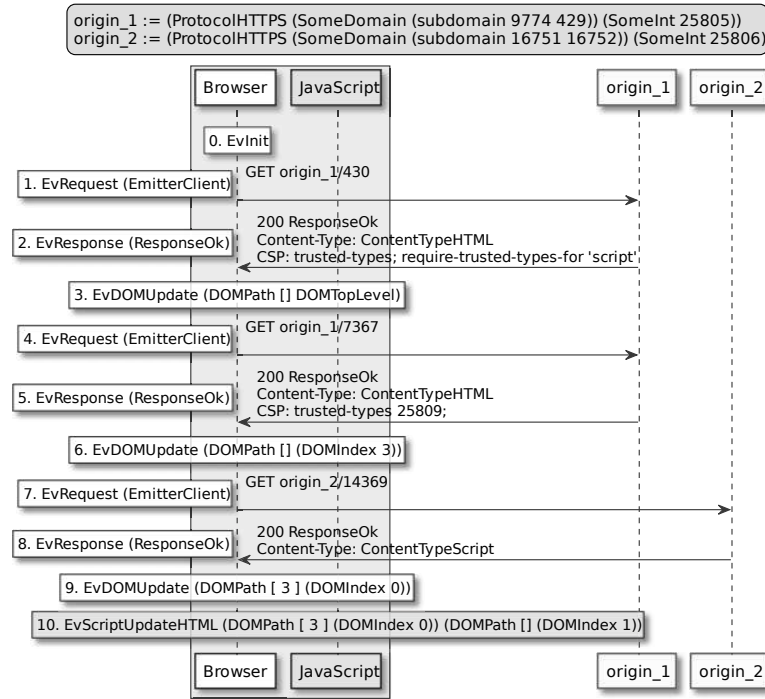


Figure 2.5: Trusted Types bypass with same-origin iframes

Types is loaded from `origin_1`. In particular, no policy is allowed, so no Trusted Type can be created; (4-6) the page contains a same-origin iframe which specifies a Trusted Types policy (`trusted-types 25809`), allowing scripts loaded in this iframe to create Trusted Types using a policy named `25809`; (7-9) a script that is loaded in the iframe modifies (10) the DOM of the parent frame using a Trusted Types sink. This is possible because the inner frame is able to create Trusted Types that are accepted by all DOM XSS sinks and because, being same origin, the inner frame can access the DOM of the parent. A similar attack on related domains is possible if the parent page performs domain relaxation, as the value of `document.domain` is used for DOM access control.

The Trusted Types draft [KW, §5.1] includes a brief discussion of a similar attack in which cross-document import of nodes would bypass the enforcement of the policy. However, the current specification does not provide any solution and suggests that other mechanisms like *Origin Policy* [DW] might be used to ensure that the same policy is deployed across the whole origin. Instead, we propose a different solution based on *non-transferable* Trusted Types, and prove the correctness of our approach within our model in Section 2.5.

2.4.3 Safe policy inheritance

The Content Security Policy specification [W3Ca, §7.8] mandates that every document loaded from a local scheme must inherit a copy of the policies of the source browsing

context, i.e., the browsing context that was responsible for starting the navigation. This corresponds to the following invariant.

Invariant I.7. *Documents loaded from a local scheme inherit the policy of the source browsing context.*

We encode the invariant in our model as follows:

```

1 Definition LSIInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ evs pt _evs frm fhtml fwd ctx lv pt_idx init_idx,
3     let get_csp wd :=
4       rp_hd_csp (dc_headers (wd_document wd)) in
5     Reachable gb evs st →
6     (* A document has just been loaded in a frame *)
7     evs = (EvDOMUpdate pt :: _evs) →
8     is_frame_in_dom_path gb (st_window st) pt frm fhtml fwd ctx →
9     is_local_scheme (wd_location fwd) →
10    (* get navigation initiator *)
11    pt = DOMPath lv (DOMIndex pt_idx) →
12    is_wd_initiator_of_idx ctx pt_idx (Some init_idx) →
13    (* The csp is equal to the req. initiator *)
14    get_csp fwd = get_csp (windows gb.[init_idx]).

```

When a frame has just loaded a document from a local scheme (lines 7-9), we require that the CSP of the navigation initiator (i.e., the source browsing context) is equal to the policy of the document loaded in the frame window (line 14).

The goal of this Web invariant is to ensure that a page cannot bypass its policy by navigating to content that is completely under its control. One such bypasses [Chrc] was caused by the behavior defined for the inheritance of policies in a previous version of the CSP specification [W3Ca, Oct. 15, 2018]: documents loaded from local schemes would inherit the policies of the embedding document or the opener browsing context.

Recently, the concept of *policy container* was added to the HTML specification [WHAb, §7.9]. A policy container is a collection of policies to be applied to a specific document and its purpose is to simplify the initialization and inheritance of policies. The introduction of the policy container in the specification allowed for clarifying the inheritance behavior for local schemes, which might differ depending on the specific scheme or URL that is used. The policy container explainer [Pol] stipulates the following behavior:

about:srcdoc An `iframe` element with the `srcdoc` attribute inherits the policies from the embedding document, i.e., the parent frame. Note that `srcdoc` iframes are in the same origin of the embedding document but their location URL is `about:srcdoc`.

about:, data: A document loaded from the `data:` or `about:` schemes inherits the policies of the navigator initiator (as mandated by the CSP specification).

blob: A document loaded from a `blob:` URL inherits the policies from the document that creates the URL, i.e., the document that calls the `URL.createObjectURL` function.

Note that in the current version of the HTML specification [WHAb, §7.11.1] the inheritance behavior for `blob:` URLs matches the one for `about:` and `data:`, thus following the CSP specification. We contacted the editors of the HTML specification [Our] asking for a clarification on the correct behavior for `blob:` URL and they confirmed that, because of the wrong ordering of a clause in the policy container construction for blobs, the initiator policy container was always replacing the creator policy container. The correct inheritance rule is to inherit the policy container of the creator of the URL [LSP], thus introducing an inconsistency between the CSP specification and the HTML specification (as `blob:` is a local scheme that is handled differently from the others).

Attack. When we configure our model to reflect a past state of the Web platform in which policies were inherited from the embedding frame and not from the navigation initiator (`c_csp_inherit_local_from_initiator (config gb) = false`), our toolchain is able to rediscover the attack trace that allows an attacker to strip the CSP policies by navigating a frame to a local scheme URL. The trace is shown in Figure 2.6: (steps 1-6) a document with no Content Security policy loads an iframe with a restrictive CSP; (7-9) the iframe contains a script which navigates (10) the frame itself (e.g., using the `window.location` setter) to a local scheme URL; (11-13) the iframe renders the content of the local scheme URL and inherits the CSP from the embedding document, which contains no policy. The resulting document has no CSP, effectively removing the iframe’s previous policy.

We can configure our model to reflect the current state of the Web platform by inheriting the policies from the navigation initiator for all local schemes:

```
c_csp_inherit_local_from_initiator (config gb) = true ^  
c_csp_inherit_blob_from_creator (config gb) = false.
```

We can verify (up to a finite size, see Section 2.6) that with this configuration the invariant holds. However, when we configure our model to reflect the planned modification of inheriting the policies of the URL creator when rendering a `blob:` URL (`c_csp_inherit_blob_from_creator (config gb) = true`), Z3 is able to find a new counterexample. The trace is similar to the one depicted in Figure 2.6 with an additional script loaded in the top-level window executing an `EvScriptCreateBlobUrl` event: (i) a page in `origin_1` with no CSP loads a same-origin iframe with a restrictive policy; (ii) a script running on the embedding document creates a new blob URL (`EvScriptCreateBlobUrl`); (iii) a script running on the inner frame navigates the frame itself (i.e., setting `window.location`) to the previously created URL. The frame loads the content of the blob and inherits the CSP from the embedding document, which does not have any policy. Similarly to the previous attack trace, the policy that was defined for the iframe has been removed by navigating to local-scheme content.

Hence, the planned modification on CSP inheritance in the HTML standard would introduce an inconsistency with the CSP specification. We responsibly reported the issue to the working group of the HTML standard [Our], who initially deemed the security

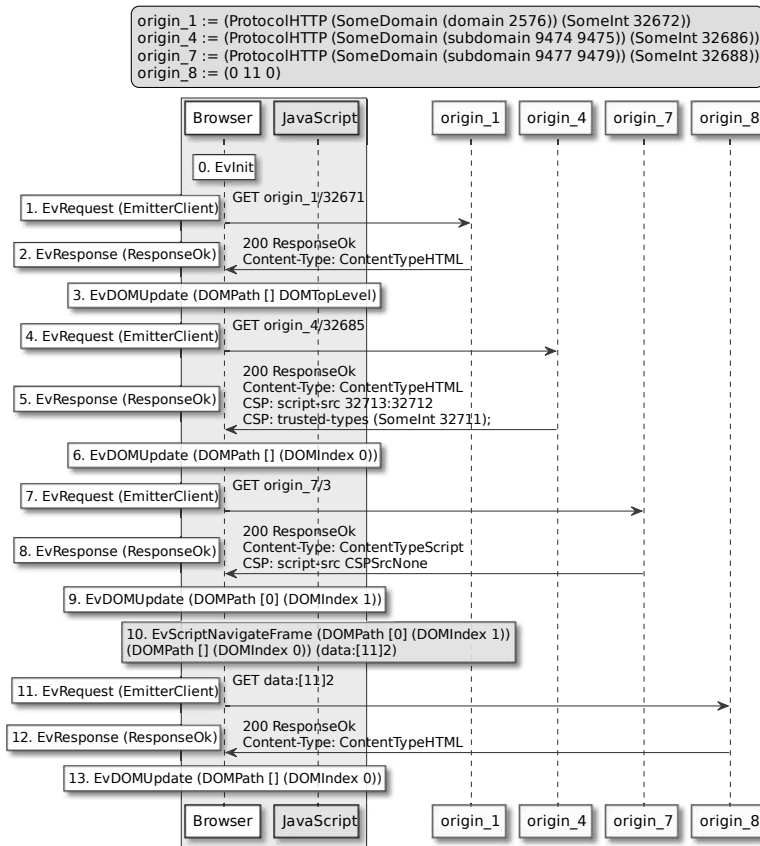


Figure 2.6: CSP bypass due to inheritance from the embedder document

implications of the attack as low. However, at the time of writing (August 2022), our invariant still holds since no final decision has been taken.

2.5 Verification of Security Properties

WebSpec supports both machine-checked and automated Web invariant proof generation. In particular, we wrote four machine-checked proofs in Coq and were able to automatically derive two further Web invariant proofs through the Spacer engine of μZ . We show here how to formally verify the security of the fix we present against the attack on Trusted Types (Section 2.4.2) through a machine-checked proof in WebSpec, and refer to Appendix A.2 and the online repository [Webb] for the remaining proofs.

According to the current draft of the Trusted Types specification, a Trusted Type object created by a page can be assigned to DOM XSS sinks belonging to different pages. This allows for bypassing the protection if a restricted document colludes with an unrestricted one. This can happen, e.g., in case of same-origin iframes (see Section 2.4.2). The specification acknowledges the issue and suggests the usage of the Origin Policy [DW] to

address the problem, which unfortunately is not currently supported by any browser.

For this reason, we propose an alternative solution that we label *non-transferable* Trusted Types, which consists in labeling each Trusted Type with the JavaScript realm (window or worker) that created it and ensuring that a type can only be assigned to DOM XSS sinks from the same realm. Therefore, our fix prevents the cross-document usage of Trusted Types. We implemented this behavior in WebSpec, which can be activated by setting the configuration option `c_tt_strict_realm_check` to `true`. The following theorem states the validity of the invariant `TTInvariant` (Section 2.4.2) when our fix is enabled:

```

1 Theorem strict_realm_check_implies_invariant :
2   ∀ gb evs st,
3     c_tt_strict_realm_check (config gb) = true →
4     c_restrict_tt_to_secure_contexts (config gb) = false →
5     TTInvariant gb evs st.
```

We recall that, according to the invariant, if a page is shipped with a CSP containing the directives `trusted-types` and `require-trusted-types-for 'script'`, then the list of events `evs` cannot contain a `EvScriptUpdateHTML` event that updates the contents of the page.

We can prove the theorem by induction on the `Reachable` relation where all the cases except `EvScriptUpdateHTML` are trivial. In this latter case we show that, by enabling strict realm checking, it is impossible to generate the correct Trusted Type for the update, since (i) the `trusted-types` directive disallows the creation of Trusted Types for the realm in which the directive is used; and (ii) the only Trusted Types that are accepted by a page with `require-trusted-types-for 'script'` are only those labelled with the realm of the page. This suffices to prove the correctness of the proposed solution within our model. The whole proof in Coq spans just over 54 LOC. For comparison, the longest of the four proofs we conducted in Coq is just 348 LOC, which demonstrates the feasibility of writing machine-checked proofs in our model.

2.6 Evaluation

In our experimental evaluation, we used WebSpec to automatically discover the attacks reported in Table 2.1. Additionally, when we implemented a fix to an attack, we ran WebSpec again to confirm that the issue had been addressed. Since the μZ solver may not terminate (see Section 2.3), we use the length of the previously discovered attack trace plus one as the maximal search size, thus obtaining confidence that the previous attack is not reachable anymore.

We report the time required by WebSpec to find the attacks and describe various optimization techniques that allowed us to drastically improve the performance of our approach. All our experiments have been conducted on a virtual machine with 32 VCPUs (2GHz AMD EPYC) and 128GB of RAM.

Table 2.2: Trace size and solving time for each attack.

| # | Query | Trace Size # Events | Solving Time | | | |
|----|---|------------------------|----------------|-----------|-----------------|-----------|
| | | | Frames Enabled | | Frames Disabled | |
| | | | Baseline | w/ Lemmas | Baseline | w/ Lemmas |
| 1 | Integrity of <code>__Host-</code> cookies | 15 | 58d 1h 30m | 23m * | × | × |
| 2 | Confidentiality of <code>HttpOnly</code> cookies | 7 | 13h 35m | 8m | 1h 46m | 1m |
| 3 | Interaction between SOP and CSP | 10 | 42d 3h | 46m * | × | × |
| 4 | Integrity of server-provided policies | 9 | 40h 35m | 18m | 15h 46m | 3m |
| 5 | Access control on Trusted Types sinks | 9 | 17h 50m | – | × | × |
| 6 | Access control on Trusted Types sinks (no sec. ctx) | 10 | 9d 20h 18m | – | × | × |
| 7 | Safe policy inheritance (inherit from parent) | 13 | 52d 10h | 1h 48m | × | × |
| 8 | Safe policy inheritance (inherit from creator) | 17 | – | 6h 5m | × | × |
| 9 | Authenticity of request Initiator | 5 | 1h 49m | – | 14m | – |
| 10 | Authorization of non-simple requests (i) | 5 | 35m | – | 5m | – |
| 11 | Authorization of non-simple requests (ii) | 10 | – | 48m | 35d 10h 51m | 7m |

×: N/A; – (Baseline): No solution could be found within 60 days;

– (w/Lemmas): None of our user-defined lemmas could be applied;

*: a lemma has been automatically extracted from the attack trace of a previous run of the solver.

The baseline performance is displayed on the third column of Table 2.2. We can observe a clear correlation between the size of the attack trace and the time required to find an attack, which is caused by the unrolling technique employed by the BMC engine of the μZ solver used in WebSpec. In particular, time increases exponentially with respect to the size of the attack trace, leading to running times of several days or weeks for traces with 10 or more events. To tackle these performance issues, we have implemented various optimizations that consist in (i) defining additional rules (or lemmas) representing common configurations (e.g., loading of a frame containing a script) that can be used by the SMT solver instead letting it rediscover the right list of events leading to these configurations, and (ii) simplifying the model at compile time (e.g., by disabling frames) so that the resulting SMT formula is easier to solve.

We describe these optimizations in the following, and we refer the reader to Appendix A.4 for a discussion on the scalability of our browser model, which confirms the effectiveness of lemmas in mitigating the complexity that arises from the addition of new Web components.

2.6.1 User-defined Lemmas

The key idea underlying this optimization is to enable users to define additional lemmas that guide the solver into constructing interesting browser states that can be used as a starting point to discover new attacks. Intuitively, lemmas represent encodings of common Web Security threat models, which map to the general preconditions of specific classes of Web attacks. Consider the following example:

```

1 Lemma script_state_is_reachable : ∀ gb,
2   script_state_constraints gb →
3   Reachable gb (script_state_events gb) script_state.

```

Here `script_state` is a Coq definition of a concrete browser state where a script is loaded in the page rendered in the top-level window. This state maps to the threat model in which an attacker controls a script running in the same page as the target Web application, as a result of, e.g., XSS or the inclusion of untrusted scripts [NIK⁺12].

The lemma encodes that this browser state is reachable by applying the list of events `script_state_events` assuming that `script_state_constraints` is satisfied. Once we prove that the state defined by a lemma is `Reachable`, the lemma can be compiled as an additional model rule to CHC logic. Since the BMC engine solves queries by iterative unrolling, it prioritizes the rules that result in the smallest amount of unrolling steps. Lemmas exploit this property by providing a one-step solution for the generation of states that would require multiple steps if the solver had to build them from scratch.

WebSpec includes the definition of five generic lemmas, which represent two variants of the aforementioned attacker controlled script, one running in a secure context and the second being served from an insecure connection, and three variants of the *gadget attacker* of [BJM08b] in which a same-origin, same-site or cross-site iframe containing an attacker controlled script is included into the target Web applications. When compiling the model to an SMT formula, the `Using Lemmas` directive is used to specify which user-defined lemmas need to be considered for the query. For our experiments, we provided every query with all the lemmas that are included in WebSpec, since we experimentally observed that the number of lemmas added to a query does not negatively impact solving time. This behavior may result from the effectiveness of μZ in discarding all non applicable initial states. For this reason, we expect only an improvement of the solver performance from the definition of a larger library of lemmas.

The results of this optimization are highlighted in Table 2.2, where we can see that the usage of lemmas always reduces the runtime to less than a day. It may however happen that the solver is not able to apply any of the user-defined lemmas, as it is the case for queries #5 and #6 (marked with $-$). In such cases no performance improvement can be obtained. For the queries marked with \star , we automatically extracted a lemma from an attack trace discovered by WebSpec and confirmed that it can be used by successive runs of the solver. The extraction of lemmas from traces has several benefits: on the one hand, it allows us to quickly test for the absence of a known attack after applying a fix to the model; on the other hand, it allows us to add new reachable browser configurations to our library of lemmas. Since these configurations represent the preconditions for the execution of a specific attack, the extraction of additional lemmas augments our counter example pipeline with a method to quickly discover novel attacks assuming known preconditions. We leave the extension of this library and the definition of a methodology to generate generic lemmas as future work.

2.6.2 Compile-Time Simplification

Configurable inlining of auxiliary relations. Our model relies on a `Reachable` relation that models state transitions, a `ScriptState` relation that models scripts knowledge, and multiple auxiliary relations that are used within `Reachable` to, e.g., recursively update the DOM. The presence of multiple relations prevents us to directly use the best performing version of the BMC engine, the linear solver, because it requires the model to be encoded as linear Horn clauses, i.e., clauses containing at most one recursive term. In order to satisfy this requirement, every auxiliary relation needs to

be inlined within the main `Reachable` relation. To this end, WebSpec automatically unrolls all the applications of recursive relations that are marked for inlining. For each relation we specify the depth of the unrolling. For instance, the declaration

```
Inline Relation is_script_in_dom_path With Depth 3.
```

says that the relation `is_script_in_dom_path`, which searches a script inside the DOM, must be unrolled up to recursion level 3. Depth 0 disables all recursive calls and expands the relation to the base case only. For instance, support for nested frames can be easily deactivated by specifying 0 as depth for all the relations handling the DOM tree.

The recursion depth affects the solving time of μZ since multiple applications of the relation need to be considered. Disabling nested frames for the queries which do not require them simplifies the compiled model and allows for faster solving. When frames are required, we set the `Depth` parameter so that a single level of nesting is allowed. Although our model can handle an arbitrary number of nested frames, a single level suffices to discover the minimum-size trace for all queries.

The effects of this optimization are shown in Table 2.2: we can see that disabling framing for the queries that do not require multilevel DOM trees considerably lowers the solving time.

Fixed size arrays. Our model uses functional arrays [McC62, Rey79] for the definition of the HTML and DOM objects and the implementation of the window/frame tree. However, functional arrays are known for significantly increasing the complexity of queries [SJ80, SBDL01]. Therefore, in order to ease the resolution, our compiler provides an optimization which turns functional arrays into arrays of a fixed size chosen at compile time. Since choosing a small size could make a query unsolvable, we run in parallel several instances of the same query with different sizes and keep the first one that succeeds. Surprisingly, a size of 5 is enough for all the queries except those for *Safe policy inheritance* (#7 and #8 in Table 2.2) which require a size of 7.

2.7 Related Work

Models of the Browser. Bohannon [Boh12] proposed Featherweight Firefox, a model of a Web browser written in Coq for the verification of security properties concerning JavaScript execution. The model supports several JavaScript features, such as DOM manipulation, XHR requests, event listeners, and code evaluation via the `eval` function. However, the set of modeled Web components comprises only windows, cookies, and selected HTML tags (`<script>`, `<div>`). Bugliesi et al. [BCFK15] extended Featherweight Firefox to formalize the security guarantees of `HttpOnly` and `Secure` cookies against network and Web attackers able to exploit XSS vulnerabilities. In [BCF⁺14] the authors use Featherweight Firefox as a starting point to develop a pen-and-paper model of a security-enhanced browser which enforces a Web session integrity property that captures attacks like CSRF and credential theft via XSS.

In contrast to WebSpec, Bohannon’s model and later extensions were developed with machine-checked proofs in mind and have not been used to automatically detect vulnerabilities. They also lack support for most of the Web features considered in our invariants, e.g., CORS, CSP, service workers. Because of their focus on Web script security, they formalize script executions using a small-step semantics. This choice allows for a precise modeling of JavaScript but hinders automatic verification, as it forces solvers to handle JavaScript programs.

Models of the Web. In their seminal work, Akhawe et al. [ABL⁺10] developed the first formal model of the Web ecosystem. The authors encoded in the model a set of security goals, which include fundamental properties of the Web platform that are assumed to hold, and a notion of session integrity capturing CSRF attacks. The validity of these goals has been checked with the Alloy Analyzer [Jac02] and their violations pointed out the existence of novel and previously known attacks.

Despite being similar in spirit to our proposal, there are important differences between WebSpec and the model of Akhawe et al. First, the model cannot be used to prove security properties, since the Alloy Analyzer uses SAT-based bounded model checking, but just to disprove them, while WebSpec can be used also to produce automated or machine-checked proofs. Second, being developed in 2010, it lacks many features of the modern Web (e.g., frames, CSP and service workers) that are a fundamental part of our model. Adding these features a posteriori would not be possible without rewriting the model from scratch, since some of them (e.g., a faithful handling of frames) are core components of Web browsers. Last, contrary to WebSpec, the model of Akhawe et al. is stateless. For this reason, temporal relations between events, e.g., the correct sequencing of requests and response pairs, need to be explicitly modeled. Considering that Web Standards are typically written using a stateful imperative style, a more natural modeling follows a stateful approach, as employed in our model.

Bansal et al. [BBDM14] developed WebSpi, a generic library that defines the basic components of the Web infrastructure (browsers, HTTP servers) and enables developers to automatically verify security properties of specific Web applications / protocols using ProVerif [Bla01]. The browser model of WebSpi is rather primitive and includes a subset of the features supported in WebSpec. This is in line with the intended usage of WebSpi, i.e., the verification of Web protocols, for which it suffices to model only the features used by the protocol under analysis. Instead, we target inconsistencies between Web features themselves, without focusing on a specific Web protocol or application, for which we need a much more comprehensive browser model. Similarly to WebSpi, WebSpec supports automated security proofs: if this does not succeed, however, we can still fall back to machine-checked proofs in Coq.

The most comprehensive and maintained model of the Web to date is the *Web Infrastructure Model* (WIM), a pen-and-paper model which has been used to assess the security of Web Payment APIs [DHK⁺22], Web protocols, e.g., OAuth 2.0 [FKS16], OpenID Connect [FKS17], and the Financial-Grade APIs [FHK19]. The browser model

of WebSpec supports most of the features of WIM browsers, except for (i) HSTS, since in our model we abstract away the network, (ii) HTTP basic authentication, because it is an application-specific server-side mechanism, (iii) the Web Payment APIs, since a sensitive usage of this API would require a detailed modeling of the server-side behavior of payment providers and merchants servers. On the other hand, WebSpec supports several client-side mechanisms and security policies, like domain relaxation, CSP and CORS, that are not part of WIM. Additionally, being a pen-and-paper model, WIM can neither be used to automatically discover security vulnerabilities, nor to develop automated or computer-assisted proofs, which are central features of our framework.

Other works. Quark [JTL12] is a WebKit-based Web browser, whose kernel has been implemented and formally verified in Coq. The kernel is responsible for handling input/output and offers services to the other possibly compromised components of the browser, which deal with various operations such as the rendering of Web pages, handling of cookies and tab management. The separation of duties between the kernel and browser’s components, together with a set of security policies implemented in the kernel, enables Quark developers to formally prove the enforcement of security properties, such as tab non-interference, and integrity of cookies and responses. This is orthogonal to our work, which instead aims at devising a formal browser model to validate Web invariants.

Automated testing is a popular methodology employed in software development processes for bug detection. An application of this methodology in the context of Web security is BrowserAudit [HMN15], a framework composed of over 400 automated tests, which can be used to verify the correctness of the implementation of Web security mechanisms in existing browsers. However, BrowserAudit cannot be used to spot bugs at the specification level, which is the goal of our work.

QuickChick [PHD⁺15, LPP18] is a framework for property-based testing written in Coq. It combines formal methods and testing to formally verify that the code of a test generated from a given property is indeed checking its correctness. QuickChick is orthogonal to our work since it focuses on test case generation: WebSpec relies on testing solely to prove the validity of the counterexamples to our invariants produced by the Z3 solver.

Summary. To conclude, WebSpec is the first framework that mechanizes formal proofs and counterexample-finding for Web invariants. In addition, our browser model is the most comprehensive one when it comes to browser-side security mechanisms, as we further detail in Appendix A.6.

2.8 Conclusion

In this chapter we presented WebSpec, the first formal framework for the security analysis of Web platform components that supports the automated detection of logical flaws and allows for the development of machine-checked security proofs. We showcased the effectiveness of WebSpec by discovering novel attacks and inconsistencies affecting current Web standards, and automatically validated our findings against major browsers.

2. WEBSPEC: TOWARDS MACHINE-CHECKED ANALYSIS OF BROWSER SECURITY MECHANISMS

Additionally, we discussed how WebSpec can be used to carry out machine-checked security proofs for vulnerability fixes. As a future work, besides expanding the model to cover more Web platform components, we plan to define additional Web invariants by reviewing newly proposed mechanisms and engaging with the community, including developers and editors of Web standards.

Web Platform Threats: Automated Detection of Web Security Issues With WPT

Abstract

Client-side security mechanisms implemented by Web browsers, such as cookie security attributes and the Mixed Content policy, are of paramount importance to protect Web applications. Unfortunately, the design and implementation of such mechanisms are complicated and error-prone, potentially exposing Web applications to security vulnerabilities. In this paper, we present a practical framework to formally and automatically detect security flaws in client-side security mechanisms. In particular, we leverage Web Platform Tests (WPT), a popular cross-browser test suite, to automatically collect browser execution traces and match them against Web invariants, i.e., intended security properties of Web mechanisms expressed in first-order logic. We demonstrate the effectiveness of our approach by validating 9 invariants against the WPT test suite, discovering violations with clear security implications in 104 tests for Firefox, Chromium and Safari. We disclosed the root causes of these violations to browser vendors and standard bodies, which resulted in 8 individual reports and one CVE on Safari.

The work presented in this chapter is the result of a collaboration with Pedro Bernardo, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão and Matteo Maffei, and has been published in the 33rd USENIX Security Symposium in 2024 under the title “Web Platform Threats: Automated Detection of Web Security Issues With WPT” [BVV⁺24]. Pedro Bernardo and I contributed equally to this work and are considered to be co-first authors. I am responsible for the formalization of the Web invariants about cookies, our first-order logic model of browser execution traces and the SMT-LIB translator

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

that generates queries for the theorem prover. I developed the kubernetes-based testing pipeline and performed the experimental evaluation. Pedro Bernardo is responsible for the browser instrumentation, he disclosed the cookies vulnerabilities to the affected parties and contributed to the experiments and the formalization of the invariants. Valentino Dalla Valle formalized the Mixed Content specification in the form of two Web invariants and reported the Mixed Content vulnerabilities. Stefano Calzavara, Marco Squarcina, Pedro Adão and Matteo Maffei were the general advisors and contributed with continuous feedback.

3.1 Introduction

Writing secure Web applications is notoriously hard, due to the heterogeneity, complexity and open-ended nature of the Web. To mitigate the challenges of secure Web application development, browsers integrate a growing list of client-side security mechanisms to assist Web developers. Examples of such mechanisms include cookie security attributes (HttpOnly, Secure and SameSite), security headers like Origin and Sec-Fetch-Data, mechanisms to secure mixed content (e.g., to avoid that HTTPS-served webpages fetch content in clear over HTTP), and sophisticated client-side protection mechanisms like Content Security Policy (CSP).

The design of such mechanisms is very delicate, as witnessed by the long list of design shortfalls (e.g., unexpected interactions with other browser components) or implementation flaws, which led to breaking well-established Web security invariants [ABL⁺10, VFB⁺23]. *Formal methods proved to be an essential tool to rigorously analyze client-side security mechanisms*, allowing for the identification of bugs and formulation of formal security proofs in such a complex environment. All state-of-the-art techniques, however, be they manual [FKS16], machine-checked [BP10], or automated [VFB⁺23, ABL⁺10], apply to *browser models*, which suffer from two fundamental drawbacks. First, client-side security mechanisms evolve over time and new ones are being proposed on a regular basis, which makes browser models extremely hard to maintain. Second, even if specifications are correct, security-critical bugs often affect the implementations [KKH⁺22, SNM17, SMWL10b, WNK⁺23]. Correctly integrating client-side security mechanisms within browsers is challenging and error-prone for various reasons. Browsers are incredibly complicated software artifacts: for instance, the Chromium codebase contains roughly 35 million lines of code, i.e., it is larger than the Linux kernel. Furthermore, browser vendors are required to translate natural language specifications, e.g., from the World Wide Web Consortium (W3C), into new code to be pushed into an already complicated codebase. Even worse, client-side security mechanisms often cannot be specified in isolation: most of them interact with core browser components like Fetch, which defines requests, responses, and the process which eventually binds them. This means that the implementation of client-side security mechanisms often requires changes to existing browser components which were not developed with such an integration in mind.

We thus tackle the following research question: *can we design a practical framework to formally and automatically detect security flaws in the implementation of client-side security mechanisms?*

In this chapter, we answer in the affirmative, putting forward a novel, formally-grounded and lightweight technique. In particular, we leverage existing community efforts in the development of Web Platform Tests (WPT) [WPTb], a cross-browser test suite designed to give browser vendors confidence that they are shipping software which is compliant with specifications and compatible with other implementations. WPT includes more than 50K tests covering a wide range of browser components, including Web security mechanisms, thus representing the largest benchmark of the intended browser behavior to date. Our approach consists in abstracting the test executions into sets of *traces* (i.e., sequences of relevant browser events), which are then matched against *Web security invariants* (i.e., intended security properties expressed in first-order logic). This way, we automatically identify traces breaking important security properties and thus pinpoint browser behaviors requiring immediate attention by browser vendors, due to their clear security implications. Furthermore, WPT is continuously updated as Web standards and new features are introduced to the Web platform, which makes our verification pipeline automatically applicable to the latest browser versions.

Contributions. More concretely, we contribute as follows:

- We formalize 9 Web invariants regarding core components of the Web platform such as Cookies and Mixed Content, encoding them in first-order logic to allow for efficient verification of browser execution traces using an automated theorem prover (Section 3.3).
- We present an automated pipeline designed to identify security-critical inconsistencies in browser implementations. Our approach leverages the WPT test suite to acquire browser execution traces, which are then matched against Web security invariants in order to identify any traces that violate Web security properties (Section 3.4).
- We demonstrate the effectiveness of our approach by validating our 9 invariants against the WPT test suite, discovering violations with clear security implications in 104 tests (Section 3.5). In particular, we discuss 10 attacks against Chromium, Firefox, and Safari concerning cookies and Mixed Content policy violations (Section 3.5.2). We responsibly disclosed all the new findings to affected browser vendors and standard bodies, which resulted in 8 individual reports and one CVE on Safari.

We publish all the artifacts developed during this research, including the definition of the Web invariants in SMT-LIB format, and our trace verification pipeline [BVV⁺23].

3.2 Background

We assume familiarity with the basic functionality of the Web platform, e.g., the HTTP protocol, HTML and JavaScript.

Web Security Primer. The traditional threat model of Web applications considers both *Web attackers* and *network attackers* [CFST17]. A Web attacker is the owner of a malicious host, which is used to mount attacks against other Web applications. Traditional examples of Web attacks include Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF). A network attacker extends the capabilities of a Web attacker with full control of the network traffic, i.e., everything which is unencrypted can be read and modified by a network attacker. Encryption can be enforced through the use of HTTPS, which provides a secure transport protocol for the Web. Browsers rely on the notion of *secure context* to identify pages satisfying minimal confidentiality and integrity requirements [W3C21].

The baseline defense mechanism of Web browsers is the *Same Origin Policy* (SOP), which is intended to enforce the intuitive invariant that content owned by a Web application should not be read or written by other Web applications. The notion of *origin* defines the security perimeter of SOP: an origin is a triple including a scheme (HTTP, HTTPS...), a host (e.g., `www.foo.com`) and a port (defaulting to 80 for HTTP and 443 for HTTPS). This way, a Web page at `https://evil.com` cannot access content served by `https://foo.com`. Since the fine-grained isolation of SOP is too restrictive for specific settings, another common Web security concept is the notion of *site*, i.e., one domain part plus the *effective top-level domain* as defined in the Public Suffix List [Moza] – also called *registrable domain* or *eTLD+1*. For example, `foo.com` and `foo.github.io` (as `github.io` is in the PSL) are sites, and `a.foo.com` and `b.foo.com` are two subdomains of the same site `foo.com`. Although `https://a.foo.com` and `https://b.foo.com` are two different origins, their same-site position might relax some security checks enforced by browsers (see below). The W3C Secure Contexts specification [W3C21] also defines the notion of *potentially trustworthy* origins as those that the browser can trust sending data securely. In particular, in addition to origins whose protocol is `https` or `wss`, the `localhost` IP address and all subdomains of `localhost` are considered potentially trustworthy even for unencrypted connections.

Cookies. Cookies are a client-side storage mechanism based on the name-value paradigm and can be set through JavaScript or using the `Set-Cookie` header of HTTP responses. In their default configuration, cookies are accessible by JavaScript using the `document.cookie` property and are attached by the client to all the requests sent to the host which set them, using the `Cookie` header. The scope of cookies can be extended to other subdomains by using the `Domain` attribute; this allows cookie sharing across sibling domains, e.g., `a.foo.com` can set cookies with the `Domain` attribute set to `foo.com`, which makes them available to `b.foo.com`. Since cookies may store sensitive data, e.g., session identifiers that must be protected to prevent session hijacking, clients offer a plethora of defensive options deployed in terms of cookie *attributes* and *prefixes*.

Cookies marked with the `Secure` attribute are only attached to requests sent over secure channels, e.g., over HTTPS, which is important to ensure their confidentiality against network attackers. The `HttpOnly` attribute makes cookies inaccessible to JavaScript, which is useful to prevent cookie theft in the presence of injection vulnerabilities like XSS. Finally, the `SameSite` attribute can be used to restrict the attachment of cookies to same-site requests, thus mitigating CSRF. If `SameSite` is set to `Strict`, no cross-site request will ever attach the cookie; if `SameSite` is set to `Lax`, top-level navigation requests with a safe method (e.g., GET) can attach the cookie even though they are fired from a cross-site position.

Since cookies have weak integrity guarantees in their default configuration, Web developers can qualify their names with special prefixes to improve protection. The `__Secure-` prefix requires the cookie to be set over secure channels with the `Secure` attribute activated. The `__Host-` prefix extends the protection of the `__Secure-` prefix by also forcing the deactivation of the `Domain` attribute, thus scoping the cookie to a specific host rather than to its site.

Mixed Content. When a document is loaded via a secure channel, all its subresources, i.e., frames, scripts, etc, must also be received securely to not compromise the integrity of the page. If any of such resources is loaded via a non-secure channel, i.e., HTTP, a network attacker can tamper with the content of the reply, opening the possibility for, e.g., executing malicious JavaScript code within a secure context.

The W3C Mixed Content specification [W3C23] regulates the fetching of subresources within documents loaded via a secure channel, defining as *mixed content* any insecurely-loaded subresource. Mixed content is categorized based on the corresponding security risks. Mixed content is *upgradeable* when the risk of allowing its usage is outweighed by the risk of breaking significant portions of the Web. Image, audio, and video content are all classified as upgradeable because the usage of such resource types is sufficiently high, while their loading is generally considered as low-risk. Upgradeable mixed content goes through protocol autoupgrading: the URL is rewritten to use the HTTPS protocol and an attempt is made to fetch the subresource securely. If the resource is not available via the new URL, it will not be loaded in the page.

Any mixed content that is not upgradeable is classified as *blockable*. Examples of blockable content are scripts, frames, XHR, and fetch requests. The risk of loading such content is much higher: for example, allowing insecurely-loaded scripts within a secure context would allow a network attacker to read or modify data accessed therein. Blockable mixed content is filtered and the subresource is not loaded in the document.

3.3 Web Invariants

A *Web invariant* is an intended security property of a Web security mechanism that should never be violated by Web browsers, i.e., any counter-example might reveal a security-relevant bug. In this chapter, we define 9 Web invariants concerning two core

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

| | Name Invariant | Description | References |
|---------------|--|---|-----------------------|
| Cookies | I.1 Integrity of Secure cookies | <i>Cookies with the Secure attribute can only be set over secure channels.</i> | [VFB ⁺ 23] |
| | I.2 Confidentiality of HttpOnly cookies | <i>Scripts can only access cookies without the HttpOnly attribute.</i> | [VFB ⁺ 23] |
| | I.3 Integrity of <code>__Host-</code> cookies | <i>A <code>__Host-</code> cookie set for domain d can only be set by d or by scripts included in pages on d.</i> | [VFB ⁺ 23] |
| | I.4 Integrity of SameSite cookies | <i>A <code>SameSite=Lax/Strict</code> cookie can only be set for domain d through HTTP responses to requests initiated by domains which are same-site with d or by top-level navigations.</i> | [CEWW22, §4.1.2.7] |
| | I.5 Isolation of SameSite cookies | <i>If a <code>SameSite=Lax/Strict</code> cookie should not be attached to a request to load a page p, then it is not attached to that request, it is not accessible by scripts in p nor attached to requests initiated by p.</i> | [sam] |
| | I.6 Cookie serialization collision resistance | <i>A cookie with name n and value v is serialized to the string "$n=v$" when attached to requests or accessed via <code>document.cookie</code>.</i> | [SAVM23] |
| | I.7 Confidentiality of Secure cookies | <i>Secure cookies are only attached to requests (resp. accessible by scripts) to potentially trustworthy URLs.</i> | [pot] |
| Mixed Content | I.8 Blockable mixed content filtering | <i>Every request performed by the browser is either a toplevel request, its URL is potentially trustworthy, or the request context does not prohibit mixed content.</i> | [W3C23, §4.4] |
| | I.9 Upgradeable mixed content filtering | <i>For every non-toplevel request performed by the browser where the URL is not potentially trustworthy, the request context does not prohibit mixed content and the request type is not upgradeable.</i> | [W3C23, §4.1] |

Table 3.1: Web Invariants

components of the Web Platform: cookies and Mixed Content. The selection and definition of these invariants is based on the following methodology. First, we focus on Web components with clear security implications and relatively compact specifications. For each selected mechanism, we abstract the expected security properties by thoroughly analyzing the specification. We then review the existing literature to identify invariants already defined in prior research. In cases where specifications prove to be ambiguous, we encode as a Web invariant the community security expectations that emerge from previous research or from our discussion with the specification maintainers. For each of these cases, we provide a bibliographic reference or a link to the GitHub discussion. Finally, we express the invariants as first-order logic formulas. Table 3.1 presents an intuitive natural language description of the invariants we encode in this work. In particular, we define 6 new Web invariants (I.4–I.9) and propose an encoding of 3 invariants from the literature (I.1–I.3). In this section, we focus on the 6 new Web invariants we propose, presenting their expected security property and encoding. We first define a model to represent browser execution traces and show how security properties can be encoded in

this model. We then proceed with the discussion of the invariants. The encoding of the remaining invariants is discussed in Appendix B.1.

3.3.1 Traces and Events

We define Web invariants in terms of browser execution traces. A *trace* is represented as a list of browser events, each mapping to a concrete browser action. Events are encoded as shown in Figure 3.1 and capture JavaScript API calls (*js*), network requests and responses (*net*), and hooks into the browser internals, e.g., *cookie-jar-set* triggers when a cookie is stored in the cookie jar. JavaScript events store a reference to the browsing context, i.e., the Window or Worker, in which the API call was executed. For each browsing context, we store a unique identifier, its location URL, and a flag indicating whether it is a secure context [W3C21] or not.

Invariants are encoded as first-order logic formulas, which should be true for all possible traces.¹ As an example, consider our encoding of the *Confidentiality of HttpOnly Cookies* (I.2) defined in [VFB⁺23].

$$\begin{aligned} \text{HTTP-ONLY-INVARIANT}(tr) := & \\ & t_1 > t_0 \wedge \\ & \text{cookie-jar-set}(name, value, \{http\text{-only}, secure, domain, path\})@_{tr}t_0 \wedge \\ & \text{js-get-cookie}(ctx, cookies)@_{tr}t_1 \wedge \\ & name \# "=" \# value \in \text{split-cookie}(cookies) \wedge \\ & \text{cookie-match}(path, domain, secure, ctx\text{-location}(ctx)) \rightarrow \\ & \quad http\text{-only} = false \end{aligned}$$

The invariant is defined as an implication, requiring the *http-only* flag to be equal to false if a set of hypotheses is satisfied. We use the $e@_{tr}t$ predicate to check if event e is present in trace tr at timestamp $t \in \mathbb{N}$. Intuitively, this invariant says that if a script successfully uses the `document.cookie` getter (*js-get-cookie* at time t_1) to obtain the *cookies* string, and if *cookies*, after splitting on the cookie separator ";", contains the string composed of the concatenation of *name*, the literal string "=", and *value*, then the *http-only* flag present when the cookie was set (*cookie-jar-set* at time $t_0 < t_1$) needs to be set to *false*. We use the *split-cookie* function to split a cookie header on the separator character ";", returning a list, and the *cookie-match* predicate to consider the case in which the cookie set at time t_0 is readable by the browsing context *ctx* where `document.cookie` is accessed. In particular, given a URL and the path, domain and security attributes of a cookie, *cookie-match* is true when the domain matching and path matching algorithms defined in the specification [CEWW22] return true and when, if the Secure attribute is set, the URL uses a secure protocol. That is, when *cookie-match* is true for a URL and a cookie, we should expect that cookie to appear in the request headers and `document.cookie` for that URL.

Invariants are expressed in quantified first-order logic using the theories of uninterpreted functions, integer arithmetic, algebraic datatypes, and strings. In particular, events

¹For readability, all variables are implicitly \forall -quantified when no quantification is specified.

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

| | |
|--|--|
| $Trace := List\ Event$ | execution trace |
| $Ctx := \langle id, location, secure-context \rangle$ | browsing context |
| $Event :=$ | browser event |
| $js-set-cookie(Ctx, arg, ret)$ | document.cookie setter |
| $js-get-cookie(Ctx, ret)$ | document.cookie getter |
| $cookie-jar-set(name, value,$ $attributes, deleted)$ | cookiejar hook on set/delete cookie |
| $net-request(id, url, method, type,$ $origin, doc-url,$ $frame-ancestors,$ $headers, body)$ | network request |
| $net-response(id, url, headers, body)$ | network response |
| $js-fetch(Ctx, url)$ | window.fetch API call |

Figure 3.1: Syntax of traces: event types.

are defined as a datatype, the $@_{tr}$ predicate is implemented as a recursive function, and auxiliary predicates can be defined as macros or functions. This combination of theories gives us flexibility in the definition of Web invariants, e.g., allowing us to encode properties about parsing and serialization, while allowing for automated verification using the Z3 theorem prover.

Integrity of SameSite Cookies

The cookie specification explicitly forbids setting SameSite cookies (either `Lax` or `Strict`) in response to non-top-level cross-site requests [CEWW22, §4.1.2.7]. For instance, assume that `https://good.com` embeds a page at `https://evil.com` as an `iframe`. If the `iframe` includes subresources from `https://good.com`, the browser should discard SameSite cookies set in responses to those requests. This behavior defines additional integrity guarantees to SameSite cookies and corresponds to the following invariant.

Invariant (I.4). *A cookie whose SameSite attribute has value Strict or Lax can only be set for domain d through HTTP responses to requests initiated by domains which are same-site with d or by top-level navigations.*

We encode this invariant as follows:

```

SAMESITE-COOKIES-INTEGRITY( $tr$ ) :=
 $t_1 < t_2 < t_3 \wedge$ 
 $net-request(id, url, \_, type, origin-url, \_, \_, \_, \_)@_{tr}t_1 \wedge$ 
 $net-response(id, url, \{set-cookie-headers\}, \_)@_{tr}t_2 \wedge$ 
 $set-cookie \in set-cookie-headers \wedge$ 
 $name \neq "" \wedge value \in split-cookie(set-cookie) \wedge$ 
 $"SameSite=" \wedge SS \in split-cookie(set-cookie) \wedge$ 
 $(SS = "Lax" \wedge same-site = SS-Lax \vee$ 
 $SS = "Strict" \wedge same-site = SS-Strict) \wedge$ 
 $cookie-jar-set(name, value, \{same-site, path, domain\})@_{tr}t_3 \wedge$ 
 $cookie-match(path, domain, \_, url) \wedge$ 
 $url-site(url, site) \rightarrow$ 
 $(type = main\_frame \vee url-site(origin-url, site))$ 

```


For every *net-response* event that successfully sets a cookie, i.e., that is followed by a *cookie-jar-set* whose parameters match the value of the response `Set-Cookie` header; if the `SameSite` attribute is set to `Lax` or `Strict`, then either the request type is *main_frame*, i.e., it is a top-level request, or the initiator of the request is same-site w.r.t the target url of the request, i.e., *origin-url*, the url of the request initiator, is in the same site of *url*. Here, the *url-site* predicate is true when its second argument is the site of the url in the first argument.

Isolation of SameSite Cookies

SameSite cookies, especially when set with the `Strict` attribute, are widely considered a robust defense against cross-site attacks such as CSRF [CEWW22] and, more recently, XS-Leaks [SSO, RPS23, SKC20]. The protection is effective as long as these cookies are not attached to requests initiated by an attacker operating from a cross-site page. For instance, the specification mandates browsers to not include SameSite cookies in requests to load cross-site iframes, nor make them available to JavaScript APIs in that context [CEWW22, §5.2.1].

We verified instead that cross-site top-level navigations can cause same-site navigations to be executed, thus attaching SameSite cookies to requests initially started by the attacker. This is the case of a pop-up window opened by a cross-site page, which executes a same-site JavaScript-based redirection via, e.g., `window.location`. Browsers consider the first request as cross-site but the second as same-site, thus attaching SameSite cookies to the second request, as captured by the specification [CEWW22, §8.8.5]. Similarly, subresources loaded in a top-level cross-site context are considered same-site and are loaded with SameSite cookies attached.

By carefully examining public discussions between browser vendors [cooa, coob, cooc], we found that the current behavior is the result of a bottom-up threat modeling process, with security implications that extend beyond what is declared in the specification: “*same-site navigations and submissions can certainly be executed in conjunction with other attack vectors such as cross-site scripting*”. Indeed, SameSite `Strict` cookies can be bypassed using JavaScript-based same-site redirectors (i.e., no XSS required) [Por], and loading authenticated subresources can introduce observable user-dependent state in the opened page, thus enabling XS-Leaks attacks, as we discuss in Section 3.5.2. We are currently engaging with browser vendors and specification maintainers to harmonize the specification and the implementations, and to clarify the security properties that should be expected from SameSite cookies based on the principle that high-sensitive resources (e.g., cookies and authenticated resources) should not flow into low-sensitive contexts (e.g., pages loaded from cross-site requests) [sam].

Invariant (I.5). *If a cookie set for domain d with the SameSite attribute set to "Lax" or "Strict" should not be attached to a request that loads a page p , then the cookie is not attached to that request, it is not accessible to scripts running in p and it is not attached to network requests initiated by p .*

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

We encode the invariant as:

$$\begin{aligned} \text{SAMESITE-COOKIES-CONFIDENTIALITY}(tr) := & \\ & t_1 < t_2 < t_3 \wedge \\ & \text{cookie-jar-set}(\text{name}, \text{value}, \{\text{secure}, \text{same-site}, \text{path}, \text{domain}, \text{host-only}\})@_{tr t_1} \wedge \\ & (\text{same-site} = \text{SS-Lax} \vee \text{same-site} = \text{SS-Strict}) \wedge \\ & \text{net-request}(_, \text{url}, \text{method}, \text{type}, \text{origin}, _, _, \text{redirs}, \{\text{cookies}\}, _)@_{tr t_2} \wedge \\ & \text{cookie-match}(\text{path}, \text{domain}, \text{secure}, \text{host-only}, \text{url}) \wedge \\ & \neg \text{cookie-match-samesite}(\text{same-site}, \text{type}, \text{origin}, \text{method}, \text{redirs}, \text{url}) \wedge \\ & (\\ & \quad (\text{js-get-cookie}(\text{ctx}, \text{cookies}')@_{tr t_3} \wedge \text{url} = \text{ctx-location}(\text{ctx})) \vee \\ & \quad (\text{net-request}(_, \text{url}', \text{method}', \text{type}', \text{origin}', \text{doc-url}', _, \text{redirs}', \{\text{cookies}'\}, _)@_{tr t_3} \wedge \\ & \quad \quad \text{doc-url}' = \text{some}(\text{url}) \wedge \\ & \quad \quad \text{cookie-should-be-sent}(\text{path}, \text{domain}, \text{secure}, \text{same-site}, \text{host-only}, \text{type}', \text{origin}', \text{url}', \text{method}', \text{redirs}')) \\ &) \rightarrow \\ & (\text{name} \neq "" \wedge \text{value} \notin \text{split-cookie}(\text{cookies}) \wedge \\ & \quad \text{name} \neq "" \wedge \text{value} \notin \text{split-cookie}(\text{cookies}')) \end{aligned}$$

Assume that there is a SameSite cookie set for a specific domain, that is, the trace contains a **cookie-jar-set** event at time t_1 , and that the browser then loads a page at time t_2 for which this cookie would have been sent if it was not SameSite (i.e., for which *cookie-match* is true but *cookie-match-samesite* is not). If there is a subsequent event at time t_3 , be it a **js-get-cookie** where the browsing context location matches the URL of the request at t_2 , or a **net-request** to which the cookie should be attached (i.e., for which the *cookie-should-be-sent* predicate is true), then the value of the cookie header (or the return value of `document.cookie`) *cookies'* should not contain the cookie set at t_1 , and that cookie was not attached to the request at t_2 .

Cookie Serialization Collision Resistance

In 2020, *nameless cookies* were introduced in the cookie RFC [nam] to standardize the legacy behavior adopted by major browsers. According to the standard, cookies with an empty name and a non-empty value must be serialized in the `Cookie` request header using only their value, without the `=` separator. To exemplify, a nameless cookie with value `foo` is serialized by compliant browsers as `Cookie: foo`. This serialization strategy is known to introduce collisions, which can be leveraged to perform cookie tossing attacks [SAVM23]. For example, a cookie set via `Set-Cookie: =foo=bar`, with empty name and value `foo=bar`, is attached to outgoing requests as `Cookie: foo=bar` resulting indistinguishable to a server from a cookie with name `foo` and value `bar` [CEWW22, §5.5, item 3].

Browsers can prevent cookie collisions by removing support for nameless cookies altogether, as in the case of Safari [SAVM23], or simply by including the `=` separator in the serialized cookie irrespectively of the content of the name or the value fields. Building on the previous example, the nameless cookie with value `foo=bar` would be serialized as `Cookie: =foo=bar`, allowing servers to distinguish it from a standard named cookie. This is captured by the following invariant.

Invariant (I.6). A cookie with name n and value v set for domain d is serialized to the string " $n=v$ " when attached to requests or accessed via `document.cookie`.

The invariant is encoded as:

```
COOKIE-SERIALIZATION-INVARIANT( $tr$ ) :=
 $t_2 > t_1 \wedge$ 
 $\text{cookie-jar-set}(\text{name}, \text{value}, \{\text{secure}, \text{same-site}, \text{path}, \text{domain}\})@_{tr}t_1 \wedge$ 
(
  ( $\text{net-request}(\_, \text{url}, \text{method}, \text{type}, \text{origin-url}, \_, \_, \text{redirs}, \{\text{cookies}\}, \_)@_{tr}t_2 \wedge$ 
     $\text{cookie-should-be-sent}$ 
      ( $\text{path}, \text{domain}, \text{secure}, \text{same-site}, \text{type}, \text{origin-url}, \text{url}, \text{method}, \text{redirs}$ ))  $\vee$ 
    ( $\text{js-get-cookie}(\text{ctx}, \text{cookies})@_{tr}t_2 \wedge \text{url} = \text{ctx-location}(\text{ctx}) \wedge$ 
       $\text{cookie-match}(\text{path}, \text{domain}, \text{secure}, \text{url})$ 
    )
  )  $\wedge$ 
 $\text{is-effective-cookie}(t_2, tr, \text{name}, \text{value}, \text{domain}, \text{path}, "") \rightarrow$ 
 $\text{name} \# "=" \# \text{value} \in \text{split-cookie}(\text{cookies})$ 
```

For every request (or access to the `document.cookie` property) at time t_2 , where a cookie stored previously in the cookie jar at time t_1 should be sent (resp. retrieved), the cookie header (or the return value of `document.cookie`) should contain the string $\text{name} \# "=" \# \text{value}$ after splitting on the separator ";". This invariant uses the three predicates *cookie-should-be-sent*, which is true if a cookie should be attached to a request, *cookie-match*, which is true if a cookie should be readable in a specific browsing context URL, and *is-effective-cookie*, which makes sure that the *cookie-jar-set* at t_1 we consider is the event that set the cookie in the cookie jar. Specifically, the predicate makes sure that there was no *cookie-jar-set* between t_1 and t_2 that overwrote the cookie stored in the cookie jar.

Confidentiality of Secure Cookies

The Cookies RFC delegates the decision of which protocols are denoted as *secure* to the specific user agent, requiring it to attach the cookies with the *Secure* attributes to URLs using such protocols [CEWW22]. Noticing this ambiguity in the RFC, we investigated how different browsers implement this behavior and discovered an inconsistency: Chromium and Firefox (behind a configuration flag) deem the `localhost` host, its subdomains, and its IP representation (`127.0.0.1`) as *secure* regardless of the protocol, and thus attach *Secure* cookies to local requests, whereas Safari does not. Similar inconsistencies apply to cookie prefixes, where only Firefox attaches prefixed cookies to `localhost`.

We contacted the HTTP Working Group [pot], notifying them about the potential differences in handling of *Secure* cookies, suggesting to disambiguate the requirements on browsers by using the *potentially-trustworthy* origin definition for determining *secure* URLs, instead of a browser-dependent definition of *secure* protocol. Our proposal is currently being discussed in the Working Group. Initial feedback suggests that the specification editors are considering modifying the phrasing to include *potentially trustworthy* origins.

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

This change in the specification would align it to the de-facto standard behavior of the majority of the top browsers, which we formalize as follows:

Invariant (I.7). *Cookies with the `Secure` attribute are only attached to requests sent to potentially trustworthy origins and are only readable by scripts running in browsing contexts whose origin is potentially trustworthy.*

The invariant is encoded as:

$$\begin{aligned} \text{SECURE-COOKIES-CONFIDENTIALITY}(tr) := & \\ & t_1 > t_0 \wedge \\ & \text{cookie-jar-set}(\text{name}, \text{value}, \{\text{secure} = \text{true}, \text{same-site}, \text{path}, \text{domain}\})@_{tr}t_0 \wedge \\ & (\\ & \quad (\text{net-request}(\text{id}, \text{url}, \text{method}, \text{type}, \text{origin-url}, _, _, _, \{\text{cookies}\}, _)@_{tr}t_1 \wedge \\ & \quad \quad \text{cookie-should-be-sent}(\\ & \quad \quad \quad \text{path}, \text{domain}, \text{false}, \text{same-site}, \text{type}, \text{origin-url}, \text{url}, \text{method}, \text{redirs})) \vee \\ & \quad \quad (\text{js-get-cookie}(\text{ctx}, \text{cookies})@_{tr}t_1 \wedge \text{url} = \text{ctx-location}(\text{ctx})) \\ & \quad) \wedge \\ & \quad \text{cookie-match}(\text{path}, \text{domain}, \text{false}, \text{url}) \wedge \\ & \quad \text{name} \# \text{"="} \# \text{value} \in \text{split-cookie}(\text{cookies}) \wedge \\ & \quad \text{is-effective-cookie}(t_1, tr, \text{name}, \text{value}, \text{domain}, \text{path}, \text{""}) \rightarrow \\ & \quad \text{is-origin-potentially-trustworthy}(\text{url}) \end{aligned}$$

Assume that there is a cookie in the cookie jar with the `Secure` attribute set, i.e., the trace contains a **cookie-jar-set** event at t_0 . If there is a network request (or an access to the `document.cookie` property) at t_1 where the cookie should be sent (resp. retrieved) and it is actually part of the attached cookies (resp. present in the return value of `document.cookie`), i.e., $\text{name} \# \text{"="} \# \text{value} \in \text{split-cookie}(\text{cookies})$, then the origin of the URL of the request (or the browsing context where `document.cookie` is called) is potentially trustworthy.

Blockable Mixed Content

For each request, the browser determines whether it should be blocked by applying the steps defined in the *Should fetching request be blocked as mixed content* algorithm [W3C23, §4.4]. In particular, a request is allowed when either its URL is *potentially trustworthy*, the context in which the request is performed does not restrict mixed content requests (e.g., a page loaded via HTTP making a fetch request), or when the request is top-level. We can define the following invariant.

Invariant (I.8). *For every network request performed by the browser, either: (i) the context does not prohibit mixed content requests; or (ii) the request URL is potentially trustworthy; or (iii) the request is top-level.*

The encoding of the invariant is:

$$\begin{aligned} \text{BLOCKABLE-MIXED-CONTENT-FILTERED}(tr) := \\ \text{net-request}(_, url, _, type, origin, doc-url, ancestors, _, _, _)@_{tr}t1 \rightarrow \\ (\neg \text{does-settings-prohibits-mixed-security-contexts}(\\ origin, doc-url, ancestors) \vee \\ \text{is-url-potentially-trustworthy}(url) \vee \\ (type = \text{main_frame} \wedge nil = ancestors)) \end{aligned}$$

The invariant uses the predicates *is-url-potentially-trustworthy*, which is true if the request URL is potentially trustworthy according to the respective algorithm of the secure context specification, and *does-settings-prohibits-mixed-security-contexts*, that is the implementation of the respective algorithm defined by the Mixed Content specification [W3C23, §4.3] and is true if the request initiator origin is potentially trustworthy, or if any ancestor of the navigation initiator has a potentially trustworthy origin. The invariant also uses the expression $type = \text{main_frame} \wedge nil = ancestors$ to check if a request is a top-level navigation.

Upgradeable Mixed Content

For upgradeable mixed content requests, e.g., loading images over insecure channels, the browser should rewrite the URL of the request by changing its scheme from HTTP to HTTPS. The mixed content specification defines the conditions for applying this rewriting in the *Upgrade mixed content request to a potentially trustworthy URL* algorithm [W3C23, §4.1]. This algorithm applies to every request by the Fetch specification, thus every successful request made by the browser for upgradeable mixed content should have been upgraded. That is, every non-top-level request whose URL is not potentially trustworthy should not be upgradeable or should be permitted by Mixed Content. This corresponds to the following invariant:

Invariant (I.9). *For every non-toplevel network request performed by the browser whose URL is not potentially trustworthy, the request context does not prohibit mixed content or the request type is not upgradeable.*

The invariant is encoded as:

$$\begin{aligned} \text{UPGRADEABLE-MIXED-CONTENT-FILTERED}(tr) := \\ \text{net-request}(_, url, _, type, origin, doc-url, ancestors, _, _, _)@_{tr}t1 \wedge \\ \neg \text{is-url-potentially-trustworthy}(url) \wedge \\ type \neq \text{main_frame} \rightarrow \\ (\neg \text{does-settings-prohibits-mixed-security-contexts}(\\ origin, doc-url, ancestors) \vee \\ \neg \text{is-mixed-content-upgradeable}(type)) \end{aligned}$$

where the presence of a request in the trace whose URL is not potentially trustworthy and whose type is different from *main_frame* (as upgradeable mixed content does not restrict toplevel requests) implies that both *is-mixed-content-upgradeable*, which checks if the request type is upgradeable (by implementing its definition in [W3C23, §3.1]), and *does-setting-prohibits-mixed-security-contexts* are false.

3.4 Trace Verification Pipeline

In this section, we will first motivate with an example the importance of abstracting WPT tests into execution traces in order to automate the discovery of Web invariant violations, and then describe our verification pipeline in detail.

3.4.1 Motivating Example

We present a simple example to motivate why looking at failed WPT tests does not already enable reasoning about security. The WPT test `/mixed-content/gen/top.meta/unset/img-tag.https.html` is a set of test cases that check the mixed content behavior of browsers when fetching `img` tags. In particular, the test expects image requests to always be performed within an HTTPS browsing context (i.e., a window with a HTTPS URL as location). This is expected, as upgradeable mixed content requests should be allowed when the browser is able to rewrite the request URL to use the HTTPS scheme, i.e., performing the auto-upgrade. This test is successful on the stable versions of Firefox and Safari, but fails on Chromium, as some of the requests fail.

The execution trace of the test contains multiple *net-request* events, each corresponding to the requests performed by the browser during execution. Specifically, for each embedding of an `img` tag, the event includes the image URL, the request type (*image*), and additional fields characterizing the request, e.g., the origin of the request initiator and the URL of the document where the new image will be loaded. The I.8 invariant mandates that for every *net-request* event, at least one of three conditions must hold for it to be compliant with the Mixed Content specification. Since the request is not top-level, i.e., its type is *image*, and it originates from a page loaded via HTTPS, i.e., *does-setting-prohibit-mixed-content* is true, then its URL must be potentially trustworthy, i.e., its scheme must be HTTPS. In the traces produced during the execution of Firefox and Safari, the *net-request* event corresponding to the embedding of the image has an insecure URL, i.e., the image is fetched via HTTP, violating the requirement of I.8. In Chrome, on the other hand, the request is auto-upgraded and the corresponding *net-request* has a potentially trustworthy URL, thus I.8 is not violated.

Since the WPT test only checks for the images to be loaded, without explicitly testing their protocol, Firefox and Safari, which do not currently implement protocol auto-upgrading [mc-b, mc-a] and perform the mixed content requests without blocking them, pass the test. Chromium, on the other hand, performs the auto-upgrading as mandated by the Mixed Content specification. However, since the image is served on a non-standard HTTP port (8000), the browser upgrades the protocol without changing the port causing a connection error.

This example highlights that the WPT test results alone may not always capture potential security concerns since failed tests do not necessarily break Web invariants, and, conversely, successful tests might break Web invariants. Tests can not only be unsuccessful because browsers implement new security features, as in the example above, but they can also fail

if the execution relies on unimplemented APIs. This further emphasizes that observing a discrepancy across the WPT results of different browsers (i.e., simple WPT-based differential testing) is not a direct indication of security issues. By verifying browser traces obtained during the execution of WPT tests, irrespectively of test results, our approach provides a deeper insight into each test. In particular, violating an invariant is a clear indicator of potential security issues in the exercised browser behavior, pinpointing the specific Web components requiring immediate attention.

3.4.2 Methodology

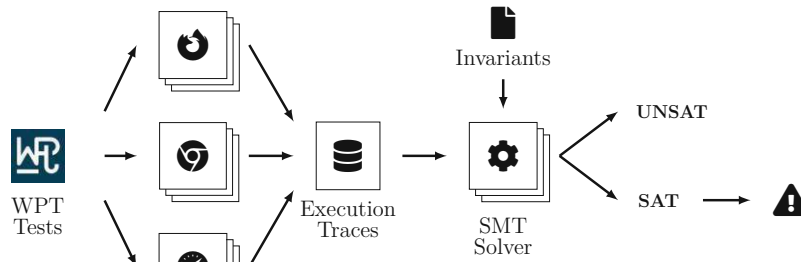


Figure 3.2: Trace Verification Pipeline.

Our methodology for detecting security-relevant issues in browser implementations leverages the WPT test suite and consists of two main stages, as shown in Figure 3.2. First, the execution traces produced by executing the WPT tests on the three major Web browsers (Chromium, Firefox, Safari) are collected into a database. Second, the obtained traces are post-processed, translated to SMT-LIB, and checked against the Web invariants we define in Section 3.3 using an SMT solver. When the solver cannot prove the validity of the invariant on a test trace (SAT, i.e., a counterexample exists), a violation is found on the specific browser. Our analysis pipeline is based on the Kubernetes container orchestration platform, allowing us to execute multiple instrumented browsers and the SMT solving in parallel. We detail in the following the main steps of the pipeline and our criteria for selecting the relevant tests.

Test Selection. The tests part of the WPT project can be classified into four main categories: (i) rendering tests, which test the graphical output of the browser (by, e.g., comparing it to screenshots) to verify that pages are displayed as expected; (ii) `testharness.js` tests, which test JavaScript interfaces available in browsing contexts, allowing to automatically check assertions about their behavior; (iii) `wdspec` test, which test parts of the WebDriver protocol and are written in the Python programming language; (iv) manual tests that require human interaction to determine their result. In this work, we focus on `testharness.js` tests, since our Web invariants cover JavaScript and browser internals behavior, ignoring most UI aspects. In particular, we consider all `testharness.js` tests of the April 2023 version of the WPT test suite. We detail our test selection

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

in Table B.1 (Appendix B.2), where we report the version (commit hash) of the test suite, the considered WPT subfolders, and the respective number of tests for each folder.

Trace Collection. We run each WPT test in its own isolated ephemeral container named *runner*. Each runner container includes a specific version of the tested browser, all its run-time dependencies, our patched version of the WPT tooling, and the browser instrumentation composed of a browser extension and a proxy (Section 3.4.3). For Safari, the runner container executes a MacOS virtual machine containing the instrumented browser. We build a runner container for Chromium (version 118.0.5961.0), Firefox (version 116.0.3) and Safari (version 16.4). Once the runner container terminates the execution of a WPT test, it stores the execution trace in JSON format in a centralized database. Note that we ignore test assertions, storing the captured trace regardless of the test results.

Verification. Upon completion of the runner container, the generated JSON file is post-processed and translated to SMT-LIB format. In particular, the events that were captured by our browser instrumentation are converted to execution traces following the format described in Section 3.3.1. It may be the case that multiple events recorded by the browser instrumentation happened simultaneously, i.e., the JSON stores multiple events with the same timestamp. This may occur when, for instance, a page containing multiple subresources is rendered: the browser may try to load all resources in parallel, thus resulting in multiple events of type *network-request* to be recorded at the same time. In such cases, the SMT-LIB translator generates multiple traces, each corresponding to a single permutation of the simultaneous events, allowing us to consider all possible orderings of the concurrent events. Note that, in practice, the number of concurrent events in WPT traces rarely exceeds four events, thus having a negligible impact on the pipeline performance.

Once execution traces are translated to SMT-LIB format, we use an SMT solver to query, for each trace, the *validity* of each Web invariant. That is, we check satisfiability of the negation of the invariant applied to each trace. This satisfiability checking may have three possible outcomes: (UNSAT) the invariant is valid, i.e., it is true for the current trace; (SAT) the invariant is not valid, i.e., the current trace is a counterexample for the invariant; (UNKNOWN) the solver was not able to prove nor disprove the invariant, hence in such cases we cannot draw any conclusion and we do not report any violation. Whenever the solver returns SAT, we obtain a model, i.e., an instantiation of the variables mapping them to the concrete values from the trace that make the invariant false. Being based on the standard SMT-LIB format, our pipeline supports all standard-compliant solvers that implement decision procedures for quantified string constraints, integer arithmetic and algebraic data types. Specifically, we currently support both the Z3 theorem prover and CVC5.

Violating an invariant may have several security implications, and for this reason, we manually inspect the execution trace of every SAT result and design a minimal proof of concept (PoC) attack to showcase the vulnerability in the affected browsers. We discuss

| Instrumentation | Ease of Implementation | Cross-Browser Compatibility | Observability |
|--------------------------|------------------------|-----------------------------|---------------|
| Browser Extensions | ↑ | ↑ | ↑ |
| Service Workers | ↑ | ↑ | → |
| WebDriver | → | ↑ | ↑ |
| Chrome Devtools Protocol | → | ↓ | ↑ |
| External Proxies | ↑ | ↑ | → |
| Source Patching | ↓ | ↓ | ↑ |

Table 3.2: Design space analysis: comparison of browser instrumentation methods. (↑ : High, →: Medium, ↓ : Low)

the discovered attacks in Section 3.5.2.

3.4.3 Browser Instrumentation

Browser instrumentation and trace collection are essential components of our pipeline. Our main goal is to develop a browser instrumentation solution that provides a balance between observability and cross-browser support, while minimizing the implementation effort. Our instrumentation must be easily integrated into existing testing pipelines such as the Web Platform Tests and work across different browsers.

Design Space Analysis

We consider several approaches to browser instrumentation:

Browser extensions are software modules that extend browser functionality. Extensions are powerful as they can access internal browser structures such as the cookie jar, monitor and intercept network traffic, and access and modify the DOM.

Service workers are scripts that run in the background and control the behavior of Web pages. They act as proxy servers between Web applications, the browser, and the network. However, they run on a different JavaScript context and have no access to the DOM.

WebDriver is a Web standard that describes a remote interface that allows the control and introspection of browsers.

Chrome Devtools Protocol (CDP) is a remote debugging protocol [CDP] which provides access to the DOM, network activity, and a JavaScript debugger.

External proxies act as intermediaries between a Web server and the browser and allow the monitoring and intercepting of network traffic.

Browser source code patching allows access to all internal browser structures and events, enabling the most comprehensive monitoring and trace collection.

We evaluate these options according to three criteria: (i) ease of implementation, (ii) cross-browser compatibility, and (iii) observability, i.e., how many events the instrumentation method can collect, and summarize the results in Table 3.2.

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

Ease of Implementation. *Browser extension* implementations generally depend on the complexity of the extension, which, for our use case, is directly proportional to the number of different events we mean to collect in the traces. However, since the extension API is well documented, and extensions are written in JavaScript, which is not verbose, we deem the ease of implementing our browser instrumentation using *browser extensions* high. *Service Workers* and the *remote protocols* would also be high in the *ease of implementation* scale for our use-case, for the same reasons as browser extensions, if not for the fact that *Service Workers* and *WebDriver* are either testing target or testing mechanisms of WPT, which means that the WPT framework and testing suite would require extra modifications, making its implementation more complex. *External proxies* also require low implementation effort for the same reasons as browser extensions, making use of expressive programming languages and well-documented APIs, with a small set of potential events to monitor. On the opposite end of the spectrum, *source code patching* requires extensive manual effort in understanding browser implementations, which are generally written in lower-level languages like C++ and are very extensive.

Cross-Browser Compatibility. *Browser extension* features are dictated by the manifest versions supported by a given browser. However, all major browsers support manifest versions v2 (Firefox and Safari) or v3 (Chromium), which significantly overlap in the supported APIs, making it possible to write powerful cross-browser extensions. The *Service Worker API* is a standard supported by all major browsers. The *WebDriver* protocol is also a standard, and its core set of functionality is supported across browsers. The *Chrome Devtools Protocol* is not a standard and, therefore, not supported consistently across different browsers [Fir], [Weba]. *External proxies* are completely browser agnostic and score high on compatibility. *Source code patching* concerns only a given browser, significantly hindering its cross-browser compatibility

Observability. *Browser extensions* provide access to browser internal structures and events, like network activity and the cookie jar, and allow the dynamic inclusion of arbitrary JavaScript in pages using only the overlapping APIs between manifest v2 and v3, completely cross-browser and independent from WPT. However, this approach also has some limitations. For instance, manifest v3 does not allow the inspection of network request and response bodies for privacy reasons [Chrb, Chra], and Chromium-based browsers no longer support manifest v2 extensions. *Service workers* are similar to *external proxies* in observability, as they are limited to monitoring network traffic since they do not have access to the DOM. Remote protocols like *WebDriver* and *CDP* also provide access to browser internal structures and events, like network activity and the cookie jar, and allow the dynamic inclusion of JavaScript in the target pages, similar in observability to browser extensions. *Source code patching* provides the highest possible level of observability since it grants access to every internal structure in the given browsers.

Implementation

Based on our design space analysis, we implemented a browser instrumentation solution which combines a browser extension with an external proxy that improves on the

limitations of the extension API with respect to its ability to inspect network traffic. Our solution provides the necessary hooks to monitor internal browser state, JavaScript API calls, and have a complete picture of the network activity when collecting browser execution traces.

Internal Browser State Monitoring. With extensions, we gain access to the internal browser state not available to regular scripts or external monitoring tools. This state includes the `CookieJar`, and network activity such as requests and responses. This internal state is accessible to extensions via background scripts, which have no access to the DOM but can make full use of the extension APIs. Network events are monitored by registering callback functions that run whenever a request is about to be sent, and when a request is deemed completed, i.e., it has a response or it was dropped. These callbacks provide access to the request and response headers, and additional information added by the browser, like the tab and frame IDs of the request initiator. The `CookieJar` can also be monitored via `onChange` callbacks, whose execution can be delayed depending on the state of the JavaScript event loop. Due to these inherent delay inconsistencies, we opted for polling the state of the `CookieJar` instead of registering callback functions, which gives us higher precision timestamps for `CookieJar` events.

JavaScript API Call Monitoring. In addition to monitoring network events and internal browser state, we focus on JavaScript API calls as another category of relevant events for our analysis. We proxy the relevant JavaScript functions, logged as events used in the invariants, to record function calls in a centralized structure located in the extension's background script. This proxying is done through `Proxy` objects and method overriding, enabling us to collect all the relevant data associated with each API call, such as its arguments and the respective browsing context. Our instrumentation logs calls to the setter and getter of `document.cookie`, but more JavaScript methods could be supported in the future using similar techniques. We adopt a dynamic approach for our instrumentation using content scripts, which are extension scripts that run in the context of webpages. Each webpage is injected with a content script that installs the proxy functions according to the extension configuration. This dynamic instrumentation is more versatile and scalable compared to code rewriting methods and allows us to efficiently track and analyze JavaScript API calls as they occur in real-time.

External Proxy. We incorporate an external proxy into our framework to overcome two main issues: (i) the restrictions imposed by browsers over network content deemed sensitive and, hence, inaccessible to background scripts but visible through a network proxy (e.g., request and response bodies); (ii) the inconsistent delay between network events and the execution of their corresponding callback event handlers. When a network request leaves the browser, the callback corresponding to its event handler is queued in the extension's JavaScript event loop and eventually executed. If the proxy intercepts the request before the callback is executed, the proxy event's timestamp is more accurate and is used as the request event timestamp in the trace.

Limitations

While our browser instrumentation technique based on extensions and proxies offers a powerful means to monitor internal browser state, JavaScript API calls, and network events, enabling comprehensive browser security analysis with cross-browser compatibility and minimal code modification or rewriting, it is essential to acknowledge the inherent limitations of this approach. These limitations include:

Browser Discrepancies. In our instrumentation, we strive to use only browser extension APIs that are compatible across browsers. However, browser behavior varies across implementations, which can introduce limitations to our approach. These inconsistencies are detected by manual inspection of our results, and whenever possible, we implement specific workarounds. But, some issues require changes to the browsers' source code and bug fixes. For example, a bug in Firefox's URL matching prevents the content scripts from being injected into opaque origins, such as data URL iframes. This limitation hinders our ability to monitor JavaScript API calls in these frames, negatively impacting the comprehensiveness of our analysis, and it cannot be circumvented without changes to Firefox's source code. For our current usage, this translates to missing events executed in iframes with opaque origins, which can lead to false negatives. Another example is Safari's resource isolation for WebDriver-controlled instances, which isolates resources such as the cookie jar. This isolation prevents our extension from effectively monitoring specific resources such as the `CookieJar` within Safari instances controlled by WebDriver, which translates to missing `CookieJar`-related events for Safari execution traces, leading to false negatives.

API Constraints. Currently our instrumentation is able to monitor the necessary components and collect the events required to reason about our invariants, i.e., `CookieJar`, network, and some JavaScript API call events. While some extensions to our instrumentation are possible, they are constrained by the availability of APIs in both JavaScript and the extension environment, and by the Same Origin Policy which is applied to the injected content scripts. For example, information such as the effective Content Security Policy (CSP) of a frame cannot be directly monitored as it is not accessible to scripts running in pages, nor to browser extensions. To support the analysis of the CSP mechanism with our approach, we must develop inference and heuristic techniques, which we could use alongside other artifacts of our instrumentation, such as response headers, to infer the CSP enforced on a given frame. Another constraint to our approach is monitoring the DOM. Content scripts injected by the extension are still subject to the Same Origin Policy. Therefore, a full picture of the DOM may prove difficult to obtain without heuristics over other events such as network activity and DOM mutation.

In summary, while our browser instrumentation technique was proven effective in collecting security-relevant browser execution traces, these limitations underline the importance of developing better introspection and instrumentation mechanisms for browser testing. These mechanisms would benefit not only our approach but also testing frameworks like WPT, which currently uses incomplete workarounds to test features like cookies and the

Content Security Policy.

3.4.4 Discussion: Extensibility

The methodology we propose is meant to enable specification maintainers and browser developers to check their security expectations, expressed as Web invariants, against multiple implementations. This way, security issues can be identified early during development and across Web platform or browser updates, e.g., for regression testing.

In this chapter, we encode 9 invariants, as discussed in Section 3.3, showing that the verification pipeline is not bound to a single security mechanism and can be extended to support additional Web features. Although we do not consider the required expertise to develop new invariants a limiting factor, given that specification maintainers already possess this knowledge, the expressiveness of the invariants may be limited by the introspection capabilities of our instrumentation. Specifically, every JavaScript API or property access that can be wrapped with Proxy objects can easily be traced, encoded as an event (as in Figure 3.1), and used in the definition of new invariants. Instead, monitoring internal browser state which is not exposed to pages or extensions, e.g., CSP, may prove to be difficult to trace without relying on heuristics or a different instrumentation approach (e.g., browser code patching [JK19]).

Automated generation. The definition of new Web invariants relies on the manual effort of understanding the security requirements of a specification and encoding them into a logical proposition. Automation could be beneficial for aiding the process, allowing more properties to be covered. Previous work on Web invariants identifies the importance of clearly defining the security properties of the Web as a way to have a *sound* scientific understanding of Web security [ABL⁺10]. Thus, the generation of Web invariants presents the challenge of retaining soundness while characterizing the relevant Web mechanisms. We leave the development of a methodology to automatically extend the set of invariants as future work.

3.5 Evaluation Results

We evaluate our methodology by verifying, using our pipeline, the 9 Web invariants we define in Section 3.3 against the execution traces of the 24896 testharness tests from the April 2023 version of the WPT suite. Note that every browser is executed 24896 times, totaling 74688 traces. We use the Z3 theorem prover as the SMT solver component since it proved to be the best performing for our invariants. We set a timeout of 10 minutes for the execution of the browser for each test, and 10 minutes for each Z3 query. When Z3 is not able to return an answer within the timeout it returns UNKNOWN. All our experiments have been conducted on a cluster with 132 VCPUs (AMD EPYC 2.0GHz) and 382GB of RAM.

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

| Invariant | 🔍 | | 🔄 | | 🚫 | |
|-----------|-----|------|-----|------|-----|------|
| | SAT | UNK. | SAT | UNK. | SAT | UNK. |
| I.1 | 0 | 0 | 0 | 0 | – | – |
| I.2 | 0 | 0 | 0 | 0 | – | – |
| I.3 | 0 | 0 | 0 | 0 | – | – |
| I.4 | 0 | 0 | 1 | 0 | – | – |
| I.5 | 10 | 0 | 6 | 0 | – | – |
| I.6 | 15 | 0 | 9 | 0 | – | – |
| I.7 | 0 | 0 | 0 | 0 | – | – |
| I.8 | 0 | 448 | 24 | 643 | 21 | 692 |
| I.9 | 0 | 355 | 18 | 509 | 0 | 628 |

Table 3.3: Trace verification results.

3.5.1 Preliminary Results

Table 3.3 reports the outcome of our analysis of the three major browsers on the WPT test suite, showing the number of tests for which Z3 found violation of a Web invariant (SAT). Additionally, we report the number of UNKNOWN results, for which our pipeline could not generate a definitive answer. Note that, given the limitations of Safari instrumentation (see Section 3.4.3), invariants about cookies are expected to always return UNSAT there (marked as – in Table 3.3), since the Safari traces never contain the *cookie-jar-set* event, which is used in the premises of our cookie invariants.

Five invariants have at least one violation. The results confirm our expectation that different implementations may exhibit different behaviors with respect to the implemented security mechanisms. In particular, although there is overlap in some of the SAT traces between different browsers, Table 3.3 highlights that some SAT results are browser-specific. We discuss in Section 3.5.2 the security implications of violating each invariant, where we group SAT results into concrete attacks against specific browsers that we present as case studies.

For four invariants our pipeline does not report any violation, so they are valid on the entirety of the execution traces produced by WPT. This may happen in the cases where the invariants are well-known and expected to hold by the literature (I.1, I.2). Additionally, we may obtain no violation if the traces generated by the test suite do not cover the specific preconditions for an attack to be performed. As an example, I.3 does not hold in the current Web platform [VFB⁺23] because of an attack that requires combining *domain relaxation*, i.e., assignment to the Document.domain property, with `__Host-` cookies. This invariant may have no SAT results because the WPT test suite never uses the two Web features together in the same test. A similar consideration applies to I.7, as the `localhost` URL is never used in cookie-related tests. We discuss these cases in Section 3.5.4, where we explore additional tests beyond what is included in WPT.

Z3 returned UNKNOWN during the verification of the Mixed Content invariants I.8 and I.9. These are caused by the complex checks that are mandated by the Mixed Content

| | Trace Collection | | | Verification | | | Total |
|---|------------------|-----|------------|--------------|--------|------------|------------|
| | avg | std | total | avg | std | total | |
| 🌀 | 28s | 6s | 23h 29m | 19s | 1m 42s | 23h 05m | 1d 22h 35m |
| 🌀 | 40s | 8s | 1d 07h 18m | 27s | 2m 06s | 1d 08h 34m | 2d 15h 52m |
| 🌀 | 27s | 8s | 1d 06h 34m | 32s | 2m 28s | 1d 14h 33m | 2d 21h 07m |

Table 3.4: Trace verification execution times.

specification, in particular the recursive checking of the entire ancestor chain for each network request, which may negatively affect the solver speed and result in UNKNOWN if the execution time exceeds the verification timeout.

Performance. The performance of our trace verification pipeline is shown in Table 3.4. The total run-time for each of the three major browsers is reported together with the time required for executing the browser (collecting execution traces) and the Z3 verification time. Executing a single WPT test on each of the browsers consistently requires less than one minute, whereas the verification with the Z3 theorem prover shows more variability, while still requiring less than a minute on average. This confirms that verifying Web invariants on the traces generated by WPT does not add substantial overhead to the execution of the testing suite, but supplements the result obtained from each WPT test with an assessment of the security of the exercised browser functionality.

3.5.2 Attacks on Major Browsers

Every SAT result obtained as the output of the Z3 theorem prover corresponds to a violation of a Web invariant on the execution trace of a specific browser, as captured by our instrumentation. These results require a manual analysis to identify and aggregate similar issues, organizing them into concrete inconsistencies. This effort is supported by the model obtained from Z3, which provides the concrete values from the trace that violate the invariant, highlighting problematic events in the trace and allowing us to easily discern the cause of the violation. A goal of our analysis of SAT results is to determine the root causes underlying these inconsistencies and to quantify their security impact, and in particular, if they can lead to concrete real-world attacks. This step is also critical in identifying any false positives introduced by the observability limitations of our browser instrumentation (Section 3.4.3). For instance, the inability to correctly observe a specific browser event may lead to the generation of a violating trace for an otherwise compliant browser. For example, a missing cookie deletion event may result in a violating trace if we expect that cookie to be attached to a subsequent network request.

We now present all the attacks resulting from the analysis of the SAT results, discussing them in the form of case studies. In particular, we aggregated all 104 invariant violations into 10 confirmed attacks and 5 false positives as shown in Table 3.5.

🌀 Framed Pages Mixed Content Bypass

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

| Invariant | Total SAT | SAT Traces | | | Type | Description (causes of SAT) |
|-----------|-----------|------------|----|----|------|---|
| | | 🕒 | 🕒 | 🕒 | | |
| I.4 | 1 | – | 1 | – | 🚩 | SameSite cookie integrity violation |
| I.5 | 18* | 1 | 2 | – | 🚩 | SameSite cookies attached to (favicon, subresource, fetch) requests (<i>requests</i>) |
| | | 10 | 5 | – | 🚩 | SameSite cookies accessible via Document.cookie (<i>non-HTTP</i>) |
| | | 1 | – | – | 🚩 | SameSite cookies attached to location.reload() network requests (<i>reload</i>) |
| | | 1 | – | – | ⊗ | Incorrect event ordering |
| I.6 | 16 | 2 | 3 | – | 🚩 | Nameless cookies serialization collision |
| | | 2 | 1 | – | ⊗ | Missing events from sandboxed iframes |
| | | 5 | 2 | – | ⊗ | Missing delete cookie event |
| | | 1 | – | – | ⊗ | Incorrectly tagged requests: missing request initiator origin |
| I.8 | 45 | – | – | 1 | 🚩 | Framed pages mixed content bypass |
| | | – | – | 1 | 🚩 | Sandbox attribute mixed content bypass |
| | | – | – | 7 | 🚩 | Mixed content beacon requests not blocked |
| | | – | 11 | – | 🚩 | Mixed content Websocket requests not blocked |
| | | – | 13 | 10 | 🚩 | Mixed content autoupgrade not performed |
| | | – | – | 3 | ⊗ | Incorrectly tagged requests: missing request type |
| I.9 | 18 | – | 18 | – | 🚩 | Mixed content autoupgrade not performed |

Table 3.5: Aggregated SAT results. (🚩: attack; ⊗: false positive; *: the same trace may contain multiple attacks)

Z3 reported SAT for Safari for the trace of the `mixed-content/nested-iframes.window.html` test, where the browser successfully performs a fetch request to an insecure endpoint coming from a frame whose origin is potentially trustworthy, violating the I.8 invariant. After some investigation, we concluded that Safari incorrectly performs mixed content checks, i.e., secure pages embedded in insecure origins were not considered potentially trustworthy, and therefore, mixed content was not blocked except for requests to load scripts, stylesheets, or requests to insecure WebSocket. For example, if `https://bank.com` contains an authenticated mixed content request (i.e. via fetch), framing it over `http://attacker.com` will cause the request to not be filtered. This behavior might incorrectly expose non-Secure cookies in clear over the network to passive network attackers. Moreover, the integrity of the fetch request (and its response) would not be ensured against network attackers, meaning that attackers could tamper with its contents to, for example, alter the control flow of JavaScript execution on the target page.

Disclosure. We disclosed the attack to the Safari developers. The issue has been fixed in Safari 16.6.

🕒 Sandbox Attribute Mixed Content Bypass

The test `mixed-content/csp.https.window.html` consists in a webpage using the `sandbox allow-scripts` CSP directive. The page is loaded via HTTPS so mixed content should be prohibited, nevertheless, a fetch request targeting an HTTP endpoint is not blocked in Safari, violating I.8. In the trace, the CSP directive is effectively setting the origin of the page to `null`. Since the null origin is not potentially trustworthy, the requests are not filtered. This vulnerability can be combined with the previous one to obtain a complete bypass of the mixed content policy: the presence of the `sandbox` directive makes the browser allow mixed content requests to scripts, stylesheets, and insecure WebSockets,

which are otherwise blocked. As a consequence, if `https://bank.com` contains a mixed content script, framing it with the `sandbox` attribute over `http://attacker.com` will allow the request to the script to be sent. A network attacker can tamper with its content to obtain code execution on `https://bank.com`, in a context where the origin is `null`. In this scenario SOP prevents certain operations (e.g., cookies access) but other attacks, such as user input tracking, and DOM modifications can still be performed. For instance, an attacker embedding the login page of `bank.com` can track user inputs by registering new listeners through the injected script and exfiltrate user credentials whenever a user is tricked into logging in.

Disclosure. We disclosed this attack to the Safari developers. The issue has been fixed in Safari 16.6 and CVE-2023-38592 was assigned to this and the previous vulnerability.

🔍 Mixed Content Beacon Requests Not Blocked

A beacon request is a non-blocking POST request sent using the `navigator.sendBeacon` API. Mixed content beacon requests are *blockable* and therefore should be filtered. However, our pipeline SAT results show that Safari performs such requests, violating I.8. When a mixed content beacon request is not blocked, attached cookies and the data attached to the request are leaked even to passive network attackers.

Disclosure. We reported the problem to the Safari developers and we are waiting for confirmation.

🔍 Nameless Cookies Serialization Collisions

Part of the SAT results reported for I.6 are caused by the serialization of nameless cookies. Our invariant expects every cookie with name n and value v to be serialized as $n = v$. However, Chromium and Firefox serialize nameless cookies where $n = ""$ simply as v . Consequently, our pipeline will report a violation whenever I.6 matches a trace where a nameless cookie is serialized. The higher number of SAT results related to nameless cookies in Firefox compared to Chromium stems from an inconsistency between the browsers: whenever Firefox encounters the JavaScript API call `document.cookie = ""`, a cookie with an empty name and value is set, unlike Chromium, which does not set any cookie. The serialization of nameless cookies enables attackers to shadow arbitrary cookies. This capability includes shadowing *Secure* cookies from insecure origins, relaxing an attacker's requirements to perform cookie tossing or eviction attacks on *Secure* cookies, which would typically require a secure origin [SAVM23].

Disclosure. The issue was already reported to the IETF HTTP Working Group by Squarcina et al. [SAVM23] during their study of cookie integrity.

🔍 SameSite Cookie Integrity Violation

Our pipeline returned SAT for Firefox in the trace of the `cookies/samesite/setcookie-navigation.https.html` test, where a cookie with the `SameSite` attribute set to `Strict` is successfully set in the response to a cross-site network request initiated from an `iframe`, violating the I.4 invariant. In particular, an `iframe` loading `https://attacker.com`

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

within `https://bank.com` might navigate itself to some page at `https://bank.com`, which sets `SameSite` cookies in the response to the navigation request. Note that this applies to both `Strict` and `Lax` `SameSite` cookies. A gadget attacker [BJM08b, ABL⁺10] can thus leverage this behavior to overwrite cookies to perform, e.g., de-authentication attacks.

Disclosure. We reported this vulnerability to Firefox developers [wpta] who confirmed the issue assigning it a severity rating of *Normal (blocks non-critical functionality)*, planning a fix for the next release.

SameSite Cookies Isolation

The SAT results returned from our pipeline for I.5 fall into three categories: *request*, *non-HTTP*, or *reload*. Traces in these categories all have a similar setup but differ in how the cookie is retrieved. The setup follows this structure: (i) a cookie *c* with `SameSite` attribute set to `Strict` or `Lax` is set for domain *d*; (ii) a top-level request initiated by domain *d'*, where *c* is not attached, opens page *p* with domain *d*, which is cross-site with *d'*. From this point, *request* traces perform a network request, initiated by *d* (from page *p*) that is considered same-site and attaches *c*, violating I.5. This request can be, for example, a subresource load, a request to load the `favicon`, or a request generated by a call to the `fetch` JavaScript API. *Non-HTTP* traces retrieve the cookie *c* through a call to `document.cookie` from *p*, violating I.5. Finally, *reload* traces perform a call to `location.reload`, triggering a same-site request that reloads page *p* and attaches *c*, which violates I.5. Note that *reload* traces are not SAT for Firefox. By manually investigating this inconsistency, we discovered that Firefox does not attach `SameSite` cookies to network requests initiated from calls to `location.reload`, as it considers these requests cross-site.

Setting the `SameSite` attribute of cookies to `Strict` is considered an effective defense against CSRF and XS-Leak attacks as these cookies are not attached to cross-site requests. However, attackers can exploit the browser behavior highlighted by I.5 SAT results to bypass these restrictions. In particular, attackers can forge same-site requests starting from a cross-origin position by abusing, e.g., redirection gadgets that trigger attacker-controlled same-site navigation requests, effectively enabling CSRF attacks. Another security implication is the possibility of performing XS-Leaks. Consider a page that loads a script depending on whether the subresource load request attaches `SameSite=Strict` cookies and that this script modifies the DOM of the target page, altering `window.length`. An attacker could navigate to this page through `window.open`, and even though `SameSite=Strict` cookies are not attached to the top-level request, they will be included in subresource loads in the target page. An attacker can then use the `length` property of the window handler to infer the authentication status of the victim.

Disclosure. We are currently engaging with the HTTP Working Group to clarify the security properties that should be expected from `SameSite` cookies [sam].

🕒 Mixed Content WebSockets Requests Not Blocked

These SAT results refer to a set of tests for the following scenario: WebSocket requests sent from a Worker using the *ws* protocol. If the Worker is created from a secure page, so its origin is potentially trustworthy, we expect the request to be blocked as mixed content. However, in Firefox it is not, violating I.8. Investigating the issue uncovered that Firefox incorrectly implements the filtering for WebSocket requests. In particular, filtering is not performed if either the origin's scheme is *blob:* or the request is sent from a Worker created in a trustworthy origin using a *data:* URI.

Disclosure. We disclosed the problem to Mozilla. The issue has been fixed in Firefox 120.

🕒🕒 Mixed Content Autoupgrade Not Performed

From the analysis of these SAT results, we observed how both Safari and Firefox do not perform protocol autoupgrading, and as a consequence, upgradeable mixed content requests are sent over the network, violating I.8 or I.9. When this happens, network attackers can tamper with the content of upgradeable requests to attempt phishing users by e.g. swapping the icons of two buttons tricking them into performing destructive operations (e.g., *delete message* instead of *send message*). To prevent these attacks, the latest revision of the specification forbids loading upgradeable mixed content, but, as of today, neither Firefox nor Safari are compliant. However, they are aware of the issue and are planning a fix [mc-b, mc-a].

3.5.3 False Positives

In this section, we examine the false positives we obtained during our evaluation of the Web invariants against WPT traces and discuss their causes.

🕒 Incorrect event ordering

For one trace, our pipeline returned SAT for I.5 due to out-of-order events. Since our monitoring of network events is based on callbacks, which are subject to scheduling delays, and our monitoring of `CookieJar` events is polling-based, the order in which these events are collected may not match the concrete browser execution. Invariant I.5 matches a specific order of events, i.e., a `cookie-jar-set` event setting cookie *c*, followed by a cross-site network request that opens a page *p* where cookie *c* is not attached, and an access to cookie *c* from page *p*. Consider a concrete browser execution where a first network request leads to a cookie being set, which is then attached to a subsequent request. If the first two events are swapped in the trace, this incorrect trace can be matched by invariant I.5, leading to a violation.

🕒🕒 Missing events from sandboxed iframes

Our pipeline reported SAT for the traces of the test `cookies/samesite/sandbox-iframe-subresource.https.html` on Chromium and Firefox for I.6. In this trace, a previously set cookie is expected to be attached to a network request from an iframe. However, since the `iframe` has the `sandbox` attribute, it cannot attach existing cookies to network

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

requests. Since our instrumentation cannot observe events originating from sandboxed iframes, nor detect whether an iframe is sandboxed, invariant I.6 cannot account for this behavior. In this trace, browsers correctly withheld a cookie that I.6 expects to be attached to a network request, leading to an invariant violation.

🔒 Missing delete cookie event

In some cases, our browser instrumentation is unable to detect cookie deletion events. Missing cookie deletion events cause some of the SAT results for I.6. Consider an execution where a previously set cookie c is deleted before a network request that would attach c , but the cookie deletion event is missing from the trace. I.6 will expect the cookie to be attached to the network request since, according to the trace, that cookie is still in the `Cookiejar`. However, since in the browser execution the cookie no longer exists, it is not attached to the network request, leading to an invariant violation.

🔒 Incorrectly tagged requests

For three traces, Z3 returned SAT for I.8 on Safari. These are caused by the lack of the request-type field in the Request object returned by the instrumentation for network events. In particular, a toplevel request to a URL which is not potentially trustworthy should be allowed. However, the absence of the request type makes the expression $type = main_frame$ false in I.8, violating the invariant. Similarly, one Chromium trace violates I.6 since a network event in the trace is missing the *origin* field, i.e., the origin of the request initiator. Since the *origin* field is used by *cookie-should-be-sent* to determine if a SameSite cookie should be attached to a request, a request missing the initiator origin information and containing no cookie can be incorrectly tagged as violating when *cookie-should-be-sent* incorrectly (because of the missing origin) determines that cookies should be present.

3.5.4 Comprehensiveness of Tests

In this section, we explore additional tests beyond those in WPT, to (i) show that our pipeline can generalize to different test suites without modifications, and (ii) to assess how the *comprehensiveness* of the individual tests, in terms of the usage of Web features, affects the discovery of inconsistencies. As mentioned in Section 3.5.1, the limited scope of tests may prevent our pipeline from discovering violations. This is the case when tests do not include actions that are preconditions for the attack, e.g., when a violation is enabled by the combination of multiple Web features.

We construct a separate test suite comprising 9 tests to exercise behavior not covered by WPT. The selected tests are shown in Table 3.6. The first group (1-5) corresponds to the violations discovered by Veronese et al. [VFB⁺23] affecting the current Web platform. These tests combine multiple features to reproduce the attack traces generated by WebSpec. For instance, the first test uses *domain relaxation* to allow a subframe to set a `__Host-` cookie for a different origin. The remaining `webspec_*` tests use a combination of CSP, Service Workers, and Trusted Types. Given that our invariants

only focus on cookies and Mixed Content, these tests are not expected to reveal new violations. The second group of tests (6-7) reproduces the browser testing performed by Squarcina et al. [SAVM23]. In particular, the tests try to perform cookie tossing, eviction based on cookie jar overflow, and serialization collisions based on nameless cookies. Each test is composed of multiple sub-tests that correspond to various combinations of cookie properties, e.g., tossing of Secure cookies over insecure channels, or eviction of `__Host-` cookies. Note that these tests are actively abusing undefined behavior to perform eviction, as the RFC does not impose a specific limit to the number of entries in the cookie jar (although implementations are allowed to set one). Finally, the last two (8-9) tests use features that are not covered by WPT. The `localhost_cookies` test sets Secure, `__Secure-`, and `__Host-` cookies for the `localhost` domain, which is never used in WPT. The `multi_nested_frames` test sets cookies using mixed-content resources loaded across multiple levels of frames, as WPT does not include cookies in mixed-content tests and uses up to two levels of nesting.

Table 3.6 reports the results of running our pipeline on the traces produced by the new test suite. The experiment confirms that new violations can be discovered using more comprehensive tests. In particular, I.3 does not hold for Firefox, where domain relaxation allows compromising `__Host-` cookies integrity. Interestingly, Chrome satisfies the invariant, since starting from version 115, the `document.domain` property is immutable [imm], preventing pages from relaxing the SOP. The I.1 invariant does not hold for Chrome, as it is possible to set Secure cookies over an insecure connection when the URL is `localhost`. This matches the behavior we discuss in Section 3.3.1 and encode in I.7. Note that Firefox violates the invariant only when a specific setting flag is enabled. The new test suite, additionally, allows us to rediscover a violation for I.6, since the `crumbles_tossing` test uses nameless cookies. Similarly, I.8 and I.9 are SAT because upgradeable mixed content is not upgraded nor blocked in both Firefox and Safari. Safari also incorrectly loads mixed-content frames if the top-level window is loaded via HTTP, regardless of the protocol used to load any intermediate frame. Specifically, in `multi_nested_frames`, the test opens a window with three nested frames, where the top-level window is loaded via HTTP, the intermediate frames are over HTTPS, and the innermost frame is over HTTP, which should be blocked.

This experiment shows that employing a more comprehensive test suite has the potential to identify additional violations. While our focus for this work is WPT, as it is currently the most complete and regularly updated browser testing suite available, our pipeline can be applied to any alternative testing suites, potentially improving its efficacy.

3.6 Related Work

Browser Testing. BrowserAudit is a test suite designed to assess the implementation of Web security mechanisms in Web browsers [HMN15]. It includes more than 400 automated test cases for SOP, CSP, CORS, cookies and security headers. While the approach is undeniably useful to detect bugs, it suffers from significant limitations compared with

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT






| Test Name | SAT | | | | | |
|-------------------------|---|---|---|-----|---|---|
| | I.1 | I.3 | I.6 | I.7 | I.8 | I.9 |
| 1 webspec_host_frames | – |  | – | – | – | – |
| 2 webspec_csp_sw | – | – | – | – | – | – |
| 3 webspec_csp_sop | – | – | – | – | – | – |
| 4 webspec_tt_frames | – | – | – | – | – | – |
| 5 webspec_csp_blob | – | – | – | – | – | – |
| 6 crumbles_tossing (5) | – | – |  | – | – | – |
| 7 crumbles_eviction (8) | – | – | – | – | – | – |
| 8 localhost_cookies (3) |  | – | – | – | – | – |
| 9 multi_nested_frames | – | – | – | – |  |  |

Table 3.6: Additional tests and new violations.

our proposal. First, test cases in BrowserAudit were manually created by the authors. Our approach instead leverages WPT, which is an actively maintained existing test suite backed up by a large community (to date, its GitHub repository counts more than 1,500 contributors). Moreover, the security implications of failed BrowserAudit tests are also manually identified: failures are categorized by the authors as warning or critical, supposedly based on their security impact according to the authors’ understanding. Our approach instead detects effective violations of Web security invariants, i.e., deviant behavior clearly contradicting existing specifications. Concretely, the latest versions of Chromium and Firefox pass all the tests in BrowserAudit except for a few warnings, showing that the current set of test cases cannot identify relevant bugs in existing browsers, as opposed to our pipeline.

Other work on the automated detection of security bugs in browsers targeted specific mechanisms or vulnerabilities. For example, DiffCSP can detect bugs in CSP implementations [WNK⁺23], while other work investigated incoherencies in the implementation of SOP [SMWL10b, SNM17]. Automated testing has also been used to detect new cross-site leaks in browsers [RPS23] and to study the support of Web security mechanisms in mobile browsers [LLHN19]. All these proposals proved effective to identify new bugs, yet they are tailored to specific needs and do not leverage general security notions like the concept of Web security invariant adopted in this chapter.

Browser Instrumentation. VisibleV8 (VV8) [JK19] is a browser instrumentation framework, implemented as a set of patches for the Chromium browser, that allows for tracing JavaScript function calls and property access during navigation. The VV8 patches are designed to minimize the modified lines of code, so that they can be easily applied to updated browser versions. Browser instrumentation implemented as patches to the JavaScript engine, compared to in-band JavaScript instrumentation (e.g., prototype patching), has the unique advantage of being tamper-proof and impossible to detect by malicious scripts. However, it suffers from being tied to a specific browser implementation and requires additional manual work to be ported to new browser versions. For this reason, in this work we opted for browser extensions, which allow, via the WebExtension

API, cross-platform instrumentation that requires minimal to no effort to be applied to any extension-supporting browsers.

Similarly to VV8, JSgraph [LVLP18] is a patch to the Chromium source code that instruments the interface between Blink and V8, allowing for the recording of audit logs related to the execution of JavaScript in the browser. JSgraph aims to provide a detailed JS and DOM-related event log to aid in analyzing and reconstructing Web attacks. To this end, the tool includes a visualization component that shows the captured events in the form of a graph, highlighting causal relationship between events. JSgraph shares its main limitations with VV8, being tied to the specific implementation of the Chromium browser, requiring a substantial amount of manual work to keep up with the constantly evolving browser code.

Formalization of Web Invariants. In their 2010 paper, Akhawe et al. [ABL⁺10] presented a formal model of the Web platform for the Alloy analyzer and used it to verify the security of Web mechanisms such as CORS, the Origin header and HTML5 forms, discovering three new vulnerabilities. The authors encode in the model a set of security goals which are grouped into *security invariants* and *session integrity*. In particular, they emphasize the importance to identify clear Web security invariants that define the desired security goals of the Web platform, proposing the definition of 4 invariants. More recently, Veronese et al. proposed WebSpec [VFB⁺23], a framework for the analysis of Web security mechanisms composed of a model of the browser in the Coq proof assistant and a toolchain for automated model-checking against Web security invariants. In particular, the authors define 10 Web invariants concerning cookies, the CSP and the CORS, discovering two new attacks and presenting a formal proof of the correctness of their proposed mitigations. Although our approach for the definition of new Web invariants presents some similarities to both works, previous research focused on models of the browser and not on specific implementations. By leveraging the WPT test suite, we can (i) automatically check the actual browser implementation behavior (i.e., execution traces) against Web invariants; and (ii) sidestep the issue of requiring to manually update a browser model to match the updates of the Web platform. Additionally, compared to previous works, we are the first to support Mixed Content, modeling its specification by defining two new Web invariants.

3.7 Conclusion

This chapter presents a novel methodology for formally and automatically detecting security issues in browser implementations of client-side Web security mechanisms. Leveraging the WPT test suite, our framework collects browser execution traces and validates them using the Z3 theorem prover against Web security invariants. We formalized and encoded a total of 9 Web invariants and discovered violations within WPT, resulting in 10 unique attacks. We reported all our findings to the affected parties and kickstarted discussions with standardization bodies to address shortcomings at the specification level. This research positively answers our initial research question, showing that the proposed

3. WEB PLATFORM THREATS: AUTOMATED DETECTION OF WEB SECURITY ISSUES WITH WPT

automated approach can provide valuable guidance to browser vendors in identifying vulnerable Web components requiring immediate attention.

Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web

Abstract

Related-domain attackers control a sibling domain of their target Web application, e.g., as the result of a subdomain takeover. Despite their additional power over traditional Web attackers, related-domain attackers received only limited attention from the research community. In this paper we define and quantify for the first time the threats that related-domain attackers pose to Web application security. In particular, we first clarify the capabilities that related-domain attackers can acquire through different attack vectors, showing that different instances of the related-domain attacker concept are worth attention. We then study how these capabilities can be abused to compromise Web application security by focusing on different angles, including cookies, CSP, CORS, postMessage, and domain relaxation. By building on this framework, we report on a large-scale security measurement on the top 50k domains from the Tranco list that led to the discovery of vulnerabilities in 887 sites, where we quantified the threats posed by related-domain attackers to popular Web applications.

This chapter presents the results of a collaboration with Marco Squarcina, Mauro Tempesta, Stefano Calzavara and Matteo Maffei and has been published at the 30th USENIX Security Symposium in 2021 under the title “Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web” [STV⁺21]. Marco squarcina is responsible for the definition of the threat model, the design of the vulnerability scanner pipeline and the DNS scanner. Mauro tempesta is responsible for the Web analyzer component of the pipeline. I am responsible for the analysis of discontinued third-party services and contributed to the

analysis of deprovisioned cloud instances. Stefano Calzavara and Matteo Maffei were the general advisors and contributed with continuous feedback.

4.1 Introduction

The Web is the most complex distributed system in the world. Web security practitioners are well aware of this complexity, which is reflected in the threat modeling phase of most Web security analyses. When reasoning about Web security, one has to consider multiple angles. The *Web attacker* is the baseline attacker model that everyone is normally concerned about. A Web attacker operates a malicious website and mounts attacks by means of standard HTML and JavaScript, hence any site operator in the world might act as a Web attacker against any other service. High-profile sites are normally concerned about *network attackers* who have full control of the unencrypted HTTP traffic, e.g., because they operate a malicious access point. Both Web attackers and network attackers are well known to Web security experts, yet they do not capture the full spectrum of possible threats to Web application security.

In this chapter we are concerned about a less known attacker, referred to as *related-domain attacker* [BBC11]. A related-domain attacker is traditionally defined as a Web attacker with an extra twist, i.e., its malicious website is hosted on a sibling domain of the target Web application. For instance, when reasoning about the security of `www.example.com`, one might assume that a related-domain attacker controls `evil.example.com`. The privileged position of a related-domain attacker endows it, for instance, with the ability to compromise cookie confidentiality and integrity, because cookies can be shared between domains with a common ancestor, reflecting the assumption underlying the original Web design that related domains are under the control of the same entity. Since client authentication on the Web is mostly implemented on top of cookies, this represents a major security threat.

Despite their practical relevance, related-domain attackers received much less attention than Web attackers and network attackers in the Web security literature. We believe there are two plausible reasons for this. First, related-domain attackers might sound very specific to cookie security, i.e., for many security analyses they are no more powerful than traditional Web attackers, hence can be safely ignored. Moreover, related-domain attackers might appear far-fetched, because one might think that the owner of `example.com` would never grant control of `evil.example.com` to untrusted parties.

Our research starts from the observation that both previous arguments have become questionable, and this is the right time to take a second look at the threats posed by related-domain attackers, which are both relevant and realistic. A key observation to make is that a related-domain attacker shares the same *site* of the target Web application, i.e., sits on the same registrable domain. The notion of site has become more and more prominent for Web security over the years, going well beyond cookie confidentiality and integrity issues. For example, the Site Isolation mechanism of Chromium ensures that pages from different sites are always put into different processes, so as to offer better

security guarantees even in presence of bugs in the browser [RMO19]. Moreover, major browsers are now changing their behavior so that cookies are only attached to same-site requests by default, which further differentiates related-domain attackers from Web attackers. In the rest of the chapter, we discuss other (normally overlooked) examples where the privileged position of related-domain attackers may constitute a significant security threat. Finally, many recent research papers showed that *subdomain takeover* is a serious and widespread security risk [LHW16, BFH⁺18]. Large organizations owning a huge number of subdomains might suffer from incorrect configurations, which allow an attacker to make subdomains resolve to a malicious host. This problem also received attention from the general media [Osb17] and the industry [Bia15]. Though these studies proved that related-domain attackers are a realistic threat, they never quantified their impact on Web application security at scale.

Contributions

In this work, we perform the first scientific analysis of the dangers represented by related-domain attackers to Web application security. In particular:

1. We introduce a fine-grained definition of related-domain attacker that captures the capabilities granted to such attackers according to the position they operate and the associated Web security threats. In particular, we systematize the attack vectors that an attacker can exploit to gain control of a domain, and we present the attacks that can be launched from that privileged position, discussing the additional gain with respect to a traditional Web attacker (§4.3).
2. We implement a toolchain to evaluate the dangers that related-domain attackers can pose to Web application security. Our toolchain builds on top of an analysis module for subdomain takeover, which significantly improves over previous results [LHW16]. We use the output of this module to perform automated Web application security analyses along different angles, including cookies, CSP, CORS, postMessage, and domain relaxation (§4.4).
3. We report on experimental results established through our toolchain. In particular, we enumerate 26M subdomains of the top 50k registrable domains from the Tranco list and discover practically exploitable vulnerabilities in 887 domains, including major websites like `cnn.com`, `nih.gov`, `harvard.edu`, and `cisco.com`. We also study the security implications of 31 third-party service providers and dynamic DNS and present a novel subdomain hijacking technique that resulted in a bug bounty of \$1,000. Importantly, we quantify for the first time the impact of these vulnerabilities on Web application security, concluding that related-domain attackers have an additional gain compared to Web attackers that goes beyond well-studied issues on cookies (§4.5).

We have responsibly disclosed the identified vulnerabilities to the respective site operators. The results of the notification process are presented in §4.6.

Table 4.1: Main DNS record types.

| Record Type | Description |
|-------------|--|
| A | Returns the IPv4 address of a domain |
| AAAA | Returns the IPv6 address of a domain |
| CNAME | Maps an alias name to the canonical domain name |
| NS | Defines the authoritative DNS record for a domain |
| CAA | Specifies the allowed certificate authorities for a domain |

4.2 Background

DNS Resolution. DNS is a protocol that stands at the core of the Internet [Moc87]. It translates mnemonic domain names to IP addresses used by the underlying network layer to identify the associated resources. The translation process, called *DNS resolution*, is done transparently to applications. For instance, when a browser attempts to visit a fully qualified domain name (FQDN), such as `www.example.com`, the local resolver forwards the request to one of the DNS servers designated by the operating system. In case the DNS server has no information on the requested domain name, it initiates the recursive resolution from the root DNS server until the *authoritative* DNS server for the domain is reached, following the *subdomain* hierarchy of the DNS system. Eventually, the authoritative DNS server returns to the client a set of Resource Records (RRs) with the format: *name, TTL, class, type, data*. A list of relevant DNS record types is summarized in Table 4.1.

DNS also supports *wildcard* RRs with the label `*`, such as `*.example.com`. Wildcard RRs are not matched if an explicit RR is defined for the requested name. In general, wildcard RRs have a lower priority than standard RRs [Lew06]. For instance, given a wildcard A record `*.example.com` and an A record for `a.example.com`, requests to `b.example.com` and `c.b.example.com` are resolved by the wildcard, while requests to `a.example.com` are matched by the corresponding A record. Notice that `c.a.example.com` is not resolvable.

Public Suffix List. While DNS defines the hierarchical structure of domain names, the Public Suffix List (PSL) is a catalog of domain suffixes controlled by registrars [Moza]. In contrast to Top-Level Domains (TLDs) that are defined in the Root Zone Database [IAN], such as `.com`, `.org`, `.net`, the suffixes listed in the PSL are called *effective TLDs* (eTLDs) and define the boundary between names that can be registered by individuals and private names. A domain name having just one label at the left of a public suffix is commonly referred to as *registrable domain*, *eTLD+1*, or *apex domain*. Domains sharing the same eTLD+1 are said to belong to the same *site*.

Cookies are scoped based on the definition of site, i.e., subdomains of the same site can share cookies (*domain cookies*) by setting their Domain attribute to a common ancestor. This attribute can never be set to a member of the PSL: for instance, since `github.io` is in the PSL, `foo.github.io` is not allowed to set cookies for `github.io`. This means that there is no way to share cookies between different GitHub Pages hosted sites.

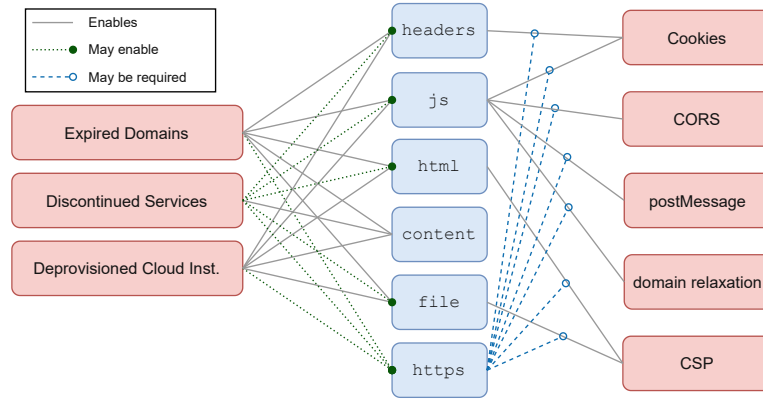


Figure 4.1: Summary of related-domain attacker instances for dangling DNS records.

4.3 The Related-Domain Attacker

We revise the threat model of the related-domain attacker in light of the directions that the Web has taken in recent years. In particular, we systematize for the first time the different attack vectors that can be exploited to escalate to a related-domain position. We also factorize the related-domain attacker into a set of capabilities and we express prerequisites of Web attacks in terms of them, as presented below and summarized in Figure 4.1 for the most common subdomain takeover vulnerabilities [LHW16]. This systematization allows for a quantification of the related-domain attacker problem, which we conduct in §4.5 by a large-scale measurement in the wild.

4.3.1 Threat Model

In its original definition, the related-domain attacker is a Web attacker who operates a malicious website that is hosted on a *related domain* of the target website [BBC11]. Two domains are *related* if they share a suffix that is not included in the PSL. For instance, consider the target site `example.com`: all its subdomains are related to the target, as well as being related to each other. Network attackers are traditionally considered out of scope, given that they could mount *person-in-the-middle* attacks via, e.g., ARP spoofing and DNS cache poisoning, which allow to easily control the IP address of any hostname accessed by the victim [CFST17].

Subdomain takeovers are often caused by DNS misconfigurations [LHW16, BFH⁺18], with consequences ranging from altering the content of a page to full host control. Additionally, organizations frequently assign a subdomain of their corporate domain to their users, who could maliciously take advantage of this implicit trust. Vulnerable Web applications can also be infiltrated to increase the privileges of attackers interested in exploiting their related domains.

As we elaborate in the following, the attack vector exploited to acquire a related-domain position is not a detail, but has an impact on the *capabilities* granted to the attacker.

Table 4.2: Capabilities of the related-domain attacker.

| Capability | Description |
|------------|---|
| headers | access and modify HTTP headers |
| js | arbitrary JavaScript code execution |
| html | alter the markup of the website with the exclusion of JavaScript |
| content | alter the textual content of the website with the exclusion of embed tags, frames and JavaScript code |
| file | host arbitrary files |
| https | operate a website under HTTPS with a valid certificate |

Note: `js` subsumes both `html` and `content`, since it is possible to edit the DOM by using JavaScript. Similarly, `html` subsumes `content`.

While full control of the host grants the attacker the ability to configure the Web server to host arbitrary content, other attack scenarios only grant more limited power. For example, exploiting a reflected XSS on a subdomain of a company poses several restrictions on the actions that can be undertaken by the attacker. This motivates the need for a new, fine-grained definition of related-domain attacker, which precisely characterizes its power based on the acquired capabilities. In §4.3.2, we map concrete attack vectors to the set of capabilities (see Table 4.2) that the attacker may acquire when escalating to a related-domain position. In §4.3.3, we link such capabilities to Web security threats, giving rise to a granular framework that defines different instances of the related-domain attacker.

4.3.2 Abusing Related Domains

We provide a comprehensive characterization of the attack vectors that can be exploited to acquire a related-domain position and identify the set of associated capabilities. While some of these attack vectors have been already analyzed in the literature in isolation (e.g., dangling DNS records [LHW16] and domain shadowing [Bia15, LLD⁺17]), it is the first time they are systematized to cover the possible abuses which enable escalation to a related-domain position. Furthermore, we introduce a novel attack vector that exploits DNS wildcards, and we point out concrete instances of roaming services, hosting providers, and dynamic DNS services which are vulnerable to the threats described in this work.

Dangling DNS Records

Dangling DNS records refer to records in the authoritative DNS servers of a domain that point to expired resources. These records should be purged right away after releasing the pointed resources. Unfortunately, this practice is often overlooked, resulting in dangling DNS records to persist indefinitely. Possible reasons include lack of communication between the person who releases the resource and the domain owner or when the pointed resource expires automatically after a certain period of time, passing unnoticed. A dangling DNS record is considered vulnerable if an unintended party can take control of the expired resource [LHW16].

Expired Domains. A DNS CNAME record maps a domain name (*alias*) to another one called canonical name. If the canonical name is expired, a third party can simply register the domain and serve arbitrary content under the alias domain. Attackers exploiting this vulnerability have full control of the host and generally can rely on all the capabilities listed in our framework. One exception is `https` in presence of a CAA DNS record [HBSHA19]: this record defines a list of Certificate Authorities (CAs) which are allowed to issue certificates for a given domain, possibly preventing attackers to rely on automated CAs like Let's Encrypt [ABC⁺19].

Discontinued Services. Third-party services are widely used to extend the functionalities of a website. Domain owners can integrate rich platforms by making them accessible under a subdomain of their organization, e.g., `blog.example.com` could show a blog hosted by WordPress and `shop.example.com` could be an e-shop run by Shopify. To map a (sub)domain to a service, an integrator typically has (i) to configure a DNS record for the (sub)domain, such as A/AAAA, CNAME or NS, to point to a server controlled by the service provider, and (ii) to claim the ownership of the (sub)domain in the account settings of the service. If the service provider does not verify the domain ownership explicitly, i.e., a DNS record pointing to the service is the only condition required to claim the ownership of a (sub)domain, an attacker could map to their account any unclaimed (sub)domain with a valid DNS record in place [LHW16].

In addition, we observe that dangling records can also occur due to the presence of DNS wildcard. Consider, for example, a site operator configuring a DNS wildcard such as `*.example.com` pointing to a service provider IP to enable multiple websites to be hosted under subdomains of `example.com`. An attacker could bind a subdomain of their choice, e.g., `evil.example.com`, to a new account on the service provider. Surprisingly, we discovered that some service providers do not verify the ownership of a subdomain even if the parent domain has been already mapped to an existing account. In practice, this allows an attacker to claim `evil.proj.example.com` also in presence of a legitimate binding for `proj.example.com`. Even worse, we found that some service providers perform an automatic redirection of the `www`-prefixed subdomains to their parent domains without preventing the `www` subdomain from being associated to a different account. We report on this novel attack in §4.5.1.

Attackers' capabilities vary depending on the platform and range from altering the content of a single page to full host control. We refer to §4.5 for the result of a thorough security investigation conducted on 31 service providers.

Deprovisioned Cloud Instances. The ephemeral nature of resources allocated in Infrastructure as a Service (IaaS) environments is known to facilitate the spread of dangling DNS records. DNS records pointing to available IP addresses in the cloud can be abused by a determined attacker who rapidly allocates IP addresses in order to control the target of the dangling DNS record [LHW16, BFH⁺18]. Similarly to expired domains,

the presence of a CAA DNS record in a parent domain could hinder the capability of obtaining a valid TLS certificate.

Corporate Networks and Roaming Services

Large organizations often assign fully qualified domain names (FQDNs) to devices in their network. This practice allows to statically reference resources in the network, irrespective of the assignment of IP addresses that may change over time. Although hosts might be inaccessible from outside of the organization network, internal users are put in a related-domain attacker position with full capabilities, excluding `https` that depends on the network configuration of the organization.

Institutions providing roaming services are similarly prone to the same issue. This is the case of eduroam, a popular international education roaming service that enables students and researchers to have a network connection provided by any of the participating institutions. As a novel insight, we discovered that system integrators at some local institutions are assigning eduroam users a subdomain of the main institution, such as `ip1-2-3-4.eduroam.example.com`, where `1.2.3.4` is a placeholder for the public IP assigned to the user connected to the eduroam network. This practice ultimately promotes any eduroam user to a related-domain attacker with full control of the host that is pointed by the DNS record. Firewall restrictions might hinder complete visibility on the Internet of the personal device of the user. Still, users' devices might be accessible within the institution network.

Hosting Providers and Dynamic DNS Services

Many service providers allow users to create websites under a specific subdomain, e.g., `<username>.github.io` on GitHub. Subdomains hosting user-supplied content are not related to each other if the parent domain is included in the PSL, as in the case of `github.io`. Unfortunately, several service providers that we reviewed did not include their domains in the PSL, turning any of their users into a related-domain attacker for all the websites hosted on the same platform.

A similar consideration applies to dynamic DNS providers. The race to offer a huge variety of domains under which users can create their custom subdomains, made it unfeasible for certain providers to maintain a list of entries in the PSL. The FreeDNS service [Fre20] pictures well the problem, with 52,443 offered domains and a declared user base of 3,448,806 active users as of October 2020, who are in a related-domain attacker position to all the subdomains and domains of the network, since none of them has been added to the PSL.

While in the case of hosting and service providers, the capabilities granted to the attacker largely depend on the specific service (see §4.5.1 for more details), a dynamic DNS service allows users to point a DNS record to a host they fully control, capturing all the capabilities discussed in Table 4.2.

Compromised Hosts/Websites

Aside from scenarios in which attackers gain control of a resource that is either abandoned or explicitly assigned to them, another way to obtain a related-domain attacker position is the exploitation of vulnerable hosts and websites. Intuitively, attackers achieving code execution on the vulnerable application have capabilities ranging from serving arbitrary content to full host control. If the exploited vulnerability is an XSS, attackers could take advantage of the ability to execute JavaScript code from a privileged position to escalate the attack against a more sensitive website.

Furthermore, attackers have been found employing a technique called *domain shadowing* [Bia15, LLD⁺17] to illicitly access the DNS control panel of active domains to distribute malware from arbitrary subdomains. Alowaisheq et al. recently discovered that stale NS records [ATW⁺20] could also be abused by attackers to take control of the DNS zone of a domain to create arbitrary DNS records. Controlling the DNS of a domain is the highest privileged setting for a related-domain attackers, since they can point subdomains to hosts they fully control and reliably obtain TLS certificates.

4.3.3 Web Threats

We identify for the first time a comprehensive list of Web security threats posed by related-domain attackers, discussing in particular the scenarios where a related-domain attacker might have an advantage over traditional Web attackers. While there exists ample literature on threats to cookies confidentiality and integrity posed by related-domain attackers [ZJL⁺15, CRB18], in this work we focus on a complete account of how related-domain attackers affect Web application security by exploring less-studied mechanisms.

Inherent Threats

Related-domain attackers sit on the same site of their target Web application. This is weaker than sharing the same *origin* of the target, which is the traditional Web security boundary, yet it suffices to abuse the trust put by browser vendors and end users on same-site content. We discuss examples below.

Trust of End Users. End users might trust subdomains of sites they are familiar with more than arbitrary external sites. For instance, attackers could exploit the residual trust associated with the subdomain's prior use [LWN⁺16] or deceive users into inserting their passwords provided by a password manager [Wal]. This is particularly dangerous on some mobile browsers, which display only the rightmost part of the domain due to the smaller display size, hence a long subdomain might erroneously look like the main site. Attackers could similarly abuse the trust inherited from the apex domain to use compromised subdomains for the distribution of malware or other types of dangerous content [LLD⁺17].

Site Isolation. Site Isolation is a browser architecture first proposed and implemented by the Google Chrome browser, which treats different sites as separate security principals requiring dedicated rendering processes [RMO19]. These processes can access sensitive data for a single site only, which mitigates the leakage of cross-origin data via memory disclosure and renderer exploits, including attacks based on Spectre [KHF⁺19, RJ21]. As acknowledged in the original Site Isolation paper [RMO19], “cross-origin attacks within a site are not mitigated”, hence related-domain attackers can void the benefits of this security architecture.

Same Site Request Forgery. The introduction of *same-site cookies* [WW20] and the recent enforcement of this security feature by default on major browsers [The20, Con20] received high praise as an effective countermeasure against CSRF [Hel17]. In the absence of other defenses [BJM08a], the restrictions introduced by same-site cookies are voided by a related-domain attacker who can mount a *same-site* request forgery attack just by including an HTML element pointing to the target website in one of their Web pages.

Cookie Confidentiality and Integrity

Cookies can be issued with the Domain attribute set to an ancestor of the domain setting them, so as to share them with all its subdomains. For example, `good.foo.com` can issue a cookie with the Domain attribute set to `foo.com`, which is sent to both `good.foo.com` and `evil.foo.com`. Hence, related-domain attackers can trivially break *cookie confidentiality* and abuse of stolen cookies [ZJL⁺15], e.g., to perform session hijacking. The Domain attribute poses risks to *cookie integrity* too: `evil.foo.com` can set cookies for `good.foo.com`, which can be abused to mount attacks like session fixation. Note that the integrity of host-only cookies is at harm too, because a related-domain attacker can mount *cookie shadowing*, i.e., set a domain cookie with the same name of a host-only cookie to confuse the Web server [ZJL⁺15].

Site operators can defend against such threats by careful cookie management. For example, they can implement (part of) the session management logic on top of host-only cookies, which are not disclosed to related-domain attackers. Moreover, they can use the `__Host-` prefix to ensure that security-sensitive cookies are set as host-only, thus ensuring their integrity against related-domain attackers.

Capabilities. The capabilities required by a related-domain attacker to break the confidentiality of a domain cookie depend on the flags enabled for it: if the cookie is `HttpOnly`, it cannot be exfiltrated via JavaScript and the `headers` capability is needed to sniff it; otherwise, just one between `headers` and `js` suffices. If the `Secure` flag is enabled, the cookie is sent only over HTTPS, hence the `https` capability is also required. As to integrity, all cookies lacking the `__Host-` prefix have low integrity against a related-domain attacker with the `headers` or `js` capabilities, since they are affected by cookie shadowing. There is one exception: cookies using the `__Secure-` prefix have

low integrity only against related-domain attackers which additionally have the `https` capability, since these cookies can only be set over HTTPS.

Bypassing CSP

Content Security Policy (CSP) is a client-side defense mechanism originally designed to mitigate the dangers of content injection and later extended to account for different threats, e.g., click-jacking. CSP implements a whitelisting approach to Web application security, whereby the browser behavior on CSP-protected Web pages is restrained by binding *directives* to sets of *source expressions*, i.e., a sort of regular expressions designed to express sets of origins in a compact way. To exemplify, consider the following CSP:

```
script-src foo.com *.bar.com;
frame-ancestors *.bar.com;
default-src https;
```

This policy contains three directives, `script-src`, `frame-ancestors` and `default-src`, each bound to a set of source expressions like `foo.com` and `*.bar.com`. It allows the protected page to: (i) include scripts from `foo.com` and any subdomain of `bar.com`; (ii) be included in frames opened on pages hosted on any subdomain of `bar.com`; (iii) include any content other than scripts over HTTPS connections with any host.

Since the syntax of source expressions naturally supports the whitelisting of any subdomain of a given parent, related-domain attackers represent a major threat against the security of CSP. For example, if an attacker could get control of `vuln.bar.com`, then they would be able to bypass most of the protection put in place by the CSP above. In particular, the attacker would be able to exploit a content injection vulnerability on the CSP-protected page to load and execute arbitrary scripts from `vuln.bar.com`, thus voiding XSS mitigation. Moreover, the attacker could frame the CSP-protected page on `vuln.bar.com` to perform click-jacking attacks. To avoid these threats, site operators should carefully vet the subdomains included in their CSP whitelists.

Capabilities. A related-domain attacker requires the capability to upload arbitrary files on the website under its control to void the protection offered by CSP against content inclusion vulnerabilities, with the only notable exception of frame inclusion which requires only the `html` capability. For active contents [W3C23], i.e., those that may have access to the DOM of the page, the attacker also needs the `https` capability if the target page is hosted over HTTPS. Regarding click-jacking protection, attackers only requires the `html` capability to include the target website on a page under their control.

Abusing CORS

Cross-Origin Resource Sharing (CORS) is the standard approach to relax the restrictions enforced by SOP on cross-origin communications, i.e., preventing JavaScript

from reading the content of responses to cross-origin requests. Consider a service at `https://www.example.com`, which needs to fetch sensitive data from `api.example.com` via JavaScript: to enable CORS, `https://api.example.com` can inspect the `Origin` header of incoming requests to detect if they come from `https://www.example.com` and, in such a case, set a CORS header `Access-Control-Allow-Origin` with the value `https://www.example.com` in the response. As an additional layer of protection, the server must also set the `Access-Control-Allow-Credentials` header to `true` if the request includes credentials, e.g., cookies, since the associated response is more likely to include sensitive content.

Related-domain attackers can abuse CORS to bypass the security restrictions put in place by SOP when the aforementioned server-side authorization checks are too relaxed, i.e., read access is granted to arbitrary subdomains. For example, if `https://api.example.com` was willing to grant cross-origin access to any subdomain of `example.com` besides `www.example.com`, a related-domain attacker could get unconstrained access to its data. To avoid these threats, site operators should be careful in the security policy implemented upon inspection of the `Origin` header, e.g., restricting access just to a few highly trusted subdomains.

Capabilities. To exploit CORS misconfigurations, an attacker needs the `js` capability to issue requests via JavaScript APIs like `fetch` and access the content of the response. The `https` capability may be required depending on the CORS policy deployed by the site operator.

Abusing `postMessage`

The `postMessage` API supports cross-origin communication across windows (e.g., between frames or between a page and the popup opened by it). The sender can invoke the `postMessage` method of the target window to transmit a message, possibly restricting the origin of the receiver. The receiver, in turn, can use event handlers to listen for the message event and process incoming messages.

Despite its apparent simplicity, the `postMessage` API should be used with care, as shown by prior research [SS13, SS20]. In particular, when sending confidential data, one should always specify the origin of the intended receiver in the `postMessage` invocation. When receiving data, instead, one should check the origin of the sender (via the `origin` property of the received message) and appropriately sanitize the content of the message before processing it.

Related-domain attackers can undermine Web application security when site operators put additional trust in subdomains. In particular, related-domain attackers can try to abuse their position to void the aforementioned origin checks and communicate with inattentive receivers that might process messages in an unsafe way, e.g., messages are provided as input to `eval` or stored in a cookie, opening the way to session hijacking

attacks. Site operators can defend against such attacks by carefully vetting authorized subdomains for communication between windows.

Capabilities. An attacker requires scripting capabilities (`js`) to open a new tab containing the vulnerable page and communicate with it via the `postMessage` API. Similarly to CORS, `https` may be needed depending on the origin checking performed by the receiver.

Abusing Domain Relaxation

Domain relaxation is the legacy way to implement communication between windows whose domains share a common ancestor. Assume that a page at `a.example.com` opens a page at `b.example.com` inside a frame. Besides using the `postMessage` API as explained, the two frames can communicate by relaxing their `document.domain` property to a common ancestor. In this case, both frames can set such property to `example.com`, thus moving into a same-origin position.¹ After that, SOP does not enforce any isolation between the two frames, which can communicate by writing on each other's DOM. Note that `example.com` must explicitly set the `document.domain` property to `example.com` if it is willing to engage in the domain relaxation mechanism, although this is apparently a no-op.

Domain relaxation can be abused by related-domain attackers, who can look for pages which are willing to engage in such dangerous communication mechanism and abuse it. In particular, when the attacker moves into a same-origin position, SOP does not provide any protection anymore, which voids any confidentiality and integrity guarantee. Websites that are willing to communicate with a selected list of related domains should refrain from using this mechanism – which is deemed as insecure – and should implement cross-origin communication on top of the `postMessage` API.

Capabilities. Besides the `js` capability needed to perform the relaxation and access the DOM of the target page, attackers need to setup their attack page on the same protocol of the target, hence the `https` capability may also be required.

4.4 Analysis Methodology

We performed a large-scale vulnerability assessment to measure the pervasiveness of the threats reported in this work, first by identifying subdomains of prominent websites that can be abused by a related-domain attacker exploiting dangling DNS records, and second by evaluating the security implications on Web applications hosted on related domains of the vulnerable websites. Our methodology is based on the pipeline summarized in Figure 4.2 and further described in this section.

¹We assume here that the two frames share the same protocol and port.

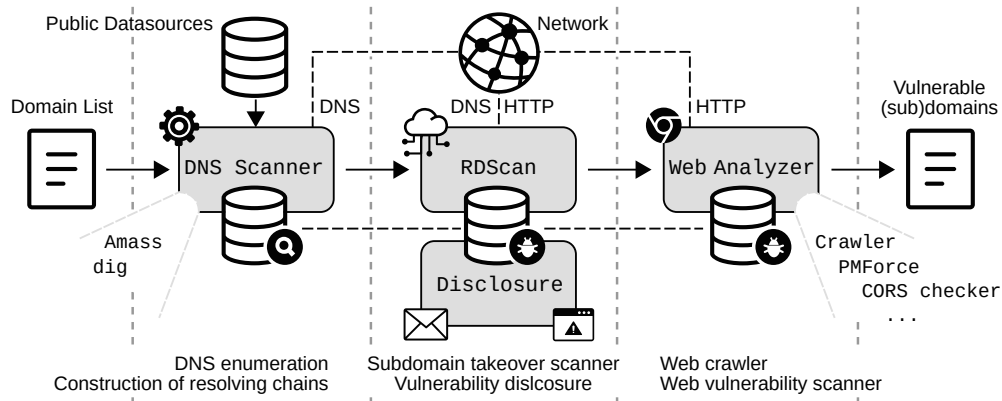


Figure 4.2: Vulnerability scanning pipeline.

4.4.1 DNS Data Collection

We enumerated the subdomains of the top 50k domains in the Tranco list [PGT⁺19] from March 2020.² The enumeration phase was based on amass [OWA20], a state of the art information gathering tool backed by the OWASP project. The tool supports several techniques to maximize the chances of discovering subdomains of a target. In our configuration, we extracted subdomains using the following approaches: (i) fetch data from publicly available sources, such as Censys [Cen20], certificate transparency logs [Sec20], search engines, etc.; (ii) attempt DNS zone transfer to obtain the complete list of RRs defined for a certain DNS zone; (iii) inspect fields of TLS certificates, e.g., Subject Alternative Name and Common Name. To speed up the enumeration phase and lower the number of network requests, we avoided bruteforcing DNS resolvers against domain name wordlists. Similarly, we explicitly disabled the resolution of subdomain alterations.

We modified amass to compute the DNS *resolving chains* of all the domains obtained in the previous step. Similarly to [LHW16], we define a resolving chain as a list of DNS RRs in which each element is the target of the previous one, starting from a DNS record of type A/AAAA, CNAME or NS. We ignore MX records because we focus on Web attacks in this study. For CNAME and NS records, we recursively perform a DNS resolution until an A/AAAA RR is detected. Unterminated DNS resolving chains can occur in presence of a record pointing to an unresolvable resource or due to the abrupt termination of amass after reaching the execution timeout limit of 5 minutes. To ensure the correctness of the results, we recompute unterminated DNS resolving chains using the dig utility.

Starting from the set of 50k domains in the Tranco list, our framework identified 26 million valid subdomains. In a previous study, Liu et al. [LHW16] used a relatively small wordlist of 20,000 entries to find possible subdomains of the Alexa top 10k list, 2,700 .edu domains, and 1,700 .gov domains. Compared to their work, our domain selection is penalized given that we do not restrict to specific TLD zones. For instance, .edu domains

²<https://tranco-list.eu/list/ZKYG/1000000>

typically have a high number of subdomains in contrast to other categories (see §4.5.1). Nevertheless, our results outperform the findings of Liu et al. by discovering on average 13 times more subdomains.

4.4.2 RDScan

After populating a database with the DNS records of the discovered subdomains, the framework detects dangling records and verifies that all the preconditions to mount a subdomain takeover attack are met. By doing so, false positives are minimized in the analysis. This component, that we call *RDScan*, has three different modules that test for the presence of the vulnerable scenarios described in §4.3.2.

Expired Domains. The detection of expired domains is performed according to the following procedure: given a resolving chain that begins with a CNAME record, our tool checks if it points to an unresolvable resource and extracts the eTLD+1 of the canonical name at the end of the chain, that we call *apex* for brevity. Then, if the *whois* command on the apex domain does not return any match, RDScan queries GoDaddy to detect if the domain can be purchased. In this case, we consider the domain of the resolving chain, i.e., the alias of the first record of the chain, as vulnerable. Notice that we only tested domains that can be registered without special requirements, i.e., we did not consider .edu domains and other specific eTLDs not offered by the registrar.

Discontinued Services. The process of finding discontinued services is summarized in Algorithm 1. RDScan traverses each resolving chain to identify whether it points to one of the services supported by our framework. This step is implemented according to the documentation provided by individual services, and typically relies on checking for the presence of (i) an A record resolving to a specific IP address, (ii) the canonical name of a CNAME record matching a given host, or (iii) the existence of a NS record pointing to the DNS server of a service. (Sub)domains mapped to services are then checked to verify if the bindings between user accounts and (sub)domains are in place. For the majority of the services considered in this study, a simple HTTP request suffices to expose the lack of a correct association of a (sub)domain. Other services require active probing to determine whether a domain can be associated to a fresh test account that we created. This has been done using the automated browser testing library puppeteer with Chromium [Pup20]. RDScan also performs the detection of DNS wildcards that might be abused as described in §4.3.2. A DNS wildcard for a domain such as `test.example.com` can be easily detected by attempting to resolve a CNAME or A DNS record for `<nonce>.test.example.com`, where *nonce* refers to a random string that is unlikely to match an entry in the DNS zone of the target domain.

Deprovisioned Cloud Instances. The detection of potentially deprovisioned cloud instances has been performed similarly to the probabilistic approach adopted by [LHW16, BFH⁺18]. We did not create any virtual machine or registered any service at cloud

Algorithm 1 Detection of Discontinued Services**Input:** Set of DNS resolving chains RC , set of supported services S **Output:** Set of vulnerable subdomains V_s

```

1: procedure DISCONTINUED_SERVICES( $RC, S$ )
2:    $V_s \leftarrow \emptyset$ 
3:   for each  $chain \in RC$  do
4:     for each  $service \in S$  do
5:        $\triangleright$  Check if a record in the chain points to the service
6:       if  $chain$  points to  $service$  then
7:          $d \leftarrow target\_domain(chain)$ 
8:         if  $d$  is unclaimed at  $service$  then
9:            $V_s \leftarrow V_s \cup \{d\}$ 
10:         $\triangleright$  Detect wildcard if the service allows a subdomain of a
11:         $\triangleright$  claimed domain to be mapped to a different account
12:        else if  $service$  vulnerable to wildcard issue then
13:           $r \leftarrow generate\_nonce()$ 
14:           $rd\_chains \leftarrow compute\_resolving\_chains(r.d)$ 
15:          for each  $rd\_chain \in rd\_chains$  do
16:            if  $rd\_chain$  points to  $service$  then
17:               $V_s \leftarrow V_s \cup \{r.d\}$ 

```

providers in this process. Instead, we collected the set of IP ranges of 6 major providers: Amazon AWS, Google Cloud Platform, Microsoft Azure, Hetzner Cloud, Linode, and OVHcloud. We tested each (sub)domain in our dataset to check whether the pointed IP was included in any of the cloud IP ranges. In case the IP falls within the address range of a cloud provider, we make sure that it does not point to a reserved resource such as a proxy or a load balancer. As the last step, we perform a liveness probe to determine if the IP is in use. This is done by executing a ping to the IP: if no answer is received, we use a publicly available dataset [Rap20] comprising a scan of the full IPv4 range on 148 ports (128 TCP, 20 UDP). If no open ports for the given IP are found, we deem the resource as potentially deprovisioned.

4.4.3 Web Analyzer

Our Web security analysis aims at quantifying the number of domains hosting Web applications that can be exploited by taking over the vulnerable domains discovered by RDScan. In particular, for every apex domain with at least one vulnerable subdomain, we selected from the CommonCrawl dataset [Com20] the list of 200 most popular related domains according to the Pagerank score [BP98]. From the homepage of these domains, we extracted the same-origin links that appear in the HTML code. For each related domain, we considered the homepage and up to 5 of these URLs as the target of our Web analysis, and we accessed these links using the Chromium browser automated by puppeteer. In the following, we present the data collection process and the security analyses we have conducted to identify the threats discussed in §4.3.3. We postpone the summary of the results to §4.5.

Analysis of Cookies

We used the puppeteer API to collect cookies set via HTTP headers and JavaScript. Our goal is to identify cookies affected by confidentiality or integrity issues. In particular, we flag a cookie as affected by confidentiality issues if, among the related domains vulnerable to takeover, there exists a domain d such that:

- d is a subdomain of the Domain attribute of the cookie;
- by taking over d , the attacker has acquired the capabilities required to leak the cookie.

We mark a cookie as affected by integrity issues if:

- the name of the cookie does not start with `__Host-`;
- we identified a vulnerable domain that grants the capabilities required to set the cookie.

We also rely on a heuristic proposed by Bugliesi et al. [BCFK15] to statically identify potential (pre-)session cookies, i.e., cookies that may be relevant for the management of user sessions.

The capabilities required to perform these attacks depend on the security flags assigned to the cookie and the usage of cookie prefixes (see §4.3.3). For instance, to compromise integrity either the capability `js` or `headers` is required and, if the prefix `__Secure-` is used, `https` is also necessary.

Analysis of CSP policies

For this analysis, we implemented a CSP evaluator according to the draft of the latest CSP version [W3C18], which is currently supported by all major browsers. This is not a straightforward task, due to the rich expressiveness of the policy and various aspects that have been introduced into the specification for compatibility purposes across different CSP versions, e.g., for scripts and styles, the `'unsafe-inline'` keyword, which whitelists arbitrary inline contents in a page, is discarded when hashes or nonces are also specified.

In our analysis, we focus on the protection offered against click-jacking and the inclusion of active contents [W3C23], i.e., resources that have access to (part of) the DOM of the embedding page. This class of contents includes scripts, stylesheets, objects, and frames.

For each threat considered in our analysis, we first check if the policy is unsafe with respect to any Web attacker. This is the case for policies that allow the inclusion of contents from any host (or framing by any host, when focusing on click-jacking protection). For scripts and styles, the policy is also deemed unsafe if arbitrary inline contents are whitelisted. If the policy is considered safe, we classify it as exploitable by a related domain if one of the vulnerable domains detected by RDScan is whitelisted and the

attacker acquires the relevant capabilities to perform the attack, which vary depending on the threat under analysis (see §4.3.3). For instance, script injection requires the `file` capability, given that attackers need to host the malicious script on a subdomain they control. Moreover, if the page to attack is served over HTTPS, the `https` capability is required due to the restrictions imposed by browsers on mixed content [W3C23].

Analysis of CORS

To evaluate the security of the CORS policy implemented by a website, we perform multiple requests with different `Origin` values and inspect the HTTP headers in the response to understand whether CORS has been enabled by the server.

Inspired by the classes of CORS misconfigurations identified in [CJD⁺18], we test 3 different random origins with the following characteristics: (i) the domain is a related domain of the target URL; (ii) the domain starts with the registrable domain of the target URL; (iii) the domain ends with the registrable domain of the target URL. While the first test verifies whether CORS is enabled for a related domain, the other two detect common server-side validation mistakes. Such errors include the search of the registrable domain as a substring or a suffix of the `Origin` header value, which results in having, e.g., `www.example.com` whitelisting not only `a.example.com` but also `atkexample.com`. For each test, we check if the `Access-Control-Allow-Origin` header is present in the response and if its value is either `*` or that of the `Origin` header contained in the request. We also control if the `Access-Control-Allow-Credentials` header is present and set to `true` (when `Access-Control-Allow-Origin` differs from `*`) to identify the cases in which requests with credentials are allowed.

We report a CORS deployment as vulnerable to Web attackers if either the second or the third test succeeds. Instead, a page is exploitable exclusively by a related-domain attacker if only the first test succeeds and, among the vulnerable related domains discovered by RDSscan, one grants the `js` capability to the attacker. Since in our tests we use the same protocol of the page under analysis in the `Origin` header, we conservatively require the `https` capability when HTTPS is used.

Analysis of `postMessage` Handlers

PMForce [SS20] is an automated in-browser framework for the analysis of `postMessage` event handlers. It combines selective force execution and taint tracking to extract the constraints on the message contents (e.g., presence of a certain string in the message) that lead to execution traces in which the message enters a dangerous sink that allows for code execution (e.g., `eval`) or the alteration of the browser state (e.g., `document.cookie`). A message satisfying the extracted constraints is generated using the Z3 solver and the handler under analysis is invoked with the message as a parameter to ensure that the exploit is successfully executed.

We integrated PMForce in our pipeline and modified it to generate, for each handler, multiple exploit messages with the same contents but a different `origin` property, e.g.,

a related-domain origin and a randomly-generated cross-site origin. We consider a page vulnerable to any Web attacker if any of its handlers is exploitable from a cross-site position. Instead, we consider a page exploitable by a related-domain attacker if its handlers can be exploited only from a related-domain position and one of the vulnerable domains discovered by RDScan grants the `js` capability to the attacker, which is required to open a tab and send messages to it. If the handlers whitelist only HTTPS origins, then the capability `https` is also required to mount the attack.

Analysis of Domain Relaxation

As a first step, the analyzer detects whether the property `document.domain` is set after the page is loaded. This task is straightforward except for the case in which the page sets the property to its original value (see §4.3.3) since this cannot be detected just by reading the value of `document.domain`. To identify this particular case, we leverage puppeteer APIs to:

- inject a frame from a (randomly generated) subdomain of the page under analysis;
- intercept the outgoing network request and provide as response a page with a script that performs domain relaxation and tries to access the parent frame, which succeeds only if the parent has set `document.domain`.

The relaxation mechanism is exploitable by a related-domain attacker if RDScan discovered a vulnerable subdomain (which is a subdomain of the value of `document.domain`) that grants the `js` capability to the attacker. If the webpage is hosted over HTTPS, the `https` capability is also required.

4.4.4 Heuristics and False Positives

Our methodology is based on testing sufficient preconditions to execute the reported attacks, thus minimizing false positives. Nevertheless, the scanning pipeline makes use of two heuristics in the RDScan and Web analyzer modules to, respectively, detect potentially deprovisioned cloud instances and label security-sensitive cookies; moreover, we identify a potential TOCTOU issue between the two modules of the analysis pipeline. We discuss below why this has only a marginal effect on the overall results of the analysis.

RDScan. We developed automated procedures to test sufficient preconditions for a takeover. Expired domains are trivially verified by checking if the target domain can be purchased. For discontinued services, we created personal testing accounts on each service considered in the analysis and used these accounts to probe the mapping between the target subdomain and the service. If we detect all necessary conditions to associate the subdomain to our account, we deem it as vulnerable. We manually vetted these conditions against our own domain. Due to ethical concerns, we did not mount attacks against real websites, but we reviewed all the occurrences of subdomain takeover vulnerabilities before disclosing them to the affected sites and found no false positives in the results (see

Section 4.6). The detection of subdomains pointing to deprovisioned cloud instances relies instead on a heuristic which might introduce false positives, as discussed in §4.4.2. We performed this investigation to capture the magnitude of the problem, but we excluded the results on deprovisioned cloud instances from the pipeline to avoid false positives in the Web analyzer. To avoid misunderstandings in this chapter, we refer to domains matching our heuristic as *potentially vulnerable*.

Web Analyzer. The Web vulnerabilities discovered by this module have been identified via dynamic testing and analysis of the data collected by the crawler. We manually verified samples of each detected vulnerability to ensure the correctness of the results and confirmed the absence of false positives. The usage of heuristics is limited to the labeling of cookies which likely contain session identifiers and are thus particularly interesting from a security standpoint; this approach has been proved reasonably accurate in prior work [BCFK15].

Interplay between the modules. The modules of the pipeline described in Figure 4.2 have been executed in sequence at different points in time. The DNS enumeration phase terminated in June 2020, while RDScan ran during the first half of July 2020. The severity of the discovered issues motivated us to immediately report them to the affected parties. Therefore, we launched a large-scale vulnerability disclosure campaign in the second half of the month. We executed the Web scanner right after that. Having the DNS data collection running first, RDScan might have missed new subdomains that were issued after the completion of the DNS enumeration. This leads to a possible *underestimation* of the threats in the wild concerning unresolvable domains and expired services. On the other hand, subdomain takeover vulnerabilities might have been fixed prior to the Web security analysis. We performed a second run of RDScan 6 months later to verify the fix rate of notified parties. Surprisingly, we discovered that, as of January 2021, 85% of the subdomains that we tested are still affected by leftover subdomain takeover vulnerabilities, confirming that the early remediation of the reported vulnerabilities had a marginal effect on the Web analysis. We provide more details on our large-scale disclosure campaign in Section 4.6.

4.5 Security Evaluation

We report on the results of our security evaluation on the top 50k domains from the Tranco list. We quantify the vulnerabilities that allow an attacker to be in a related-domain position, and we provide a characterization of the affected websites. Then, we delve into the security of 31 service providers by discussing common pitfalls and the capabilities that could be abused by an attacker. Finally, we present the outcome of our Web analysis, and we identify practical vulnerabilities by intersecting the capabilities on vulnerable domains with the threats found on Web applications hosted on their related domains. Table 4.3 provides a breakdown of the results by combining attack vectors and Web threats: the values reported in the cells represent the number of vulnerable

Table 4.3: Breakdown of the results in terms of affected domains/sites.

| Attack Vector | Takeover | | Web mechanisms exploitable <i>exclusively</i> by related-domain attackers | | | | | | Relaxation | |
|-----------------------|--------------|------------|---|----------------|----------------|---------------|------------------|---------------|--------------|--------------|
| | Domains | Sites | Cookies Domains | Sites | CSP Domains | Sites | CORS Domains | Sites | Domains | Sites |
| Expired Domains | 260 | 201 | 5,394/5,394 | 195/195 | 35/141 | 13/28 | 35/317 | 16/107 | 9/11 | 6/8 |
| Discontinued Services | 1,260 | 699 | 18,798/19,020 | 662/674 | 104/294 | 32/75 | 196/1,980 | 37/392 | 49/88 | 24/55 |
| ↳ WordPress | 466 | 320 | 13,803/13,803 | 312/312 | 43/168 | 23/52 | 164/1,221 | 21/186 | 30/49 | 14/28 |
| ↳ Shopify | 326 | 254 | 2,638/2,638 | 244/244 | 32/66 | 5/12 | 26/459 | 11/153 | 7/19 | 5/15 |
| ↳ Tumblr | 310 | 24 | 404/404 | 23/23 | 1/2 | 1/2 | 5/29 | 2/12 | 1/2 | 1/2 |
| ↳ GitHub | 42 | 25 | 899/899 | 24/24 | 22/49 | 1/5 | 2/116 | 2/18 | 2/3 | 2/3 |
| ↳ Webflow | 24 | 20 | 601/601 | 18/18 | 0/0 | 0/0 | 2/122 | 2/14 | 1/3 | 1/3 |
| ↳ Ngrok | 22 | 13 | 250/250 | 13/13 | 7/9 | 2/2 | 0/17 | 0/5 | 8/11 | 1/3 |
| ↳ Helpscout | 18 | 17 | 425/425 | 16/16 | 1/4 | 1/3 | 0/28 | 0/6 | 0/2 | 0/2 |
| ↳ Others | 52 | 37 | 464/724 | 22/35 | 1/7 | 1/3 | 0/25 | 0/8 | 9/10 | 2/3 |
| Total | 1,520 | 887 | 23,178/23,400 | 845/857 | 139/428 | 45/100 | 224/2,254 | 41/488 | 57/97 | 29/61 |

Note: Deployment of CSP only considers policies that are not trivially exploitable by a Web attacker (§4.4.3) and whitelist one or more related domains. CORS policies are only exposed to requests coming from whitelisted origins [CJD⁺18]: for the deployment we report the count of domains/sites vulnerable either to Web attackers or related-domain attackers that were discovered during dynamic testing. `postMessage` is omitted since related-domain attackers have no gain compared to Web attackers.

domains/sites compared to those deploying the corresponding Web mechanism. We discuss these results in the following.

4.5.1 Attack Vectors and Capabilities

RDScan identified 1,520 subdomains exposed to a takeover vulnerability, distributed among 887 domains from the top 50k of the Tranco list. Most of the vulnerabilities are caused by discontinued third-party services (83%), with expired domains being responsible for the remaining 17%. The analysis of deprovisioned cloud instances discovered 13,532 potentially vulnerable domains, confirming the prevalence of this threat as reported in previous work [LHW16].

Characterization of Vulnerable Domains

As expected, the likelihood of a domain to be vulnerable is directly related to the breadth of its attack surface, i.e., the number of subdomains we found. Figure 4.3a pictures well this correlation, showing that around 15% of the domains with more than 50,000 subdomains are vulnerable. Figure 4.3b outlines the distribution of vulnerable domains depending on the rank of each site in the Tranco list. Sites in top positions are more likely to have a vulnerable subdomain than those with a lower rank.

The analyzed websites have been further partitioned into categories in Figure 4.3c. Special care has to be taken when considering dynamic DNS: the 49 domains listed in this category are those used by dynamic DNS services, such as `ddns.net`, `noip.com`, `afraid.org`. RDScan identified vulnerable subdomains belonging to 8 domains, but 4 of them were listed in the PSL. We excluded these domains from our analysis, given that taking control of one of their subdomains would not put the attacker in a related-domain position with respect to the parent domain. The same principle has been adopted when

4. CAN I TAKE YOUR SUBDOMAIN? EXPLORING SAME-SITE ATTACKS IN THE MODERN WEB

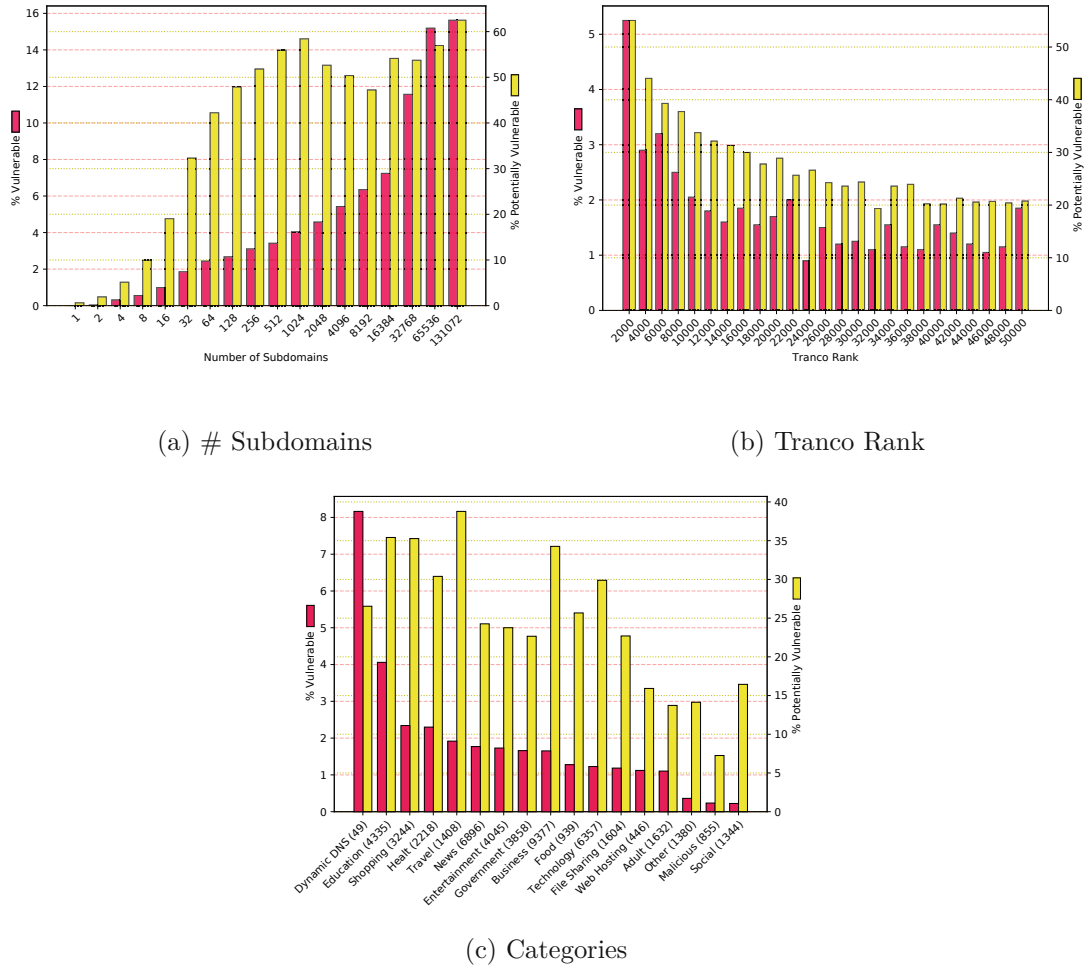


Figure 4.3: Characterization of vulnerable domains.

evaluating service and hosting providers offering subdomains to their users. We refer to §4.5.1 for a detailed analysis of Dynamic DNS services and hosting providers.

The second most affected category concerns education websites. We found that academic institutions generally have complex and heterogeneous public-facing IT infrastructures that translate into a high number of subdomains. By restricting the analysis to the .edu TLD, we observed 1,229 domains having on average 6,033 subdomains each. The percentage of domains with at least one vulnerable subdomain is 7.32%, which is substantially higher than any other TLD considered. For comparison, the percentage in .com is 1.81%.

Overall, we identified vulnerabilities affecting top domains across all categories. To exemplify, we found subdomain takeover vulnerabilities on news websites like `cnn.com` and `time.com`, university portals like `harvard.edu` and `mit.edu`, governmental websites like `europa.eu` and `nih.gov`, and IT companies like `lenovo.com` and

Table 4.4: Attackers' capabilities on vulnerable services.

| Service | Wildcard | Redirect (www) | PSL | Capabilities |
|-----------------|----------|----------------|-----|-----------------------|
| agilecrm | ! | — | ! | js https |
| anima | ! | — | — | js https |
| campaignmonitor | ! | — | ! | content |
| cargo | ! | ✓ | ! | js |
| feedpress | ! | — | — | html |
| gemfury | ! | — | — | file https |
| github | ! | — | ✓ | js file https |
| helpscout | ! | — | ! | js file https |
| jetbrains | ✓ | — | ! | content |
| launchrock | ! | ! | ! | js https |
| ngrok | ? | ? | ✓ | js file headers https |
| persona | ! | ✓ | ! | js https |
| pingdom | ! | — | — | js |
| readme.io | ! | — | ! | js https |
| shopify | ! | ! | ! | js https |
| smartjobboard | ! | ✓ | ! | js https |
| statuspage | ✓ | — | ! | js https |
| strikingly | ? | ? | ! | js https |
| surgesh | ✓ | ✓ | ! | js https |
| tumblr | ! | — | ! | js file https |
| uberflip | ? | ? | — | js https |
| uptimerobot | ! | — | — | content |
| uservice | ? | ? | ! | js https |
| webflow | ? | ? | ! | js https |
| wordpress | ✓ | ✓ | ! | js https |
| worksites | ! | ✓ | ! | js https |

Note: We use the following notation: service not affected (✓); service is vulnerable (!); the conditions of *redirect* and *PSL* do not apply (—); could not evaluate, e.g., due to payment required, no public registration form, etc. (?). Helpscout allows to host only arbitrary *active content* files (js, css); Gemfury allows to host only arbitrary *passive content* files (images, media, ...); Launchrock implicitly associates every subdomain to the mapped domain, not only the *www* subdomain.

cisco.com. Although most of the discovered issues could be easily fixed by routinely checking the validity of DNS records, our large-scale vulnerability assessment raises concerns due to the number and pervasiveness of the identified threats.

Analysis of Third-Party Services

We examined 26 service and hosting providers and 5 dynamic DNS services for a total of 31 third-party services. Our selection comprises services mentioned in previous work [LHW16] and community efforts [EdO], excluding those that required payment to carry out our analysis.

The results are summarized in Table 4.4. We combined manual testing and review of the documentation to assess the capabilities available to a registered user of each service. We also evaluated the considered services against the security pitfalls described in §4.3.2: (i) *wildcard*, the domain ownership verification allows attackers to claim subdomains of an already mapped domain, e.g., due to the presence of a wildcard DNS entry; (ii) *redirect*, if the *www* subdomain of a mapped domain automatically redirects to the parent domain, e.g., `www.shop.example.com` redirects to `shop.example.com`, whether the former

can be claimed by a different account; (iii) *PSL*, if the service allows users to create a website under a specific subdomain, whether the parent domain of the assigned website is included in the *PSL*.

Table 4.3 shows the distribution of the vulnerable subdomains across service providers. The majority of the vulnerable subdomains (93%) are hosted on the first four most used services: WordPress, Shopify, Tumblr, and GitHub Pages. These prominent services give users the ability to host a website with a valid TLS certificate for the associated domain. Users are allowed to customize the markup and JavaScript code of the pages, and for Tumblr and GitHub Pages, users are allowed to upload arbitrary files to their websites. In general, the capabilities obtained by an attacker controlling a service vary depending on the specific platform, ranging from content only (UptimeRobot) to full host control (ngrok). We found that 19 out of 26 services grant the `js` and `https` capabilities, while 21 provide the `js` capability alone. The `file` capability is the most uncommon, being available for 4 services only.

Surprisingly, we discovered that in 20 out of the 31 analyzed services, any registered user controls a website that is in a related-domain position to all the other websites hosted on the platform. Tumblr and WordPress, along with 11 additional services, even share their primary domain with user-controlled websites, e.g., `attacker.tumblr.com` is related to `tumblr.com`. Only GitHub and ngrok prevent this threat by including the apex domains assigned to their users in the *PSL*.

Lastly, we found that 17 services have issues with the ownership verification mechanism. Among the four most used services, only WordPress prevents attackers from claiming subdomains of an already mapped domain. Moreover, 8 service providers perform an automatic redirection from the `www` subdomain to the parent domain. Therefore, users of these services might erroneously assume that the `www` subdomain is implicitly bound to their account and cannot be claimed by others. Only Shopify and Launchrock do not prevent this subdomain from being mapped to different accounts. We reported to GitHub and Shopify, two of the major service providers, the vulnerabilities discovered on the domain ownership verification process. GitHub acknowledged the problem and told us that they “[...] are exploring various changes to the custom domain flow that will improve this situation by requiring formal domain ownership verification”. Shopify awarded us \$1,000 for the report and shipped a fix on April 12, 2021.

Dynamic DNS Services. The adoption of the *PSL* across different dynamic DNS providers is shown in Table 4.5, together with the number of domains that a user can choose from. We observed that only 2 providers listed all their domains in the *PSL*. Noip and DynDNS left out a small number of the domains they offer, but it is not clear to us whether this is due to negligence or if this is a deliberate choice. Instead, FreeDNS, with more than 50k domains, did not include any of them in the list, leaving their massive user base at risk. We reported this major flaw to the FreeDNS maintainer, who acknowledged it but took no action, as it would be impossible to maintain an updated list of thousands of domains in the *PSL*, given the lack of an API to manage *PSL* entries.

Table 4.5: PSL on dynamic DNS services.

| Service | # Domains | PSL |
|---------------------|-----------|------------|
| afraid (FreeDNS) | 52,443 | ❗ 0/52,443 |
| duckdns | 1 | ✅ 1/1 |
| dyndns | 293 | ❗ 287/293 |
| noip | 91 | ❗ 85/91 |
| securepoint | 10 | ✅ 10/10 |

Table 4.6: Web security abuses by related-domain attackers (RDA).

| Mechanism | | Deployed | | Exploitable by RDA | |
|-------------------|------------------|----------|-------|--------------------|-----------|
| | | Domains | Sites | Domains | Sites |
| Cookies | all | 23,400 | 857 | <i>C</i> 15,025 | 826 |
| | | | | <i>I</i> 23,178 | 845 |
| | session | 15,179 | 846 | <i>C</i> 5,051 | 687 |
| CSP | | | | <i>I</i> 14,964 | 834 |
| | script inclusion | 1,144 | 260 | 901 (0) | 212 (0) |
| | style inclusion | 961 | 232 | 930 (0) | 225 (0) |
| | object inclusion | 1,027 | 250 | 598 (+12) | 123 (+5) |
| | frame inclusion | 967 | 229 | 664 (+45) | 152 (+12) |
| | framing control | 1,676 | 360 | 344 (+97) | 59 (+21) |
| CORS | all | - | - | 2,254 (+224) | 488 (+51) |
| | with credentials | - | - | 179 (+63) | 71 (+27) |
| postMessage | | 14,045 | 823 | 14 (0) | 11 (0) |
| Domain Relaxation | | 97 | 61 | 57 | 29 |

Note: *C* and *I* denote cookie confidentiality and integrity. Numbers within parenthesis represent the improvement compared to a Web attacker; when missing, the Web attacker cannot perform the attack.

4.5.2 Web Threats

We now turn the attention to the Web application security implications of our analysis, as summarized in Tables 4.3 and further detailed in Table 4.6.

We start by discussing confidentiality and integrity of session cookies. Overall, our crawler collected 85,169 cookies, out of which 24,924 have been labeled as session cookies by our heuristic. Among these, we identify 3,390 (14%) cookies from 5,051 (33%) domains on 687 sites (81%) whose confidentiality can be violated by a related-domain attacker. This shows that related-domain attackers can often get access to session cookies, which may enable attacks like session hijacking. Our analysis also shows that the state of cookie integrity is even worse: in particular, we identify 24,689 (99%) session cookies from 14,964 (99%) domains on 834 (99%) sites which do not have integrity against a related-domain attacker, hence may enable attacks like session fixation and cookie forcing. This increase comes from the fact that related-domain attackers can compromise the confidentiality of domain cookies alone, while they can break the integrity of any cookie by exploiting *cookie shadowing* [ZJL⁺15]. The fraction of domains not affected by integrity issues is only due to the lack of capabilities available for the subdomain we could possibly take over. The only robust way to improve cookie integrity in this setting is the adoption of

the `__Host-` prefix, which is unfortunately negligible in the wild: we only identified one cookie using it in our dataset.

Concerning CSP, the first observation we make is that, as reported by previous studies [WSLJ16, CRB18, RBC⁺20], the majority of CSPs in the wild suffer from incorrect configurations, voiding their security guarantees even against Web attackers. Remarkably, however, related-domain attackers are more powerful than traditional Web attackers for real-world CSPs, being able to bypass the protection mechanism on 139 additional domains. This is apparent for object injection, frame injection, and framing control. For example, we quantified the following increase in the attack surface for frame injection: 45 (+7%) domains are exploitable exclusively by controlling one of the vulnerable subdomains identified in our dataset.

As to the other mechanisms, CORS deployments are significantly more at risk against related-domain attackers rather than against traditional Web attackers. In particular, we identify 224 (+11%) new exploitable cases, including 63 (+54%) cases with credentials. Note that the use of CORS with credentials is particularly delicate from a security perspective, hence the strong percentage increase in the number of vulnerable cases is concerning. Domain relaxation, instead, can be abused by related-domain attackers in 57 out of 97 domains (59%) making use of this mechanism. Exploiting domain relaxation puts a related-domain attacker in the same origin of the target Web application, hence bypassing all Web security boundaries: this is a critical vulnerability, which deserves attention. Domain relaxation is a bad security practice, which should better be avoided in the modern Web. Finally, our analysis of `postMessage` shows that all sites suffering from unsafe programming practices are already vulnerable against Web attackers, i.e., for this specific attack vector related-domain attackers are no more powerful than traditional Web attackers, at least based on the collected data. In other words, sites either do not enforce any security check or restrict communication to selected individual origins: this might be a consequence of the `postMessage` API granting access to origin information, rather than site information directly.

4.6 Disclosure and Ethical Considerations

RDSan identified 1,520 vulnerable subdomains on 887 distinct domains, of which 260 are subdomains pointing to an expired domain and 1,260 are those mapped to a discontinued service (see §4.4.2). Besides disclosing the vulnerabilities found on service providers (§4.5.1), we also attempted to notify all the websites affected by the issues we discovered. Prior work [LDC⁺16, SPR⁺16, SPL⁺18] showed that the identification and selection of correct security contact points is the main issue behind an overall unsatisfactory remediation rate. To maximize the chances of a successful notification campaign, we examined the following sources until a valid point of contact was found: (i) the list of bug bounty and security disclosure programs maintained by Bugcrowd [Bug20]; (ii) the `security.txt` file [FS20] in the root directory of the vulnerable domains and under the `/.well-known/` folder [Not19]; (iii) the Abusix [Abu20] database, queried with the

ip addresses of the vulnerable domains to collect the associated email contacts; (iv) a WHOIS lookups [Dai04]. We validated the obtained email addresses to avoid reporting vulnerabilities to unrelated parties, e.g., by checking whether the domain part of the email address matches any of the input domains. Unfortunately, using this procedure we could not find any security contact for the majority of the considered domains (62%). To inform them about their security vulnerabilities, we contacted our national CERT that willingly agreed to disclose the issues to the affected parties on our behalf. Among the few contacted websites with a bug bounty program, F-Secure awarded us with €250 for the reported subdomain takeover vulnerability.

Aside from vulnerability disclosure programs, our notification campaign is fully automatic: we sent an email to all the identified contacts containing a high-level description of the vulnerabilities and a link to the security advisory on our Web application which contains a detailed description of the problems found for a given domain, the required actions to fix the reported vulnerabilities, and instructions to opt out from future scans.

4.6.1 Outcome of the Notification Campaign

We performed a second run of RDScan on January 2021, 6 months after the first analysis, to picture the state of vulnerable instances left in the wild after our disclosure. We repeated the test for the whole set of expired domains instances. Concerning discontinued services, we focused on the 3 largest providers (WordPress, Shopify and Tumblr), representing 87% of the vulnerable subdomains found in the first round. Overall, we covered 1362 out of the original 1520 vulnerable subdomains (90%), which translates to 781 out of 887 sites (88%). To account for possible changes in services occurred in the meanwhile, we verified the takeover preconditions included in RDScan. After the conclusion of the analysis, we manually assessed a random sample of 10% of the results to ensure the correctness of the procedure without finding any discrepancy.

We discovered that only 200 out of 1362 subdomains (15%) have been fixed during this time frame, for a total of 125 sites over 781 (16%). We noticed that the sites which we contacted directly exhibit a noticeably higher fix rate (31% subdomain, 22% sites) than those alerted by our national CERT (10% subdomains, 14% sites). Unfortunately, we also observed that a considerable amount of sites fixed only a subset of their vulnerable subdomains, resulting still affected by threats posed by related-domain attackers.

The overall remediation rate of our notification campaign is in line with previous studies [SPR⁺16]. Nonetheless, we report that our procedure to identify appropriate contact points turned out to be successful considering that 34% of the contacted parties accessed the full vulnerability report on our Web application.

4.6.2 Ethical Considerations

We consciously designed our vulnerability scanning framework to avoid raising network alerts or causing harm to the analyzed targets. Specifically, the subdomain takeover assessment phase has been carried out mostly by DNS queries and simple HTTP requests.

Active websites have never been affected by our tests since we restricted the analysis to abandoned DNS records. We did not perform any large-scale portscan, but we opted, instead, for using a public dataset consisting of a scan of the full IPv4 range on 148 ports.

We also avoided checking the availability of IP addresses on cloud providers by iterating over the creation of multiple virtual machines, since this practice could interfere with the normal operations of the cloud platforms. Similarly, the Web analysis module did not execute attacks against the targets, but limited its operations to the passive collection of data (cookies and security policies), simple HTTP requests, and client-side testing. Overall, our approach proved to be lightweight and unobtrusive: we did not receive requests from the analyzed websites to opt out from future scans, and no complaints concerning our activity were sent to the abuse contact of the IPs used to perform the analysis.

4.7 Case Studies

We report on manually vetted case studies of confirmed attacks. All vulnerable parties have been promptly informed of the discovered issues, as discussed in §4.6.

4.7.1 Site Impersonation

We provide a concrete example of how the Shopify vulnerability described in §4.5.1 could have been abused to impersonate a major website. As of September 2020, the e-shop of fox.com was hosted on Shopify and made available at `shop.fox.com` using a custom domain mapping. Our scan verified the two preconditions to connect `www.shop.fox.com` to a Shopify store under our control, i.e., the existence of a DNS A record pointing the domain `www.shop.fox.com` to 23.227.38.65 (the IP address owned by Shopify to map custom domains) and that `www.shop.fox.com` was not associated with any registered store on Shopify.

We manually investigated the e-shop of fox.com and found that the redirection performed by Shopify from `www.shop.fox.com` to `shop.fox.com` caused the `www`-prefixed subdomain to be referenced in the store as a legitimate URL³. By taking over `www.shop.fox.com`, criminals could have abused this implicit trust to mount severe attacks against the legitimate store, such as phishing, reputation damage, and credential stealing. We notified the vulnerability to Shopify on August 27, 2020 and received a bounty for our disclosure. Around one month after the report, we noticed that FOX moved its e-shop to a different domain (`maskedingershop.com`). We have no evidence to assert whether this change is connected to our disclosure.

³See <https://web.archive.org/web/20200113052608/https://shop.fox.com/pages/faq> for a page mentioning `www.shop.fox.com`.

4.7.2 Session Hijacking

We describe an example of a subdomain takeover vulnerability that could have been exploited to hijack authenticated user sessions at the FedEx website. RDScan discovered a dangling DNS affecting the `cn.grantcontest.fedex.com` subdomain due to a CNAME record pointing to the purchasable domain `cngrantcontest.com`.

After taking control of the subdomain, attackers could escalate their privileges by exploiting the insecure configuration of session cookies on the main website. We manually verified that authenticated sessions with `www.fedex.com` were built upon domain cookies, which are sent by default to all subdomains (see §4.3.3). Thus, authenticated users would disclose their session cookies to the attackers just by visiting the compromised subdomain. After acquiring the victim's cookies, an attacker could automatically break into the victim's session and access confidential data stored on the Web portal. We notified FedEx about the takeover vulnerability in August 2020. The company acknowledged our findings and, as of January 2021, we confirmed that the vulnerability was fixed.

4.7.3 Leakage of PII data

Now we show how a related-domain attacker can abuse misconfigurations in the CORS policy to access personally identifiable information (PII) of a user on the F-Secure website. Our vulnerability scanning pipeline detected a CNAME record `uk.safeandsavvy.f-secure.com` pointing to the deleted WordPress blog at `safeandsavvyuk.wordpress.com`. Notice that subdomains of deleted blogs still resolve to a WordPress IP thanks to a CNAME wildcard for `*.wordpress.com`. To take over the F-Secure subdomain, an attacker could simply create an account on `wordpress.com` and set `uk.safeandsavvy.f-secure.com` as a custom domain.

We observed that WordPress allows paid accounts to install plugins which enable the inclusion of arbitrary scripts as part of the blog's theme. The ability to execute JavaScript from a subdomain of `f-secure.com` would allow attackers to exploit a CORS vulnerability identified by our Web analyzer on `api.my.f-secure.com`. Such domain was configured to relax the SOP on requests originating from any subdomain of `f-secure.com`, even when cookies are attached. An attacker could trick a victim into visiting a page on the compromised subdomain which performs a `fetch` request to, e.g., the `https://api.my.f-secure.com/get_userinfo` endpoint to read private information such as past billing details, tokens, etc. We notified F-Secure through their bug bounty program in August 2020 and received €250 for the report.

4.8 Related Work

Related-Domain Attackers. The notion of related-domain attacker was first introduced by Bortz, Barth, and Czeskis [BBC11]. Their work identified the security risks posed by related domains against (session) cookies and proposed a possible solution called *origin cookies*. A similar defense mechanism, i.e., the `__Host-` prefix, was eventually

integrated into major Web browsers. Other than that, related-domain attackers received only marginal attention from the security community, with a few notable exceptions. Zheng et al. discussed the security implications of the lack of cookie integrity in many top sites, considering both network and related-domain attackers [ZJL⁺15]. Calzavara et al. presented black-box testing strategies for Web session integrity, including related-domain attackers in their threat model [CRRB19]. Related-domain attackers have also been considered in formal Web security models, again in the context of Web sessions [CFG⁺20]. Our work significantly advances the understanding of related-domain attackers by discussing new security threats, which go beyond Web sessions and have been quantified in the wild through a large-scale measurement.

Attacking Subdomains. Subdomain takeover is an infamous attack, which has been covered by a body of work. Liu et al. [LHW16] studied the threat posed by dangling DNS records, e.g., records that contain aliases to expired domains or pointing to IP addresses hosted on cloud services. The authors performed a large-scale analysis that uncovered the existence of hundreds of dangling records among the subdomains of the top 10k sites of Alexa and under the .edu and .gov zones. With respect to [LHW16], we improved the subdomain enumeration part by a factor of 13 and increased the number of analyzed services from 9 to 31. Also, the paper does not extensively analyze the Web security implications of subdomain takeover. Borgolte et al. [BFH⁺18] improved on the results of [LHW16] concerning deprovisioned cloud instances and proposed an extension of the ACME protocol used by some CAs for domain validation (e.g., *Let's Encrypt*). Schwittman et al. [SWW19] studied these domain validation techniques and discovered several vulnerabilities that could be exploited by attackers to obtain valid certificates for domains they do not own.

Liu et al. [LLD⁺17] proposed a technique to detect shadowed domains used in malware distribution campaigns, i.e., legitimate domains that are compromised to spawn an arbitrary number of subdomains after taking control of the DNS configuration panel at the registrar. Alowaisheq et al. [ATW⁺20] recently demonstrated a domain hijacking attack that relies on the exploitation of stale NS records. Zhang et al. [ZZS⁺20] showed how a domain with HTTPS misconfigurations can be abused by a network attacker to force the communication over HTTP with its related domains. However, the authors consider two domains as related if they share the same TLS certificate, which differs from the definition considered in this work. A large body of works studied the problem of domain impersonation (e.g., [AJPN15, KML⁺17, RGW⁺19]) where attackers trick users to interact with their malicious websites by using domain names that mimic those of honest sites. An example is provided by doppelganger domains [Wir11] which are spelled similarly to legitimate subdomain names except for the dots that separate the components of the domain name. We consider all these threats out of the scope of our analysis, as they have different security implications than the vulnerabilities we discuss.

Web Measurements. Meiser et al. [MLS21] studied the cross-origin data exchange practices of 5k websites to assess to which extent their security could be affected by

the presence of an XSS vulnerability on one of their communication partners. In our work, we study a similar problem, but we restrict our focus to related domains, and we consider other mechanisms that are out of scope for [MLS21], e.g., CSP. Chen et al. [CJD⁺18] performed a large-scale measurement of CORS misconfigurations. Among the 480k domains that they analyzed, they discovered that 27.5% of them are affected by some vulnerability and, in particular, 84k trust all their subdomains and can thus be exploited by a related-domain attacker. Son and Shmatikov [SS13] analyzed the usage of the Messaging API on the top 10k Alexa websites. The authors found that 1.5k hosts do not perform any origin checking on the receiving message, while 261 implement an incorrect check: (almost) all these checks can be bypassed from a related-domain position, although half of them can also be bypassed from domains with a specially-crafted name. More recently, Steffens and Stock [SS20] proposed an automated framework for the analysis of postMessage handlers and used it to perform a comprehensive analysis of the first top 100k websites of the Tranco list. The authors discovered 111 vulnerable handlers, out of which 80 do not perform any origin check. Regarding the remaining handlers, the authors identified only 8 incorrect origin validations, showing an opposite trend with respect to [SS13]. Finally, insecure configurations of CSP have been analyzed in a number of research papers [WLR14, WSLJ16, CRB18, RBC⁺20]. However, none of these works considered the problem of related-domain attacks.

4.9 Conclusion

In this chapter, we presented the first analysis tailored at quantifying the threats posed by related-domain attackers to the security of Web applications. We first introduced a novel framework that captures the capabilities acquired by such attackers, according to the position in which they operate, and we discuss which Web attacks can be launched from that privileged position, highlighting the advantages with respect to traditional Web attackers. We also studied the security implications of 31 third-party service providers and dynamic DNS to identify the capabilities that a related-domain attacker acquires when taking over a domain hosted by them, and presented a novel subdomain hijacking technique that resulted in a bug bounty of \$1,000. Then, we described the design of our automated toolchain used to assess the pervasiveness of these threats in the wild. The toolchain consists of an analysis module for subdomain takeover that identifies which subdomains can be hijacked by an attacker. Next, the Web security module quantifies how many related domains can be attacked from the domains discovered in the previous step. We performed a large-scale analysis on the 50k most popular domains, and we identified vulnerabilities in 887 of them, including major websites like `cnn.com` and `cisco.com`. Then, we correlated for the first time the impact of these vulnerabilities on the security of Web applications, showing that related-domain attackers have an additional gain compared to Web attackers that goes beyond the traditional cookie issues.

Cookie Crumbles: Breaking and Fixing Web Session Integrity

Abstract

Cookies have a long history of vulnerabilities targeting their confidentiality and integrity. To address these issues, new mechanisms have been proposed and implemented in browsers and server-side applications. Notably, improvements to the Secure attribute and cookie prefixes aim to strengthen cookie integrity against network and same-site attackers, whereas SameSite cookies have been touted as the solution to CSRF. On the server, token-based protections are considered an effective defense for CSRF in the synchronizer token pattern variant. In this paper, we question the effectiveness of these protections and study the real-world security implications of cookie integrity issues, showing how security mechanisms previously considered robust can be bypassed, exposing Web applications to session integrity attacks such as session fixation and cross-origin request forgery (CORF). These flaws are not only implementation-specific bugs but are also caused by compositionality issues of security mechanisms or vulnerabilities in the standard. Our research contributed to 12 CVEs, 27 vulnerability disclosures, and updates to the cookie standard. It comprises (i) a thorough cross-browser evaluation of cookie integrity issues, that results in new attacks originating from implementation or specification inconsistencies, and (ii) a security analysis of the top 13 Web frameworks, exposing session integrity vulnerabilities in 9 of them. We discuss our responsible disclosure and propose practical mitigations.

This chapter presents the results of a collaboration with Marco Squarcina, Pedro Adão and Matteo Maffei and has been published at the 32nd USENIX Security Symposium in 2023 under the title “Cookie Crumbles: Breaking and Fixing Web Session Integrity” [SAVM23]. Marco Squarcina is responsible for the cross-browser evaluation of cookie integrity attacks, the discovery of new cookie integrity vulnerabilities and the definition

of the CORF attack. Pedro Adão is responsible for the security analysis of the top Web frameworks and the responsible disclosure of the discovered issues. I am responsible for the formalization of the Web frameworks in applied pi calculus and the verification of the correctness of our proposed mitigation for all frameworks. Matteo Maffei was the general advisor and contributed with continuous feedback.

5.1 Introduction

HTTP cookies are the oldest and most widely used mechanism for state sharing between Web clients and servers. They are a cornerstone of Web sessions and play a crucial role in the authentication and authorization of users. Despite their prominence in Web applications, cookies have a long history of vulnerabilities and several known pitfalls [BJM08a, BBC11, Lun13, SMWL10a, Zal11].

Entire classes of attacks revolve around compromising either the confidentiality or the integrity of cookies [CFST17]. For instance, *session hijacking* attacks aim to leak the value of a session cookie (e.g., via *cross-site scripting*) and use it to obtain unauthorized access to a website [OWAb]. *Session fixation* attacks involve compromising cookie integrity to force an attacker-controlled cookie in the victim's browser, and then impersonate the victim on the target website [Kol02]. *Cross-site request forgery* (CSRF) attacks, instead, are a typical session integrity violation problem where the attacker issues cross-site requests from the victim's browser to execute unwanted actions on a website in which the victim is authenticated [BJM08a].

In response to these attacks, new mechanisms have been proposed on both the client and the server side. On the client side, major browsers now support the updated cookie standard RFC6265bis [CEWW22] which includes extended security features compared to the original RFC from 2011 [Bar11a]. A notable example is the *SameSite* attribute, which has been touted as a robust solution against CSRF attacks [Hel17, Hel19]. Other changes focused on strengthening cookie integrity against same-site and network attackers, with improvements to the Secure flag and the introduction of `__Host-` and `__Secure-` cookie name prefixes [Netb]. On the server side, traditional protections against CSRF attacks include the usage of a secret token shared between browsers and servers [BJM08a]. This approach has been widely adopted by popular Web frameworks and considered an effective defense in the *synchronizer token pattern* variant [OWAa, LKP21].

In this chapter, we question the effectiveness of existing protections and study the real-world security implications of cookie integrity issues. In particular, we focus on network and same-site attackers [BBC11], a class of attackers increasingly becoming a significant threat to Web application security [STV⁺21]. We show how security mechanisms considered to be robust against these threat models can be bypassed, exposing Web applications to session integrity attacks such as session fixation and *cross-origin request forgery* (CORF). We suggest that these vulnerabilities are due to compositionality challenges between Web standards, browsers, and servers, and we propose a set of countermeasures to reconcile these issues. Overall, our research contributed to 12 CVEs,

27 vulnerability disclosures, and updates to the RFC of the cookie standard [CEWW22]. We identified novel attack vectors that bypass modern cookie protections and precisely characterize a class of attacks called *CORF token fixation* that highlights weaknesses in current CSRF protections. We performed a systematic security analysis of the top 13 Web frameworks, exposing session integrity vulnerabilities in 9 of them. We showed that these vulnerabilities are not only implementation-specific bugs but are caused by compositionality issues of security mechanisms or flaws in the standard. We also discussed the response of developers to our responsible disclosure and proposed mitigation strategies to improve the security of the Web ecosystem.

Contributions. Our contributions are summarized as follows:

- We extend the work of Squarcina et al.[STV⁺21] to propose a taxonomy of threat models that describes network and same-site attackers in terms of their capabilities and goals (Section 5.3).
- We perform a thorough cross-browser evaluation of known cookie integrity attacks and introduce new attacks classified along 4 different categories: serialization collisions due to nameless cookies, server-side parsing vulnerabilities, cookie jar desynchronization issues, and broken composition of (compliant) parsers. We present our methodology and discuss the result of a measurement study on nameless and prefixed cookies (Section 5.4).
- In Section 5.5, we precisely define the class of CORF token fixation attacks which captures known and novel bypasses of real-world CSRF protections, including the synchronizer token pattern which is considered robust against same-site and network attackers.
- Section 5.6 presents a systematic security analysis of the top 13 Web frameworks, exposing CORF and session fixation vulnerabilities in 9 of them. We discuss the response of developers to our responsible disclosure and propose a set of practical countermeasures to prevent our attacks.
- We formally verify the correctness of our proposed mitigation to the synchronizer token pattern using the ProVerif protocol verifier [Bla01] (Section 5.7).

We publish all artifacts developed during this research, including the browser test suite (Section 5.4.3), the dataset and processing code of our measurement (Section 5.4.4), the ProVerif models and scripts (Section 5.7), as well as the reproducible proof-of-concept attacks against Web frameworks (Section 5.6) [BVV⁺23].

5.2 Background

In the following, we provide an overview of cookie attributes, including existing mechanisms for cookie integrity, and CSRF protections. We first revise standard notions such as *origins* and *sites* being instrumental to the rest of the chapter.

5.2.1 Origins and Sites

The *same-origin policy* (SOP) [Bar11b] defines the traditional Web security boundary between websites. The SOP is based on the notion of origin, defined as a tuple of scheme, host, and port. For instance, the origin of `https://example.com:443` is `<https, example.com, 443>`. The SOP prevents an origin from reading or modifying the contents of a different origin. However, some components of the Web platform have a different scope. Cookies, for instance, are scoped to the *registrable domain* of the website that set them. A registrable domain is a domain name with one label on the left side of an *effective top-level domain*, as defined by the *Public Suffix List* (PSL) [Moza]. Hosts sharing the same registrable domain are considered to be same-site, e.g., `example.com`, `auth.example.com`, and `api.staging.example.com` all belong to the same site `example.com`. Same-site hosts are also called *sibling domains*.

In recent years, the definition of same-site evolved to include the URL scheme [webc]. Hence, sibling domains with different schemes are considered same-site, but not *schemeful same-site*. To avoid ambiguities, we maintain both terminologies and refer to same-site only when the scheme is irrelevant.

5.2.2 Cookies

Cookies are the main state management mechanism of the Web, allowing servers to maintain a stateful session over the stateless HTTP protocol [CEWW22]. Servers can set a cookie in the browser through the `Set-Cookie` header. This cookie is then automatically attached by the browser to all following HTTP requests to the server via the `Cookie` header. Additionally, JavaScript code running in Web pages can access and set the value of cookies using the traditional `Document.cookie` property or the new Cookie Store API [Neta].

Attributes. Cookies can be configured with attributes, or flags, which specify additional properties or constraints. The `Path` attribute allows to limit the cookie to a set of URL paths, i.e., the browser will include the cookie in HTTP requests if the path of the request URL matches or is a subdirectory of the `Path` attribute. The `Domain` attribute broadens the scope of a cookie. The value of this attribute can be assigned to any of the parent domains of the origin that sets the cookie, up to the registrable domain. For instance, a server at `foo.example.com` can set a cookie with `Domain=example.com` to specify that the cookie should be attached to all subdomains of `example.com`. If the attribute is omitted, the browser will send the cookie only to the host that set it. `HttpOnly` prevents the cookie from being accessed by JavaScript, e.g., via the `Document.cookie` property. The `Secure` attribute limits the scope of the cookie to secure connections. Browsers must reject the insertion of a cookie from a non-secure origin if the cookie jar already contains a secure cookie with the same name and scope.

Same-Site Cookies. The `SameSite` attribute has been introduced in 2016 as a defense in depth protection against CSRF attacks by confining cookies to same-site

requests [WG16]. In particular, the standard defines three same-site policies: `Strict`, cookies are attached to same-site requests only, i.e., no cookie is attached to cross-site requests; `Lax`, cookies are attached to same-site requests and cross-site top-level navigations, e.g., clicking on a link, using the GET request method; `None`, cookies are attached to all requests, cross-site included. According to the standard, SameSite cookies follow the *schemeful same-site* definition to determine whether a request is cross-site. This is in contrast to Domain cookies which do not consider the URL scheme, unless used in combination with the Secure attribute. SameSite cookies also represent one of the most effective protection against XS-Leaks, an emerging class of attacks that exploits gaps in the *same-origin policy* (SOP) to infer information such as PII and the authentication status of a user from a cross-site position [SKC20, SRBS19, GPJV20, VGFSR⁺22]. The SameSite attribute restricts the ability to initiate authenticated requests to same-site attackers, thus preventing traditional Web attackers from leaking the user’s state on a website.

Cookie Prefixes. Cookie prefixes, originally introduced in 2015 [Wes], enable additional security constraints on cookies based on their name. The specification defines two prefixes: when a cookie name begins with `__Secure-`, the cookie must be set with the Secure attribute and from a page served over HTTPS; when the name of a cookie starts with `__Host-`, in addition to all restrictions of the `__Secure-` attribute, the Path attribute must be explicitly set to `/`, and it must not contain the Domain attribute, locking the scope of the cookie to the host that created it. These additional constraints guarantee the integrity of `__Host-` cookies against same-site attackers, as such cookies are unaffected by shadowing attacks performed from a same-site position (see Section 5.4).

5.2.3 CSRF Protections

CSRF attacks are a well-known class of attacks where the adversary executes unauthorized state-changing actions under the victim’s authenticated session. A CSRF attack is always preceded by a setup phase where the attacker prepares a malicious website that silently performs a cross-site request to the target website to execute the unauthorized action, e.g., via an automatic form submission or the fetch API.

Over the years, many types of CSRF defenses have been proposed in the literature, including (i) origin/referrer checks, (ii) token-based mechanisms to ensure request unguessability, (iii) the SameSite cookie attribute, and (iv) explicit user interaction such as CAPTCHAs [BJM08a, LKP21]. All these protections have some limitations and drawbacks. For instance, SameSite cookies are not effective against attacks performed from a same-site position. To avoid ambiguity, we use the term *Cross-Origin Request Forgery* (CORF) in the rest of the chapter, as it includes the attack scenario of a network or same-site attacker. We focus our analysis on token-based protection techniques as they are the most common defense adopted by Web frameworks [LKP21], and – as shown in Section 5.7 – can offer robust protection if correctly implemented. The main idea is to send an unguessable parameter t , commonly named CSRF token, with every state-changing request, typically as a hidden input field in a form. By ensuring that t remains

secret to the attacker, cross-origin forged requests will be discarded by the target website, as the token t is missing. Below, we discuss the two most popular token-based protection patterns [OWAa, LKP21].

Synchronizer Token Pattern (STP). In STP, the server generates CSRF tokens and inserts them in every webpage that may lead to a state-changing operation, e.g., as a hidden field in a form for transferring funds. This token is then bound to the user's session and the server validates newly received tokens by verifying the correctness of this binding. Multiple implementations (see Section 5.6) generate a fixed CSRF secret s per session, and use it to derive CSRF tokens $t(s)$. Other implementations generate a fresh CSRF secret s per request, and derive CSRF tokens $t(s)$ similarly to the previous case. In this pattern, secrets are always linked to the user session, irrespective of whether it is stateful or stateless. In the former case, secrets are stored in the server session, whereas in the latter, client-side storage mechanisms, e.g., cookies, are used to synchronize the secret between the server and the browser.

Double Submit Pattern (DSP). In this pattern, the CSRF token is a random value stored in a cookie other than the session cookie. The server typically renders the CSRF token in the HTML page as a hidden input field, and the browser sends it back to the server as part of the authenticated request. The server then verifies the validity of the request by checking the equivalence between the cookie value and the CSRF token. This makes DSP more suitable for stateless sessions, as it does not require the server to store the CSRF secrets or tokens in the session. Notice that CSRF cookies can be encrypted or signed with a fixed key or secret stored on the server. In this case, the server-side validation should account for an additional decryption or validation step before performing the comparison. Additionally, servers could store a CSRF secret in a cookie and use it to derive the CSRF token: whenever the CSRF secret or token is not cryptographically bound to the current session identifier, we still refer to this pattern as DSP.

5.3 Threat Model

In this work, we aim to investigate the security risks that arise from the interaction between a website and a victim's browser when a network or a same-site attacker can forge cookies scoped to the target website. As shown in recent works, these two threat models are still relevant today. According to Zheng et al. [ZJL⁺15], only 0.13% of the top 1M websites in 2015 were protected from network attackers thanks to full HSTS deployment. The situation improved in 2022, although 90% of websites remain vulnerable [bHA22]. Large-scale studies on subdomain takeover vulnerabilities demonstrated the impact of same-site attackers. In 2016, Liu et al. [LHW16] identified 227 of the Alexa top-10K sites affected by vulnerable subdomains. Borgolte et al. [BFH⁺18] studied deprovisioned cloud instances, finding 700K vulnerable domains. Squarcina et al. [STV⁺21] estimated 13K potentially vulnerable domains due to deprovisioned cloud instances and discovered 887 sites with other subdomain takeover vulnerabilities among the top 50K sites in the

| Capability | Description |
|----------------------|--|
| <code>headers</code> | Control arbitrary HTTP response headers at w_a . |
| <code>js</code> | Execute arbitrary JavaScript on a page at w_a . |
| <code>https</code> | The scheme of w_a is <code>https</code> . |

Table 5.1: Capabilities required to set cookies in the victim’s browser from a sibling domain of the target (w_a).

Tranco list. They also discussed the dangers posed by corporate networks, roaming services, and dynamic DNS providers, which put users in a same-site position without carrying out attacks.

We consider a range of threat models corresponding to different levels of control and visibility that an attacker may have over the network and sibling domains of the website. To exclude trivially vulnerable scenarios, we assume that the victim accesses the target website over a correctly-configured secure channel. We do not discuss specific attack vectors that can be exploited to acquire a certain position since they have been extensively covered in the past [Det14, LHW16, BFH⁺18, STV⁺21]. We focus, instead, on the capabilities of standard threat models that are relevant to violations of cookie integrity. To do so, we build on the framework introduced by Squarcina et al. [STV⁺21]. Table 5.1 outlines the capabilities that are relevant to set cookies, assuming a target website w , the set of its sibling domains S_w , a website controlled by an attacker $w_a \in S_w$, and the victim’s browser B . Different combinations of these capabilities enable precise characterization of the threat models considered in this work, as shown in Figure 5.1.

SS-HOST-S maps to a *same-site attacker*, also called *related-domain attacker*, with full control over a sibling domain of the target with a valid TLS certificate. This attacker can render arbitrary content over a secure channel, having the full set of capabilities `https`, `js`, and `headers`.

SS-HOST-I is similar to **SS-HOST-S**, excluding the ability to host pages over a secure channel. This threat model captures the case where an attacker controls a sibling domain of the target but cannot obtain a valid TLS certificate, e.g., due to the presence of a CAA DNS record defining a strict allow-list of permitted CAs [STV⁺21]. The capabilities are `js` and `headers`.

SS-XSS-S is a same-site attacker obtaining indirect control over a sibling domain via a script injection vulnerability (XSS) on a page served via HTTPS. Since the attacker is not in control of the response headers returned by the page, the capabilities are `https` and `js`.

SS-XSS-I is an attacker with an XSS vulnerability on a sibling domain served over an insecure connection. The only available capability is `js`.

NET maps to a standard network attacker who can fully control cleartext traffic generated by the victim’s browser. This attacker is able to intercept, modify, and inject network traffic of any sibling domain of the target domain, including the target domain

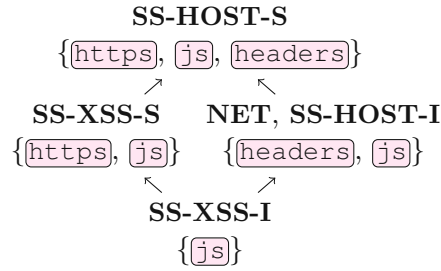


Figure 5.1: Taxonomy of threat models for cookie integrity violations.

itself. These capabilities translate into the `headers` and `js`, similarly to the **SS-HOST-I** attacker. Notice that network attackers cannot manipulate cleartext network traffic if the domain enforces a strict HSTS policy that includes the `includeSubDomains` directive [ZJL⁺15].

We also formulate a precise definition of *cookie integrity violations*, taking into account the cookie’s intended recipient. We assume that the attacker aims to compromise a cookie $c = \langle n, v \rangle$ with name n and value v , stored in the victim’s browser B for the origin o . In a *server-side integrity violation*, the attacker implants a cookie $c' = \langle n', v' \rangle$ in the victim’s browser B with the goal of forcing the browser B to send c' to o . The server at o parses the `Cookie` header obtaining a cookie with name n but tampered value $v' \neq v$. We refer to a *client-side integrity violation* when the attacker causes the JavaScript `Document.cookie` property on o to return a key=value pair where the key corresponds to n and the value is chosen by the attacker. Additionally, we consider cookie eviction attacks as integrity violations, i.e., attacks that evict the cookie c from requests to o or remove the cookie from the key=value pairs returned by the `Document.cookie` API on o .

5.4 Violationg Cookie Integrity

In this section, we show how attacker capabilities, and therefore the standard threat models discussed in Section 5.3, map to concrete attacks. First, we systematize known cookie integrity pitfalls and evaluate them on the top 3 Web browsers. Then, we introduce a range of novel attacks along 4 attack classes enabled by inconsistencies between servers, browsers, and the cookie specification. We show that these attacks are possible in practice and can be used to break cookie integrity in unprecedented ways. Finally, we discuss the methodology adopted to discover these issues and report on a measurement study performed using the HTTP Archive dataset [Arc].

5.4.1 Weak Integrity

Due to their legacy design, cookies have a long history of integrity issues, as documented in the cookie specification [CEWW22]. A comparison of the top 3 browsers on the





| Attack | RFC | Browsers | | |
|--|---|---|---|---|
| |  |  |  |  |
| Tossing (creation date, latest first) | ✗ | ✓ | ✓ | ✗ |
| Tossing (insecure over secure cookie) | ✗ | ✓ | ✓ | ✗ |
| Eviction (cookie jar overflow) | ✓ | ✗ | ✗ | ✓ |
| Eviction (___Host- via secure cookies) | ✓ | ✗ | ✗ | ✓ |
| Serialization collision (=a=b→a=b) | ✓ ^{≥04} | ✗ | ✗ | ✓ |
| Serialization collision (___Host-) | ✗ ^{≥11} | ✗ ^{<104} | ✗ ^{<105} | ✓ |
| Cookie jar desynchronization | ✗ | ✓ | ✗ | ✓ |
| Server-side parsing issues | ✗ | — | — | — |
| Parser-chaining | ✓ | — | — | — |

Table 5.2: Evaluation of cookie integrity attacks against the cookie standard RFC6265bis-11 and browsers: Chrome (v109), Firefox (v109), and Safari (v16.0). ✓ compliant, ✗ violation, ✓ unaffected, ✗ vulnerable, — does not apply.

integrity pitfalls discussed below is included in Table 5.2 together with the new attacks introduced in this section.

Cookie Tossing

Cookies scoped for a target origin o are sorted by standard-compliant browsers by the most-specific matching `Path` attribute, meaning that cookies set with `Path=/foo` are sent before cookies with `Path=.`. When `Path` attributes are equal, cookies are sorted by creation time, i.e., cookies set first are sent before cookies that are set later. Although the standard states that servers should not rely on the order of cookies sent by browsers, most implementations only consider the first occurrence of a cookie name in the `Cookie` header field [ZJL⁺15]. Since attributes are not sent along with cookies, duplicated cookies with the same name but different `Path` attributes are indistinguishable to the server [CEWW22, §5.7.3].

Attackers can exploit this behavior to violate cookie integrity. For example, consider a Web application at `https://site.tld/login/index.php` that sets a cookie via the response header `Set-Cookie: sid=good; Path=.`. Assume also an attacker in control of `http://atk.site.tld/`. The attacker can set a domain cookie for `site.tld` with name `sid` and value `evil`. By setting a more specific path in the new cookie, the attacker can cause the victim’s browser to send the attacker’s controlled cookie first, as in Figure 5.2. This specific attack is called *cookie tossing*, or *shadowing*.

As mentioned in Section 5.2, `___Host-` prefixed cookies are considered to be unaffected by shadowing attacks from a same-site position. Furthermore, the standard specifies that secure cookies have strong integrity against non-secure origins. To summarize, cookie tossing requires the `https` capability only for cookies with the `Secure` flag. Otherwise, either the `headers` or the `js` capability is needed.

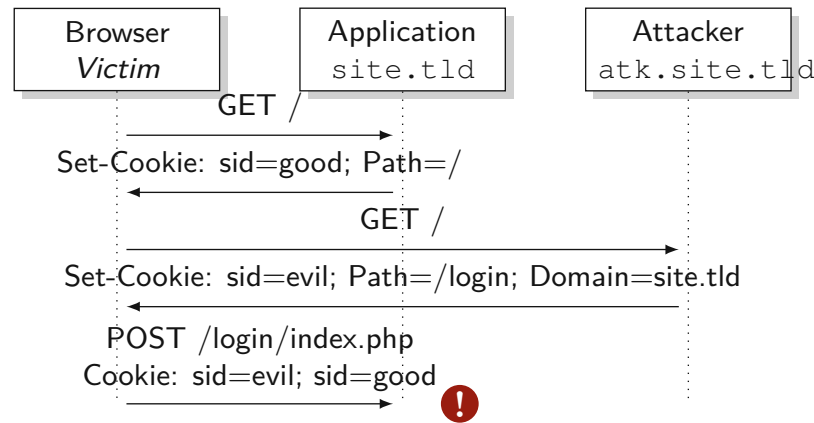


Figure 5.2: Cookie tossing attack.

Eviction Techniques

Cookies are evicted from the browser’s storage when the storage limit is reached. The eviction policy and precise limits are not specified by the standard, and are left to browser vendors to decide. In practice, recent versions of Firefox and Chrome limit the size of the cookie jar to 180 cookies per schemeful site, while Safari does not enforce any limit. In addition, browsers evict cookies in a least-recently-used (LRU) fashion, i.e., the oldest cookies are evicted first. This is problematic because it allows attackers to control the eviction of cookies by overflowing the cookie jar, and then use cookie tossing to replace the evicted cookies with their own. It is worth mentioning that the `HttpOnly` flag does not provide integrity against an attacker with the `(js)` capability. Indeed, while `HttpOnly` cookies cannot be read via JavaScript, they can be evicted by any of the threat models considered in this work. On the other hand, the `Secure` flag does provide integrity against attackers without the `(https)` capability, since modern browsers partition cookies by scheme.

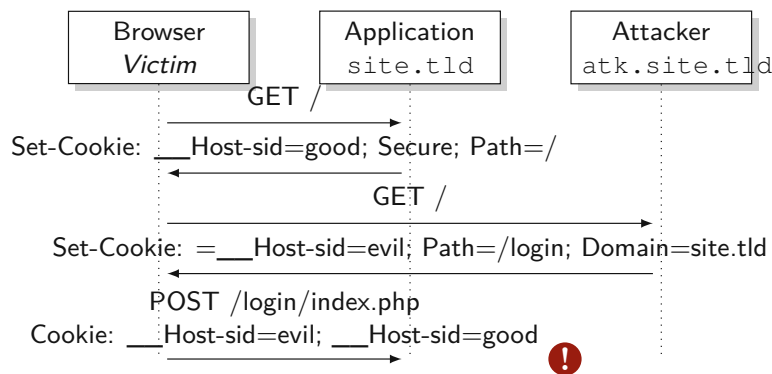
5.4.2 Novel Attacks

The cookie standard evolved in recent years to provide stronger integrity guarantees. In particular, the `__Host-` prefix was proposed in 2015 [Wes] to prevent cookie tossing attacks. In the following, we present a range of novel cookie integrity attacks that exploit issues in the cookie standard, server and client implementation problems, and the combination of both.

Nameless Cookies and Serialization Collisions

In 2020, a change to the cookie standard¹ added support to *nameless cookies*, i.e., cookies set with empty name and non-empty value. This change was motivated by some servers

¹RFC6265bis, Accept nameless cookies: <https://github.com/httpwg/http-extensions/commit/0178223>

Figure 5.3: `__Host-` cookie bypass via nameless cookies.

setting cookies with empty names, and the cookie standard did not specify how to parse them. As a result, the standard now mandates browsers to parse the `Set-Cookie: token` header as a nameless cookie with value `token`. This cookie must be serialized as `Cookie: token`, without any `=` character.

We found that this design introduces a novel attack vector that can bypass even the `__Host-` prefix. Consider, for instance, a page at `site.tld` that sets a named cookie `sid=good`. A same-site attacker can set a nameless cookie scoped to `site.tld` with value `sid=evil`. This can be done via either the `Document.cookie` property or the HTTP response header `Set-Cookie: =sid=evil; Domain=site.tld`, which is a valid header. According to the standard, the attacker-controlled cookie is serialized as `Cookie: sid=evil`, resulting indistinguishable to the server, or to frontends using the `Document.cookie` getter, from a cookie named `sid`.

This attack is particularly dangerous because it can violate the integrity guarantees enforced by `__Host-` cookies. Indeed, any attacker in our taxonomy can shadow a cookie `__Host-<name>=<value>` by forcing in the victim's browser a nameless cookie via `Set-Cookie: =__Host-<name>=<value>; Domain=<domain>`. An example of the attack flow is in Figure 5.3.

The same attack vector can shadow arbitrary secure cookies from an insecure origin. As explained in Section 5.2, browsers must reject a cookie set from a non-secure origin if the cookie jar contains a secure cookie matching the name of the new cookie scoped to the same site. Since secure cookies are partitioned differently from insecure ones, the `https` capability is typically required to perform an eviction or a cookie tossing attack against a secure cookie. This attack, however, lowers the preconditions for the integrity violation of secure cookies, requiring only the `headers` or the `js` capability.

Disclosure. The attacks above are representative of a larger class of serialization issues that we reported to the IETF HTTP Working Group on the cookie standard [HTT22] and jointly disclosed the `__Host-` cookie bypass to the Chrome [Chr22a] and Firefox [Moz22c]

security teams who issued CVE-2022-2860 and CVE-2022-40958, respectively.² Chrome fixed the issue in version 104 and Firefox in version 105. Safari is not affected by this vulnerability because it deviates from the standard since it serializes nameless cookies by prefixing the value with `=`. Our contributions and extensive discussion with browser maintainers [Chr22b] led to updates to the cookie standard [CEWW22, §5.6, point 22] that now mandates browsers to reject nameless cookies with a value starting with a case-insensitive match for `__Host-` or `__Secure-`.

Server-Side Parsing Issues

The cookie standard [CEWW22, §5.5] describes a set of parsing rules for the `Set-Cookie` header that user agents must follow. Unfortunately, the standard does not clearly specify how servers should parse cookies received via the `Cookie` header. This discrepancy causes server-side cookie integrity violations whenever servers parse two distinct cookies as the same one.

Although the problem is not new per se [ZJL⁺15], we discovered a new vulnerability that bypasses `__Host-` cookies in PHP [PHP22], the server-side language used by 78% of websites [W3T23]. Due to the legacy design derived from `register_globals` [PHP], PHP replaces spaces, dots, and open square brackets with the underscore symbol `_` in the keys of `$_POST` and `$_GET` superglobal arrays. The same string transformation applies to the keys of the `$_COOKIE` superglobal array. As a result, an attacker can fixate a cookie in the victim's browser via `Set-Cookie: __Host-sid=evil; Domain=site.tld`, that is parsed by PHP as `Cookie: __Host-sid=evil`. This vulnerability extends integrity concerns to all cookies that contain the underscore symbol, e.g., non-secure origins can use this bug to shadow secure cookies. Similarly, the HTTP server component of the ReactPHP library incorrectly parses the `Cookie` header by url-decoding cookie names [Dat22]. This vulnerability can be exploited to bypass `__Host-` cookies using percentage-encoded names, e.g., a cookie set via `Set-Cookie: %5F%5FHost-sid=evil; Domain=site.tld` is parsed by ReactPHP as `Cookie: __Host-sid=evil`.

We also discovered a vulnerability in the Werkzeug library, the HTTP middleware used by the popular Flask framework [Dat23]. The `Cookie` header is incorrectly parsed by stripping all leading `=` symbols. To exemplify, a nameless cookie set via `Set-Cookie: ==__Host-sid=evil; Domain=site.tld` is parsed by Werkzeug as a name-value pair corresponding to `(__Host-sid, evil)`.

All threat models discussed in Section 5.3 can mount these attacks that exploit server-side parsing issues, meaning that only the `headers` or `js` capabilities are required.

Disclosure. The PHP vulnerability was assigned CVE-2022-31629 and fixed in PHP 7.4.31, 8.0.24, and 8.1.11. ReactPHP issued CVE-2022-36032 after our report and fixed

²The `__Host-` bypass vulnerability was reported 3 weeks earlier as an independent effort by Axel Chong who is credited on both CVEs. Our issues were merged into the previous vulnerability reports to jointly discuss the mitigation and additional edge cases.

Listing 5.1: Cookie jar overflow desynchronization in Firefox.

```
// Assume an empty cookie jar then set 181 cookies
for (let i=1; i<=181; i++)
    document.cookie = 'a'+i+'=';
// Count the number of cookies
document.cookie.split(";").length
> 181 // Higher than the limit of 180 cookies per site
```

the vulnerability in version 1.7.0. The Werkzeug vulnerability obtained CVE-2023-23934 and has been patched in version 2.2.3.

Cookie Jar Desynchronization

We identified two vulnerabilities in Firefox that cause a desynchronization between the cookies listed by `Document.cookie` and the actual content of the cookie jar. We experimentally discovered that a cookie jar overflow operated via JavaScript sets more cookies than the maximum number of cookies allowed on a single site. Surprisingly, these cookies can only be retrieved via the `Document.cookie` API and are not effectively set in the cookie jar, i.e., they are not attached to subsequent HTTP requests [Moz22a].

The issue can be easily reproduced using the JavaScript code snippet in Listing 5.1. This example stores 181 cookies (a1 to a181) in `Document.cookie`, however, manual inspection of the cookie jar reveals that only 151 cookies are set (a31 to a181). Attempts to clear the cookie jar via the Firefox storage inspector, setting an expiration date in the past via the `Set-Cookie` header, or using the `Clear-Site-Data` header [W3C17], fail to remove the first 30 cookies (a1 to a31). This set of cookies survives page reloads and schemeful-same-site navigations. It is also preserved in new schemeful-same-site windows created via the `Window.open` method. The only way to remove them is to set a past expiration date via JavaScript, or by closing the browser tab.

The described issue can be exploited to violate client-side cookie integrity and requires the `js` capability, with the optional `https` capability if the target website is on a secure origin. Notice also that this inconsistent state could introduce vulnerabilities in applications trusting cookies read from `Document.cookie`, providing a novel avenue for attacks. For instance, frontends often set custom HTTP headers using the values of specific cookies read via the `Document.cookie` property. Notable examples are ASP.NET [Mic22] and Angular [Ang22].

The second desynchronization issue happens when there is a secure cookie set by a domain, and a page on a same-site non-secure origin tries to set another cookie with the same name using `Document.cookie` [Moz22b]. We discovered that the insecure cookie is not stored as required by the standard, but it is listed by the `Document.cookie` property. This inconsistency can create confusion on frontends that rely on `Document.cookie` to

read cookies. However, the security impact of this second desynchronization issue is limited since it only affects insecure origins that are trivially vulnerable to cookie integrity attacks.

Disclosure. We reported both issues to the Firefox security team in June 2022. According to Firefox developers, the root cause of these problems is the composition of cookies’ access control policies with Firefox’s implementation of Site Isolation, project Fission [Moz22d]. The second issue has been fixed in Firefox 112 and obtained CVE-2023-29547, whereas the first one is still under active investigation as of May 2023.

Parser Chaining Vulnerabilities

The serialization collision previously discussed introduces a new attack vector against chains of cookie parsers. We investigated the presence of this configuration in real-world applications by studying the AWS API Gateway, a service that acts as a frontend for other AWS services. The AWS Lambda proxy integration for HTTP APIs enables developers to bridge an API route with a Lambda function, passing request payloads to the Lambda function using a JSON message exchange format. According to the documentation [Ser22]: “Format 2.0 includes a new cookies field. All cookie headers in the request are combined with commas and added to the cookies field. In the response to the client, each cookie becomes a set-cookie header.”

From our tests, this proxy introduces an additional parser that serializes the cookies in the request payload. As a result, a cookie attached to a request, such as `Cookie: =__Host-sid=evil` corresponding to a nameless cookie with value `=__Host-sid=evil`, is serialized by the AWS Lambda proxy as `{"cookies": ["__Host-sid=evil"], ...}`, resulting indistinguishable from a legitimate cookie named `__Host-sid`. Notice that this specific attack is not prevented by recent Chrome and Firefox mitigations against `__Host-` cookie collisions, since the cookie value starts with the `=` symbol.

Disclosure. We reported the issue to the AWS security team in October 2022 that deployed a fix in November 2022. The mitigation consists of discarding key-value cookie entries starting with the `=` symbol followed by a case-insensitive match for `__Host-` or `__Secure-`. This approach, combined with modern browsers that adhere to the latest draft of the cookie standard [CEWW22], effectively protects against the threat described in this section.

5.4.3 Discovering Cookie Integrity Issues

The methodology used to discover these attacks consisted of three main stages.

Browser Testing. We performed a comprehensive evaluation of known cookie integrity attacks across the top-3 browsers (Chrome, Firefox, and Safari). Inspired by the WPT project [WPTb], we developed a suite of test cases that simulated various types of attacks and evaluated the behavior of the browsers. The test cases were designed to cover all possible combinations of secure and insecure origins between the victim and a same-site

attacker. We also tested different ways to set cookies, i.e., via the Set-Cookie header or using the JavaScript `Document.cookie` property. The test cases were run on the latest browser versions, and the results were analyzed to identify any inconsistencies between the browsers. Additionally, we used BrowserStack³ to test all releases from January 2021 to January 2023 of the three major browsers against our test suite and identify any changes in the behavior over time. This phase was crucial to uncover little-known discrepancies between the browsers. For instance, Safari sorts cookies by placing the most recent one first, while Firefox and Chrome serialize cookies starting from the oldest one, as mandated by the specification. We also verified that Safari does not prevent cookie tossing of secure cookies from non-secure cookies, which is a violation of the standard [CEWW22]. Additionally, we experimentally verified that Safari does not enforce limits on the maximum number of cookies stored for a single site. Finally, the test suite enabled the automatic discovery of the cookie jar desynchronization issue in Firefox, which was previously unknown to the security community.

Reviewing the Cookie Standard. Whenever a discrepancy was found between the browsers, we manually reviewed the cookie standard [CEWW22] to determine what was the expected behavior. During this phase, we learned that the standard introduced support to nameless cookies in 2020 and we discovered the serialization collision issues. We engaged with the IETF HTTP Working Group and browser vendors to address the problems as we found them.

Testing Server-Side Parsers. As a third stage of the analysis, we investigated the presence of inconsistencies in the cookie parsers of the server-side languages and core HTTP handling libraries used by the frameworks discussed in Section 5.6. For each target considered in our analysis, we developed a small *reflector* program that parses the Cookie header and returns pairs of cookie names and values. Then, we wrote a simple fuzzer to generate variations of the Cookie request header and automatically assessed how the header was parsed by our programs. We acknowledge that this approach does not constitute a systematic evaluation of server-side parsing inconsistencies. However, our initial analysis provided strong evidence of the pervasiveness of the issue. We leave such comprehensive study as future work.

5.4.4 Measurement of Cookie Name Prefixes and Nameless Cookies

We present the results of our measurement of the prevalence of cookie name prefixes and nameless cookies in the top 100K websites. We based our evaluation on the public HTTP Archive dataset [Arc] and performed all queries against the database provided by the Web Almanac initiative [bHA22]. We considered the website popularity rank in the Chrome User Experience Report (CrUX) [Goo], which distinguishes the popularity of origins by orders of magnitude (top 1K, 10K, 100K, etc.). CrUX introduced the rank metric in February 2021 [HP21], thus we restricted the measurement to the last 2 years to avoid any bias due to mixing different ranking metrics. We also excluded third-party

³<https://www.browserstack.com/>

| Rank | Origins | Secure | ___Host- | ___Secure- | Nameless |
|------|---------|---------------|------------|------------|-----------|
| 1K | 732 | 537 (73.4%) | 6 (0.8%) | 1 (0.1%) | 1 (0.1%) |
| 10K | 5952 | 4005 (67.3%) | 14 (0.2%) | 19 (0.3%) | 6 (0.1%) |
| 100K | 58068 | 35098 (60.4%) | 113 (0.2%) | 109 (0.2%) | 86 (0.1%) |

Table 5.3: Number of origins from the 2022-06-01 dataset setting cookies, and the percentage of origins using the Secure attribute, cookie prefixes, and nameless cookies.

cookies from our analysis and focused instead on first-party cookies to avoid popular CDNs and analytics services from affecting the results.

Table 5.3 reports the outcome of our measurement performed on the dataset from June 2022. The table shows the number of origins that use the Secure attribute, the ___Host- and ___Secure- prefix, and nameless cookies. Figure 5.4 provides a direct comparison between July 2021 and June 2022 of the adoption of cookies on the top 100K origins. As expected, prominent websites are more inclined towards well-established security features such as the Secure attribute. We found that more than 70% origins in the top 1K range are using secure cookies, while the percentage decreases to 60% in the top 100K range. Interestingly, while the adoption of secure cookies remained overall stable in the last 2 years for the top 1K websites, lower-ranked origins are increasingly adopting the Secure attribute. This trend becomes even more evident by focusing on the adoption of the ___Host- prefix. Despite numbers being still low, the popularity of ___Host- prefix is growing rapidly in the top 10K and top 100K ranges. Overall, 77 origins used the ___Host- prefix in 2021, in contrast to the 133 origins that used it in 2022, which corresponds to a 72% increase in one year. On the other hand, the distribution of nameless cookies is more stable over time and does not show a clear correlation with the website rank.

Table 5.4 provides a characterization of ___Host- and nameless cookies, showing the most common names and values, respectively, across the top 100K origins. Intuitively, the names adopted by ___Host- cookies suggest that they are used to store sensitive data such as session identifiers or CSRF tokens. Nameless cookies, instead, are likely to be the result of misconfigurations on the server side, since the most common values match cookie attribute identifiers. A manual analysis of the full collection of nameless cookies did not reveal any clear intended usage. To the best of our knowledge, our study is the first to measure the prevalence of nameless cookies in the wild. The results suggest that nameless cookies are a byproduct of misconfigurations and are not actively used by websites. For these reasons, we advocate for the removal of nameless cookies from the cookie standard and browsers to eradicate this source of confusion and the serialization collision vulnerabilities discussed in Section 5.4.2. Conversely, we believe that the increasing adoption of ___Host- cookies is a positive trend that should be further promoted among Web developers and security practitioners.

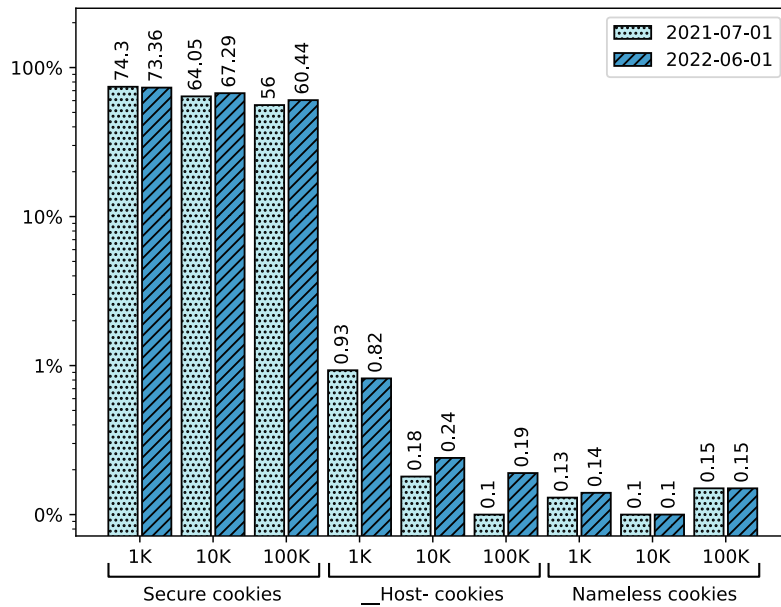


Figure 5.4: Deployment of cookies between 2021 and 2022.

| Host- cookie names | # | Nameless cookie values | # |
|---------------------------|----|---------------------------|----|
| Host-next-auth.csrf-token | 26 | HttpOnly | 50 |
| Host-GAPS | 23 | <empty string> | 16 |
| Host-csrf-token | 13 | Secure | 6 |
| Host-PHPSESSID | 10 | = | 5 |
| Host-SESSION_LEGACY | 5 | ACookieAvailableCrossSite | 4 |
| Host-SESSION | 5 | =0 | 3 |
| Host-sess | 4 | secure | 1 |
| Host-SWAFS | 3 | * | 1 |
| Host-session | 3 | ^(.*)\$ \$1 | 1 |
| Host-js_csrf | 3 | =1 | 1 |

Table 5.4: Top-10 Host- cookie names and nameless cookie values from 2022-06-01.

5.5 CORF Token Fixation

We present a class of attacks that we call *CORF Token Fixation* that undermine implementations of the synchronizer token pattern in the presence of network or same-site attackers. The synchronizer token pattern is considered a robust CSRF protection against the same-site threat model [OWAa] and is widely used in Web applications [LKP21]. However, as we show in Section 5.6, common implementations are vulnerable to CORF attacks. The term *CSRF Token Fixation* has been used in the past to refer to a vulnerability affecting the Devise authentication library [Val13]. Although this vulnerability is an instance of our attack class, we provide for the first time a precise characterization

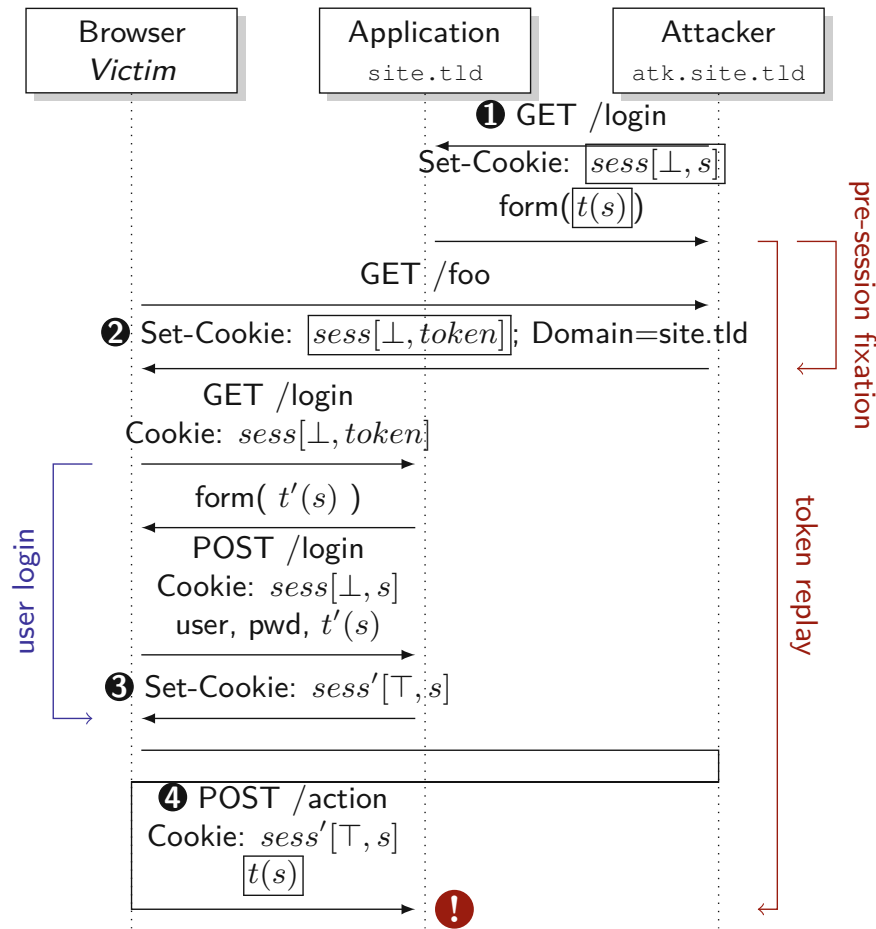


Figure 5.5: CORF token fixation attack (pre-login).

of the attack flow and discuss a more general instance of the problem. Moreover, by factorizing the attacks into *fixation* and *replay* phases, we show how known bypasses to the double submit pattern can be framed in this class.

5.5.1 Token Fixation Attacks

Figure 5.5 shows an instance of a token fixation attack (pre-login) that performs a state-changing request to a token-protected endpoint (`/action`). User sessions are represented as $sess[logged-in-status, csrf-secret]$, where $sess$ is the identifier for a session containing the *logged-in-status* and the *csrf-secret* value. Sessions can be stored on the server or the client side: in the first case, typically referred to as *stateful*, the cookie includes only the session identifier; in the latter, known as *stateless*, the content of the session is used as the cookie value, possibly after being encoded and signed. The attack flow is identical in both scenarios.

The attack has the following preconditions: (i) the target application uses the synchronizer token pattern, storing the CSRF secret in the session; (ii) the application constructs a pre-session for guest users (i.e., not logged-in) and has at least one CSRF token-protected form visible to guests. Alternatively, the CSRF token can be derived from information present in the pre-session. In the diagram, $t(s)$ represents the token that is attached to forms and derived (e.g., hashed or encoded) from the CSRF secret s ; (iii) the CSRF secret is shared unchanged between the pre-session and the session.

When these preconditions are satisfied, the attack is performed as follows: ❶ the attacker visits the target application and obtains the value of the pre-session cookie and the CSRF token that is bound to that pre-session; ❷ the attacker performs a pre-session fixation attack [Kol02], setting the victim pre-session cookie to the value previously obtained by the attacker; ❸ by logging into the application, the user has an authenticated session $sess'$ which shares the CSRF secret s with the attacker-known pre-session $sess$; ❹ the attacker causes the victim's browser to execute a crafted request towards the `/action` endpoint, attaching the value of the token $t(s)$ obtained in the first step. Given precondition (iii), the secret was preserved during the login process, so a valid token for the pre-session is accepted as a valid token for the authenticated session. This allows the attacker to perform a CORF attack that bypasses the CSRF token protection.

Note that the encoding/serialization mechanism used to derive a token from the secret s may generate different tokens ($t(s)$ and $t'(s)$ in the figure) for different requests, e.g., by including an expiration date. In such cases, a server could disallow expired tokens or only accept the last token that was generated. Still, an attacker could bypass this protection by executing again step ❶ before constructing the request ❹ to obtain a valid token. Furthermore, the attack can be performed even if the victim has an already established authenticated session with the website. Besides setting a more specific path in the injected cookie, as described in Section 5.4.1, the attacker can forcibly logout the victim from the website using a cookie eviction technique (see Section 5.4.1) before fixating the pre-session cookie.

Post-Login Variant. The double submit pattern typically stores the CSRF secret in a separate cookie from the session. Hence, overwriting/shadowing this cookie (fixation phase) is sufficient to perform the attack, assuming that the attacker subsequently crafts a request to the protected endpoint with a CSRF token that matches the value of the overwritten cookie (replay phase). Notice that this attack variant does not require fixating the pre-session, thus lowering the set of preconditions compared to the STP bypass. Additionally, the post-login attack can be commonly performed without prior knowledge of a valid CSRF token. Still, whenever the server performs additional validation checks, an attacker can obtain a valid cookie for the application and its related CSRF token and use them to carry out the attack.

5.5.2 Mitigations

Token fixation attacks are enabled by cookie integrity violations from network and same-site attackers. Hence, preventing cookie tossing from sibling domains, i.e., via the `__Host-` cookie prefix, would trivially prevent the attacker from executing the fixation phase (step ②). However, `__Host-` cookies may introduce compatibility issues on applications that use multiple origins. For instance, sharing the same session at `accounts.example.com`, where users log in, with the rest of the application at `example.com`, requires setting domain cookies.

Token Secret Refresh for STP. A robust mitigation for token fixation attacks for websites that implement the synchronizer token pattern consists in refreshing the value of the CSRF secret in the user session upon login. This update has the effect of using different secrets in the pre-session and in the authenticated session, so that precondition (iii) of the pre-login attack is no longer satisfied. This leads to the rejection of pre-session tokens in authenticated sessions and prevents the attacker from executing step ④ of Figure 5.5, since the token obtained at step ① is not valid for the new user session.

Mitigating Attacks Against DSP. In 2012, Wilander [Wil12] proposed a variation of the double submit pattern named *triple submit cookies* to address a specific version of the attack. The mechanism employ random identifiers for both the name and value of the cookie, attaching only the random value to forms, and leveraging `HttpOnly` cookies to not disclose the random name with client-side scripts. The server-side validation of the submitted token may require storing the random name in the user session (stateful triple submit), or enforcing that the request contains only a single cookie with a random name, discarding the request otherwise (stateless). The stateful variant is equivalent to a synchronizer token pattern, where the random name acts as the CSRF secret and is stored in the user session. The stateless variant relies on the assumption that cookies cannot be erased since, otherwise, the attacker can forge a request with a single random-name cookie [Lun13]. This assumption is only valid for Safari (see Section 5.4.1), thus the stateless triple submit is not effective in the general case. Consequently, the post-login attack can only be mitigated by (i) using `__Host-` prefix cookies, which are subject to compatibility issues, or (ii) switching to the synchronizer token pattern and refreshing the secret upon login.

5.6 Systematic Evaluation of Web Frameworks

We perform a study of Web development frameworks aimed at detecting session integrity vulnerabilities that may derive from the composition of security libraries, focusing on session management and CSRF protection components. In particular, we apply the threat models defined in Section 5.3 and leverage the techniques described in Section 5.4 to conduct the CORF token fixation attacks presented in Section 5.5. Albeit developers are ultimately responsible for securing their Web applications, we believe security abstractions should provide defaults that ensure safe composition. Hence we conducted the study on

5.6. Systematic Evaluation of Web Frameworks

| Framework | Lang. | Auth. Library | CSRF Library | CSRF Protection | | CORF Token Fixation | | Session Fixation |
|------------------------------|--------|---------------------------------|--------------------------------|-----------------|-----|---------------------|--------|------------------|
| | | | | STP | DSP | Pre-L | Post-L | |
| Express (4.18.1) [expa] | JS | passport (0.5.3) [Han] | csurf (1.11.0) [csu] | 🟡 | 🟡 | 🔴→🟢 | 🔴 | 🔴→🟢 |
| Koa (2.13.4) [koa] | JS | koa-passport (4.1.3) [koac] | csurf (3.0.8) [koab] | 🟡 | — | 🔴 | — | 🔴→🟢 |
| Sails (1.5.3) [saib] | JS | in cookies as in docs | csurf (1.10.0) [csu] | 🟡 | — | 🔴→🔴 | — | 🔴→🔴 |
| Fastify (4.13.0) [fasa] | JS | fastify/passport (2.2.0) [fasc] | csrf-protection (6.1.0) [fasb] | 🟡 | 🟡 | 🔴→🟢 | 🔴→🟢 | 🔴→🟢 |
| Django (3.2.13) [dja] | Python | built-in | built-in | 🟡 | 🟡 | 🟢 | 🔴 | 🟢 |
| Flask (2.1.2) [flaa] | Python | flask-login (0.6.1) [flab] | flask-wtf (1.0.1) [flac] | 🟡 | — | 🔴 | 🔴 | 🟢 |
| Tornado (6.2.0) [tor] | Python | in cookies as in docs | built-in | — | 🟡 | — | 🔴→🔴 | 🟢 |
| Laravel (9.1.5) [lar] | PHP | built-in | built-in | 🟡 | — | 🟢 | — | 🟢 |
| Symfony (5.4.19) [syma] | PHP | built-in | security-csrf (5.4.19) [symb] | 🟡 | — | 🔴→🟢 | — | 🟢 |
| CodeIgniter 4 (4.2.1) [ci4a] | PHP | shield (1.0.0-beta) [ci4b] | built-in | 🟡→🟡 | 🟡→— | 🔴→🟢 | 🔴→— | 🟢 |
| Yii2 (2.0.45) [yii] | PHP | built-in | built-in | 🟡 | 🟡 | 🟢 | 🔴 | 🟢 |
| ASP.NET Core (6.0.4) [dot] | C# | built-in | built-in | 🟡 | — | 🟢 | — | 🟢 |
| Spring (5.3.19) [spra] | Java | Spring Security (5.6.3) [sprb] | Spring Security (5.6.3) | 🟡 | — | 🟢 | — | 🟢 |

Table 5.5: Analyzed Web frameworks, and their respective authentication and CSRF libraries. 🟡 default, 🟢 available, 🟢 unaffected, 🔴 vulnerable, — not implemented. 🟢 safe (insecure options available), 🔴 vulnerable (secure options available).

the default settings enabled by each framework. Moreover, we discuss relevant opt-in options that are listed in the documentation and assess how they affect security. As part of our work, we responsibly performed a coordinated disclosure of all the identified issues.

Selection Criteria. The selection criteria for the analyzed Web development frameworks follow the approach adopted by Likaj et al. in their comprehensive study [LKP21]. First, we considered the top 5 languages used for Web development in 2022 according to [Git22], i.e., JS, Python, Java, C#, and PHP, and then selected the most used frameworks from this pool. For this purpose, we used the GitHub metrics *watch*, *fork*, and *stars*, collected on April 8, 2022. We then picked the top 10 of each category. This selection resulted in a total of 13 frameworks. We refer the reader to Appendix C.1 for the complete framework list and the associated GitHub metrics.

5.6.1 Frameworks Analysis Methodology

We conducted a manual security analysis to expose Web session integrity vulnerabilities in the selected frameworks. For each framework, we followed the official documentation to develop a toy application that includes a login form and a state-changing endpoint protected by a token-based CSRF mechanism. The login and CSRF functionalities were implemented using the official libraries provided by the framework. When official libraries were not available, we used external libraries that are widely used by the community, thus being considered the *de facto* standards. In two cases, we had to implement the session management functionality at the application level following the instructions provided in the documentation since no standard libraries were available. For each framework, we also developed an automated routine to simulate the attacker’s website and to mechanize the CORF token fixation attacks.

We performed a coordinated disclosure of the identified vulnerabilities, and assisted framework developers to understand the threat model and to implement appropriate solutions that would improve the baseline security of their frameworks. We focused

our disclosure on unsafe defaults, avoiding reports that would have been perceived by developers as potentially deceptive. For instance, we reported vulnerabilities on the double submit pattern only when this CSRF protection mechanism was set as default or it was the only one available. Double submit is indeed known to be vulnerable against same-site attackers, although it provides some protection against standard Web attackers.

Table 5.5 summarizes the results of our analysis categorizing each framework by language and including the selection of the libraries used to implement the login and CSRF functionalities, as well as the adopted CSRF protection mechanisms and the tested versions. The table also shows the outcome of our disclosure, denoted with an arrow symbol. Out of the 13 analyzed frameworks, we identified 12 supporting the synchronizer token pattern, among which 7 were found vulnerable to CORF token fixation attacks (pre-login). Furthermore, 6 frameworks implemented the double submit pattern, resulting vulnerable to the post-login attack variant. We also discovered 3 frameworks vulnerable to session fixation attacks, thus allowing an attacker to fully compromise the victim's account.

5.6.2 Synchronizer Token Pattern Bypasses

In the following, we present the security analysis of vulnerable real-world implementations of the synchronizer token pattern. All vulnerable frameworks, excluding CodeIgniter 4, failed to refresh the CSRF secret after a successful login, thus allowing an attacker to perform a CORF token fixation (pre-login) by reusing the CSRF token issued for the attacker's session following the steps described in Figure 5.5.

Passport-Based: Express, Koa, Fastify

Several frameworks based on Node.js integrate with the Passport authentication middleware to support authenticated user sessions. Express natively integrates with Passport, Koa requires an additional Passport middleware (koa-passport), and Fastify provides its own port of Passport (fastify-passport). The CSRF protection is implemented by the csrf CSRF token middleware in Express, while Koa uses a different middleware called koa-csrf; Fastify, instead, provides CSRF protection via the csrf-protection plugin. All implementations support the synchronizer token pattern with the CSRF secret being stored in the session object. The login and user validation functions are performed by the `authenticate` function of Passport (and `fastify-passport`). We discovered that this function does not clear, nor reinitializes, the attributes in the session object other than those specific to Passport, e.g., the `passport` attribute. Hence, the session attribute `csrfSecret` (`secret` in Fastify) is not renewed upon successful authentication, satisfying the condition (iii) of our attack. Consequently, CSRF tokens issued to the attacker during the pre-session fixation step can be used to forge CORF requests after the victim authenticates on applications developed using these frameworks.

Disclosure. We reported this issue to the Passport developer, who promptly fixed it in version 0.6.0 by clearing all attributes from the session object after login, effectively

solving the vulnerability on Express. However, for backward compatibility, Passport 0.6.0 supports the `keepSessionInfo` option that enables Web developers to opt out from the new safe behavior, and preserve the session attributes between pre-sessions and authenticated sessions. This option is set to `false` by default. CVE-2022-25896 was issued for this vulnerability. Fastify developers promptly fixed the vulnerability in version 2.3.0 by clearing all attributes from the session object after login and assigned CVE-2023-29020 to this vulnerability. The release also introduced support to the `clearSessionIgnoreFields` option that enables Web developers to define a set of session attributes to be preserved between pre-sessions and authenticated sessions. On the other hand, the new version of Koa middleware (6.0.0, published on February 2023) does not benefit from the best practices implemented in Passport 0.6.0 and remains vulnerable. We are currently in touch with the developers to identify an effective mitigation.

Symfony

Symfony provides user management natively and relies on the official library `security-csrf` for CSRF protection. Symfony supports three different ways to handle session identifiers and session content while authenticating users, called *strategies*. The default strategy (`MIGRATE`) regenerates the session identifier upon login, but preserves the remaining session attributes. As the CSRF secret is not refreshed, the framework is vulnerable to the pre-login CORF token fixation attack. One specificity of Symfony is that the granularity of the CSRF mechanism can be configured to support distinct CSRF secrets depending on the endpoint. In this case, the pre-login attack still succeeds against all endpoints where it is possible to obtain a valid CSRF token under a pre-login session. The attacker simply needs to execute step ❶ towards all these endpoints to populate a pre-session with the corresponding CSRF secrets before executing step ❷.

Disclosure. This vulnerability was reported to the Symfony developers who updated the `MIGRATE` strategy to clear the CSRF storage in new versions of the library (v4.4.50, v5.4.20, v6.0.20, v6.1.12, v6.2.6). We stress that the two other strategies are either insecure or could introduce compatibility problems on websites based on Symfony: `NONE` preserves the same session after authentication, leading to session-fixation attacks, whereas `INVALIDATE` regenerates the session identifier and deletes all other attributes in the session. CVE-2022-24895 was issued after our disclosure.

Sails

Sails does not implement a login handler function, however it ships with a generator [saia] that bootstraps a template application providing a user-management service based on `express-session` [expb]. Sails can be configured to enable CSRF protection out of the box via the `csrf` library. Given that the user-management logic is hard-coded at the application level and that the session object is not refreshed upon login, any token generated before authentication is still valid after the user authenticates, thus satisfying

the precondition (iii) of the attack. We expect Web developers to build their applications starting from the generated template application. For this reason, we consider this unsafe code pattern to be likely inherited by real-world websites.

Disclosure. The unsafe code pattern was reported to the Sails development team. As a result, a new version of the generator was released (2.0.7) with support for `__Host-` cookie prefixes in production mode (non-default). Using a `__Host-` cookie for the session addresses the vulnerability, although Web developers must be aware of cookie scope restrictions that may hamper the deployment of the protection, as discussed in Section 5.5.2.

Flask

Flask-based applications supporting user authentication often rely on the Flask-Login library for session management and Flask-WTF to provide CSRF protection using WTForms [wtf]. Login and user validation are performed by the `login_user` function that, similarly to Passport, does not clear nor reinitialize the attributes in the session object other than those specific for Flask-Login, thus satisfying precondition (iii) of the attack.

Disclosure. This vulnerability was disclosed to the developers of Flask and Flask-login, proposing a fix that would allow developers to define a set of opt-in attributes to be preserved upon login and to clear all others. Given that the two libraries operate separately, developers proposed instead to clear all attributes from the session and let application developers explicitly copy the attributes that should be preserved. A pull request for this issue is still open.

CodeIgniter 4

CodeIgniter 4 provides user management via the (official) library Shield [ci4b], while CSRF protection is included natively and can be easily enabled. CodeIgniter 4 offers the synchronizer token pattern and double submit as CSRF protections, with the latter being the default option. For both mechanisms, the framework supports the option to regenerate the CSRF secret upon each CSRF-protected action (default), or to preserve the secret per session, via the option `security.regenerate = true` and `false` respectively. Similarly to the previous cases, CodeIgniter 4 is vulnerable to the CORF token fixation (pre-login) attack when the CSRF secret is not refreshed at login. However, we discovered that CodeIgniter 4 is also vulnerable when the CSRF secret is regenerated at login via the `security.regenerate = true` setting.

CodeIgniter 4 sessions objects are stored on the server and contain CSRF secrets as attributes called `csrf_test_name`. When a user accesses the application, a session object `sess` is created with secret `s`, and upon login, a new session `sess'` is created with secret `s'`. However, while creating `sess'`, the attribute `csrf_test_name` of `sess` is also updated to `s'`. Thus, the attack illustrated in Figure 5.5 is still possible as the attacker, knowing `sess`,

can perform an additional request between steps ❸ and ❹ to, e.g., `/login`, and obtain a fresh token $t'(s')$ that is valid for both the pre-session *sess* and the authenticated session *sess'*.

Disclosure. This vulnerability was communicated to the developers of Shield, who released a new fixed version of the library (1.0.0-beta.2) that (i) always refreshes the CSRF secrets at login, (ii) deletes pre-sessions upon login, and (iii) discontinues the double submit pattern in combination with Shield. CVE-2022-35943 was issued for this vulnerability.

5.6.3 Double Submit Pattern Issues

All analyzed frameworks implementing the double submit pattern were vulnerable to CORF token fixation attacks (post-login). Although this pattern is known to enable same-site attackers to bypass CSRF protections, our study aimed at identifying if any of the frameworks was applying mitigations such as the `__Host-` cookie prefix. We concluded that none of the frameworks applied the above mitigation. Fastify tried to mitigate this attack by including information related to the logged-in user in the CSRF token in order to prevent cookie tossing. It turns out that the attack was still possible if the `userInfo` associated with the target was predictable.

Disclosure. As discussed in Section 5.6.1, we did not contact developers of frameworks that were already applying safe defaults (Express) or were already aware of the risks associated with the double submit pattern (Django). The other vulnerabilities were communicated to the developers of the 4 remaining frameworks. Fastify addressed the vulnerability by performing an HMAC of the `userInfo` in order to prevent cookie tossing. CVE-2023-27495 was issued for this vulnerability. The CodeIgniter 4 Shield library disallowed the combination with the double submit pattern, relying now only on the synchronizer token pattern as a more robust CSRF protection. Tornado added optional support for the `__Host-` prefix to the CSRF cookie in version 6.3.0⁴. Yii2 developers initially replied to our disclosure but, to the best of our knowledge, did not follow up on the issue.

5.6.4 Session Fixation Vulnerabilities

We also found 3 frameworks vulnerable to session fixation attacks. Session fixation attacks happen when pre-session cookies are preserved after authentication, thus allowing an attacker to hijack the session of an authenticated user violating its confidentiality and integrity [Kol02]. The attack flow is the following: (i) the attacker obtains an unauthenticated session cookie `session_cookie=S` by visiting `https://example.com`; (ii) the victim is lured into visiting `https://atk.example.com` that sets a domain cookie for `https://example.com/` in the victim's browser, such that `session_cookie=S`; (iii) the victim authenticates on `https://example.com/`; (iv) the attacker uses the session cookie

⁴<https://www.tornadoweb.org/en/stable/releases.html>

`session_cookie=S` to hijack the victim's session at `https://example.com/`. Notice that regenerating the session cookie prevents session fixation, but it is not enough to mitigate CORF token fixation attacks if CSRF secret values still propagate unchanged to the authenticated session.

Passport

We identified a session fixation vulnerability in Passport stemming from the fact that the session attribute `sessionId` of the pre-session was not cleared nor reinitialized upon login, but rather preserved after user authentication.

Disclosure. This vulnerability was disclosed to the developers of the Passport library and was fixed in version 0.6.0 using the `Session.regenerate` method of the `express-session` module to generate a new `sessionId` after a successful login. CVE-2022-25896 was issued for this vulnerability.

Fastify

A session fixation attack similar to the one in Passport was also identified in Fastify when using the `fastify/session` plugin as the underlying session management mechanism (stateful).

Disclosure. This vulnerability was disclosed to the developers of `fastify-passport` and was fixed in version 2.3.0 using the `session.regenerate` method of `fastify/session` to generate a new `sessionId` after a successful login. CVE-2022-29019 was issued for this vulnerability.

Sails

A session fixation attack similar to the one in Passport was also identified in Sails. We recall that, although Sails does not implement a native login interface, it provides an application template that bootstraps a project. Consequently, unsafe code patterns embedded in the application template could be inherited by real-world websites.

Disclosure. This unsafe code pattern was disclosed to the Sails team. No particular action was taken to mitigate this unsafe pattern in the template application, although the addition of the optional `__Host-sails.sid` in production mode described before mitigates the impact of this attack.

5.7 Formal Verification of Web Frameworks

We complement the analysis of the top Web frameworks (Section 5.6) with the formalization of their session management mechanism and CSRF protections. The goal of our formalization is to verify the correctness of the mitigation to vulnerable synchronizer token patterns, i.e., the CSRF secret refresh discussed in Section 5.5.2. To this end, we

use the WebSpi [BBM12] library for the ProVerif [Bla01] protocol verifier, which enables automated security proofs for Web applications.

Our formalization focuses on the 7 frameworks that are vulnerable to the pre-login token fixation attack and resulted in 4 different framework models that differ depending on whether the session is stored on the client or the server side, and on implementation details of the synchronizer token pattern adopted by the framework. This is the case since most JavaScript frameworks share the user management mechanism based on the Passport library, and, for instance, Express and Sails implement CSRF protection with the csrf library. The framework models implement a common API used by a generic application model to implement login and protected form elements. The application is run in parallel with a powerful same-site attacker that can overwrite any cookie on its sibling domains, independently from path or flags/prefixes. This attacker model over-approximates the threat models in Section 5.3, essentially considering cookies with no integrity and resulting more powerful than **SS-HOST-S**. This over-approximation ensures stronger security proofs, which are valid irrespective of integrity assumptions on cookies.

A CSRF attack results from an unauthorized authenticated request to a protected endpoint performed by the attacker, thus we define our expected security property as follows.

Invariant. Every action executed by a token-protected endpoint must be explicitly initiated by an honest user by performing a request containing the token.

We encode the invariant as a *correspondence assertion* [WL93] between the two events (i) *app-action-successful*, that happens when the server successfully validates the CSRF token and performs the token-protected state-changing action, and (ii) *app-action-begin*, that happens when the honest user submits the form that contains the CSRF token.

$$\forall(c : \text{Cookie})(b : \text{Browser})(token : \text{CSRFToken}). \\ \text{event}(\text{app-action-successful}(c, token)) \Rightarrow \text{event}(\text{app-action-begin}(b, token))$$

Intuitively, the correspondence requires that every instance of the *app-action-successful* event must be preceded by the *app-action-begin* event. This property explicitly forbids execution traces where the attacker successfully executes the protected action without the honest user submitting the form.

ProVerif confirms that the property does not hold on any of the four models, producing counterexamples that closely resemble the token fixation attack of Figure 5.5. We then update the models to include the token refresh mitigation, i.e., generate a new CSRF secret upon user login (Section 5.5.2). Additionally, we refresh the session identifier on the model for Sails, Express, and Fastify (see session fixation attacks, Section 5.6.4). With these modifications, we obtain four fixed models for which ProVerif proves that our correspondence property is valid. Notice that in the presence of a session fixation attack, refreshing the CSRF secret is not enough for the property to hold, as the attacker can perform a full session hijacking attack and execute the token-protected action.

| $P, Q :=$ | Processes |
|---|--|
| 0 | null process |
| $P \mid Q$ | parallel composition |
| $!P$ | replication |
| $\nu n.P$ | name restriction |
| $c(x).P$ | message input from channel c |
| $\bar{c}\langle M \rangle.P$ | message output to channel c |
| if M then P else Q | conditional (else is optional when Q is 0) |
| let $x = M$ in P | term evaluation, local variable definition |
| insert $(d(M_1, \dots, M_n)).P$ | insert record into table d |
| get $(d(M_1, \dots, M_n)).P$ | read record from table d |
| $R(M_1, \dots, M_2)$ | macro usage |

Table 5.6: Syntax of processes.

This analysis shows that refreshing the CSRF secret upon login makes the synchronizer token pattern a robust mitigation for CORF attacks, even in presence of same-site or network attackers who can fully compromise cookie integrity.

5.7.1 Formal Model of Web Frameworks

In the following, we describe our modeling of Web framework using the applied pi calculus [AF01], introducing only the parts of the formalism relevant for the understanding of the model. In particular, Table 5.6 reports the syntax for processes, which, in addition to the standard (simple) processes, includes tables (or databases) operations for storing (*insert*) and retrieving (*get*) persistent data [BSCS]. Additionally, we use an extended term language that supports functions and records, to ease the definition of the framework API.

We define a framework as a set of functions and type definition for handling user management and CSRF protection.

$$\begin{aligned}
 \text{Framework} := \{ \quad & \textbf{type } Session \\
 & \text{empty-session} : unit \rightarrow Session \\
 & \text{valid-session} : Session \rightarrow bool \\
 & \text{session-from-cookie} : Cookie \rightarrow Session \\
 & \text{session-to-cookie} : Session \rightarrow Cookie \\
 & \text{is-loggedin} : Session \rightarrow bool \\
 & \text{login-user} : (Session, User) \rightarrow Session \\
 & \text{logout-user} : Session \rightarrow Session \\
 & \text{generate-token} : Session \rightarrow Session \\
 & \text{serialize-token} : Session \rightarrow CSRFToken \\
 & \text{validate-token} : (CSRFToken, Session) \rightarrow Bool \quad \}
 \end{aligned}$$

In particular, each framework model provides definitions for:

Session. The implementation for the *Session* abstract data type and the functions to construct, validate and serialize such session to a cookie: *empty-session*, *valid-session*, *session-from-cookie*, *cookie-of-session*. The session could be stored in the server or client side: in the first case, *session-from-cookie* fetches from the server storage the session corresponding to the session id in the cookie, in the second case, the function decodes the cookie value into a session that can be later validated with *valid-session*. Similar considerations apply for the inverse operation.

User management. The implementation of the user management functions such as *is-loggedin*, *login-user*, *logout-user*. Such function modify the current session with the status of the user if, e.g., the credentials are valid.

CSRF protection. The implementation of the generation and validation of CSRF tokens: *generate-token*, *serialize-token*, *validate-token*. These function may rely on implementation specific-types for the CSRF secret that can be stored in the session. The *generate-token* function generates the CSRF secret, possibly updating the session; similarly, the validation and serialization of the CSRF token may depend on the secret stored in the session (see synchronizer pattern, Section 5.2.3)

The functions that are part of the *Framework* API are used in a generic model of a Web application composed of 3 endpoints, */login*, */logout*, */action*, supporting multiple HTTP methods. The */login* and */action* endpoints return a CSRF token protected form for GET requests, and execute the corresponding action, e.g., checking user credentials after validating the token, for POST requests. The */logout* endpoint only supports POST requests. We model the application as an applied pi calculus process, consisting of the parallel composition of 5 *handler processes*, one for each combination of endpoint and HTTP method. We discuss in the following the definition of the application, focusing on the process handling POST requests to the */action* endpoint.

$$\begin{aligned}
 &App(Host, \{session-from-cookie, is-loggedin, validate-token, \dots\}) := \dots \mid \\
 &\quad !(httpServerRequest(((https://, Host, /action), headers, POST(token), c)). \\
 &\quad \quad \text{let } s = session-from-cookie(cookie(headers)) \text{ in} \\
 &\quad \quad \text{if } valid-session(s) \wedge is-loggedin(s) \wedge validate-token(s, token) \text{ then} \\
 &\quad \quad \text{event}(app-action-successful(cookie(headers), token)). \\
 &\quad \quad \overline{httpServerResponse}(((https://, Host, /action), httpOk, c)))
 \end{aligned}$$

The *App* process takes as parameter the host in which the application is running and an instance of a framework. The */action* handler reads from the *httpServerRequest* channel, provided by the WebSpi [BBM12] library, a POST requests to the URL composed of the https scheme, the application host and the */action* path. The obtained request includes the HTTP *headers* and the submitted CSRF *token*. The handler then builds a session from the received cookie header using the framework function *session-from-cookie*. When the session is valid, and belongs to a logged-in user, the CSRF *token* is validated with the *validate-token* function, executing the event *app-action-successful* upon successful

validation. The process then returns to the client the *httpOk* response by writing it to the *httpServerResponse* channel.

The generic application model *App* is run on the `example.com` host in parallel with a same-site attacker *Attacker*, hosted on the sibling domain `attacker.example.com`. Additionally, the application is visited by an unbounded number of distinct registered users, modeled by the *UA* process. We use the *Users* table to store and validate registered users for the *App* application.

$$\begin{aligned} & \text{Browser} \mid \text{HttpServer} \mid \\ & \text{Attacker}(\text{attacker.example.com}) \mid \\ & \text{App}(\text{example.com}, \text{Framework}) \mid \\ & !(\nu id).(\nu pwd).\text{insert}(\text{Users}(id, pwd)).\text{UA}(id, pwd) \end{aligned}$$

The *Browser* and *HttpServer* processes are part of the WebSpi library and provide browser and server functionality, for example encrypting/encoding and decrypting/decoding requests from the network and sending them to the correct application on the *httpServerRequest* and *httpServerResponse*, or attaching cookies to browser requests.

The process modeling users *UA* follows a similar structure to the *App* process, consisting in the parallel composition of all possible user actions. We focus on the sub-process modeling the submission of the token-protected form.

$$\begin{aligned} \text{UA}(id, pwd) := & (\nu b : \text{Browser}).(\dots \mid \\ & \text{newPage}_b((p, \text{example.com/action}, \text{form}(\text{token}))). \\ & \text{event}(\text{app-action-begin}(b, \text{token})). \\ & \overline{\text{pageClick}_b}(\langle p, \text{example.com/action}, \text{POST}(\text{token}) \rangle)) \end{aligned}$$

The process reads from the *newPage_b* channel, which is defined in WebSpi and models the loading of a page in the user (private) browser. The obtained page from `example.com/action` includes a form containing a CSRF *token*. The process then executes the event *app-action-begin* before submitting the form (*pageClick_b*), performing a POST request which includes the *token*. This event is only performed by the user modeled by the *UA* process before the (explicit) form submission, so it will not be present in traces in which the attacker performs a CSRF.

5.8 Related Work

Several studies have focused on cookie integrity issues, with a particular emphasis on session integrity [BBC11, ZJL⁺15, NMY⁺11, BCFK15, CRB19, CFG⁺20]. In their seminal work, Bortz et al.[BBC11] introduce the related-domain attacker model and propose a mechanism, named *origin cookies*, to bind cookies to specific origins. The `__Host-` prefix builds on this proposal and has been integrated into modern browsers [CEWW22]. Other studies suggest browser extension, e.g., to transparently strip session (cookie) identifiers

from network requests to avoid session hijacking [NMY⁺11, BCFK15]; Calzavara et al. [CFG⁺20] focus on the server-side by proposing a type system for verifying session integrity of PHP code against a variety of attackers, including network and related domain attackers. These works, except for [ZJL⁺15], do not assess the implications of the lack of cookie integrity for real world application. Zheng et al. [ZJL⁺15] present an empirical assessment of cookie injection attacks on the Web, taking into account both browser-side and server-side cookie handling inconsistencies, and discovering attacks on popular Web sites (e.g., Google, Amazon). Similarly to our work, the authors discover browser implementation differences in storage limits for cookies and cookie ordering in requests, and inconsistencies in server-side languages such as the automatic percent decoding of cookie names in PHP. Our findings uncover that, even after seven years, these types of cross-browser/language inconsistencies are still relevant and also affect newly introduced security mechanisms such as `__Host-` prefix cookies.

Recently, Squarcina et al. [STV⁺21] measured and quantified the threats posed by same-site attackers to Web application security. In their study of cookies, they discovered that the majority of the cookies on sites vulnerable to subdomain takeover has no integrity against related domain attackers. The authors highlight that the `__Host-` prefix was used only once in their dataset. Our measurement (Section 5.4.4) confirms the infrequent usage of the prefix in the wild, but shows a promising positive trend on its adoption in the last 2 years, especially on lower-ranked websites. Sanchez-Rola et al. [SRDB⁺21] performed a large-scale measurement to characterize cookie-based Web tracking. The study shows that third-party script inclusion enables cookie sharing in the context of first-party cookies, thus enabling third parties to set cookies on behalf of the visited website. Additionally, the authors report on instances of *cookie collisions*, where scripts from different actors in the same website access cookies created with the same name but different semantics. This setting matches our **SS-XSS-S** threat model (Section 5.3), where different parties gain control of a domain on a page served via HTTPS. However, unlike the study of Sanchez-Rola et al., which does not consider domain cookies, we focus on cookie integrity violations from attacker-controlled subdomains.

Concerning the analysis of Web frameworks, Likaj et al. [LKP21] evaluated the mechanisms implemented by major Web frameworks, quantifying their exposure to CSRF attacks as a result of implementation mistakes, cryptography-related flaws, cookie integrity violations, or leakage of CSRF tokens. The authors discover that 37 out of 44 frameworks are affected by such issues. Our analysis of Web frameworks (Section 5.6) shows that further implementation issues in the synchronizer token pattern (deemed secure in [LKP21]), originating from the composition of different libraries, lead to a bypass of the protection in the presence of same-site attackers. For instance, the CORF token fixation attack sidesteps the Flask framework protection, which was considered secure in previous work.

5.9 Conclusion

This study is a modern look at cookie integrity issues and their impact on Web application security. Our research showed that integrity vulnerabilities are not limited to implementation bugs, but are a pervasive threat across the Web due to compositionality problems at multiple levels. We engaged with browser vendors, the IETF HTTP Working Group, and Web framework developers to address the discovered issues, which resulted in several high-impact updates, e.g., Chrome and Firefox, PHP (the server-side language powering 78% of all websites), major authentication libraries such as Passport (2M weekly downloads), and the cookie standard [CEWW22].

Conclusion and Directions for Future Research

6.1 Conclusion

In this thesis, we have shown the importance of a rigorous and formal definition of Web security in terms of invariants that are guaranteed to be valid across the Web platform. We have proposed two methodologies for validating Web invariants on a model of Web specifications and on browser implementations that allowed us to discover new inconsistencies and propose sound mitigations. We studied the lesser known Web threat model of the related domain attacker, studying its impact on application security and focused on cookies, discovering new integrity violations.

More specifically, we presented WebSpec, the most comprehensive formal model of the Web browser in terms of supported security mechanisms that allows for automated bug-finding and machine-checked security proofs. The WebSpec verification pipeline allowed us to discover a new attack on cookies, caused by the interaction with legacy APIs and a new inconsistency in a planned modification to the HTML standard.

We then presented a lightweight methodology for automatically detecting security flaws in browser implementations that leverages the WPT test suite to obtain execution traces that are validated against Web invariants. Our verification methodology discovered 10 new attacks against Chromium, Firefox and Safari.

We presented the results of a large-scale measurement study on subdomain takeover on the top 50K domains, discovering takeover vulnerabilities in 887 sites. We analyzed the Web security implications of such vulnerabilities in terms of cookie integrity and confidentiality, CSP and CORS deployment, showing that subdomain takeover offer an additional advantage to the related-domain attacker compared to the traditional Web attacker.

We finally studied the issues that affect the integrity of cookies and presented new attacks originating from undefined behavior in the specifications or implementation flaws. We defined and presented the new class of CORF token fixation attacks that allows related-domain attackers to bypass token-based CSRF protections and proposed a formally verified mitigation.

6.2 Directions for Future Work

WPT-based specification mining. The translation of browser execution traces to SMT-LIB that we presented in Chapter 3 may be further generalized so that multiple execution traces, as opposed to a single one, are extracted from a single execution of the browser. This process would allow to abstract the specific details of single tests (e.g., the specific constants/URLs used by the test suite) and obtain a logic formula representing the property that the test aims to validate. Thus, the first research direction tackles the problem of automatically extracting specifications from the WPT cross-browser testing suite. This extraction methodology builds on the browser instrumentation infrastructure we developed for trace verification and will be composed of multiple phases. Starting from a browser execution trace of a WPT test, we aim to extract the property the test verifies. We first start with *naming*, that is, abstracting the concrete values that the execution trace is composed of (e.g., setting a cookie named `secure_session` to value 1) to \forall -quantified variables. This abstraction step allows us to convert a concrete test trace to a family of traces that share the same structure but can use, e.g., any cookie name. In the second phase, the abstracted trace, which is composed of sequentially-executed events, is further abstracted so that strict-sequentiality (i.e., two events must follow each other exactly) of events is relaxed, when possible, to allow additional arbitrary events to happen between the ones composing the test trace. This step, which we call *sequentiality abstraction*, will convert execution traces to logical formulas (e.g., using LTL) describing the ordering requirements of events. Once we obtain an abstracted formula, we can additionally abstract *common testing patterns* so that, e.g., matching a specific regex is abstracted into checking if a string is a prefix of another. To retain soundness, this process requires human intervention as the last phase, to approve or edit the generated formula to match the meaning of the test. We are currently working on sequentiality abstraction and have a prototype implementation of the naming phase that can be applied to WPT tests on cookies.

Invariant-directed browser fuzzing. The verification of invariants on browser execution traces presented in Chapter 3 has the potential for forming the basis of a fuzzing schedule for browsers that guides test generation with the goal of violating an invariant. Instead of relying on code coverage information, mutating test cases to improve the coverage of exercised code, an invariant-aware fuzzer may decide to mutate a test case so that more sub-propositions of a Web invariant are true. Web invariants are defined in the form of implications $(H_1 \wedge \dots \wedge H_n) \Rightarrow (C_1 \vee \dots \vee C_m)$. When the hypotheses are not valid, the invariant is trivially true. Only when all hypotheses are valid, the

conclusion are checked against the browser execution trace. A fuzzer that maximizes the amount of valid hypotheses for each of the considered invariants would discover violations with higher-probability, as the tested traces are more likely not to be trivially true. This fuzzing methodology can be complemented by the specification mining approach described earlier. An abstracted version of a test case (i.e., a test specification), where the concrete constant used during execution are replaced with \forall -quantified variables, represents a template that, when used for test generation, guarantees that all derived tests preserve the semantics of the original test.

Typing-Based verification of Web invariants. The Z3-based verification of Web invariants presented in Chapter 2 does not currently allow for the separated verification of modules, i.e., the entire model needs to be converted to a logical formula and verified with Z3. For this reason, different verification strategies could complement the Z3-based analysis and improve on the performance of Web invariant verification. One such approach is based on typing and specifying every Web components as a separate module. A type-based approach allows us to prove Web invariants at the module level and check for composition issues only across module boundaries, resulting in a dramatic performance improvement since each module needs to be verified in isolation only once. We are currently exploring solutions based on Information Flow Control (IFC) type systems, and in particular, we are working on embedding an IFC type system in the Coq language using the RUSSELL [Soz07] refinement-typing extension. This work includes a reimplementation of WebSpec with a focus on modularity and a clear separation of type-level specification (used for verification) and run-time behavior (modeling browser code). The resulting model will highlight the minimum set of run-time security checks that are required for Web invariants to retain validity. Moreover, it may enable the use of the rich type-level specifications for executable code (e.g., Rust) of real browsers (e.g., Servo).

Formal analysis of cross-context interactions. As an application platform, the Web represents an abstraction of the underlying operating system where the Web browser is executed. This abstraction, however, does not always prevent the hardware resources or operating system constraints (i.e., the context where the browser is running) to affect Web applications. For instance, the site isolation mechanism of Chrome and Firefox, that isolates every site into a separate process, offers less security on mobile devices, where hardware constraints force the browser to isolate only a portion of the sites, running all other in the same process. Similarly, extensions of the Web platform allowing to bridge native OS features to Web applications may violate Web security invariants as we have shown in our recent IEEE S&P publication [BSVL24]. For this reason, we argue that the composition of the Web platform and its context must undergo formal analysis to the same extent of the individual Web platform components. This formalization would clarify the security assumption that each Web mechanism and thus the platform as a whole make in its interaction with the system it is embedded into.

List of Figures

| | | |
|-----|---|-----|
| 1.1 | Research threads and questions | 3 |
| 1.2 | Overall Methodology | 5 |
| 2.1 | The WebSpec framework | 13 |
| 2.2 | Browser State and overview of its components (\rightarrow : referenced by): rendering components are on the left of the state, networking on the right | 14 |
| 2.3 | DOM Path Datatype | 26 |
| 2.4 | Host Cookies Inconsistency | 27 |
| 2.5 | Trusted Types bypass with same-origin iframes | 28 |
| 2.6 | CSP bypass due to inheritance from the embedder document | 31 |
| 3.1 | Syntax of traces: event types. | 46 |
| 3.2 | Trace Verification Pipeline. | 53 |
| 4.1 | Summary of related-domain attacker instances for dangling DNS records. | 75 |
| 4.2 | Vulnerability scanning pipeline. | 84 |
| 4.3 | Characterization of vulnerable domains. | 92 |
| 5.1 | Taxonomy of threat models for cookie integrity violations. | 110 |
| 5.2 | Cookie tossing attack. | 112 |
| 5.3 | Host- cookie bypass via nameless cookies. | 113 |
| 5.4 | Deployment of cookies between 2021 and 2022. | 119 |
| 5.5 | CORF token fixation attack (pre-login). | 120 |
| A.1 | HttpOnly Inconsistency | 163 |
| A.2 | Origin Header Inconsistency | 164 |
| A.3 | Authorization of non-simple requests (<i>i</i>): cross-origin redirection of form-generated PUT request | 166 |
| A.4 | Authorization of non-simple requests (<i>ii</i>): cross-origin redirection of pre-flight request | 167 |
| A.5 | CSP Inconsistency | 169 |
| A.6 | Service Workers Cache Inconsistency | 171 |
| A.7 | WPT test corresponding to the Trusted Types attack trace (Figure 2.5) | 176 |

List of Tables

| | | |
|-----|--|-----|
| 2.1 | Web Invariants. | 24 |
| 2.2 | Trace size and solving time for each attack. | 33 |
| 3.1 | Web Invariants | 44 |
| 3.2 | Design space analysis: comparison of browser instrumentation methods. (↑ : High, →: Medium, ↓ : Low) | 55 |
| 3.3 | Trace verification results. | 60 |
| 3.4 | Trace verification execution times. | 61 |
| 3.5 | Aggregated SAT results. (⚡: attack; ✖: false positive; ★: the same trace may contain multiple attacks) | 62 |
| 3.6 | Additional tests and new violations. | 68 |
| 4.1 | Main DNS record types. | 74 |
| 4.2 | Capabilities of the related-domain attacker. | 76 |
| 4.3 | Breakdown of the results in terms of affected domains/sites. | 91 |
| 4.4 | Attackers' capabilities on vulnerable services. | 93 |
| 4.5 | PSL on dynamic DNS services. | 95 |
| 4.6 | Web security abuses by related-domain attackers (RDA). | 95 |
| 5.1 | Capabilities required to set cookies in the victim's browser from a sibling domain of the target (w_a). | 109 |
| 5.2 | Evaluation of cookie integrity attacks against the cookie standard RFC6265bis-11 and browsers: Chrome (v109), Firefox (v109), and Safari (v16.0). (✔ compliant, ✖ violation, ✓ unaffected, ⚡ vulnerable, — does not apply. | 111 |
| 5.3 | Number of origins from the 2022-06-01 dataset setting cookies, and the percentage of origins using the Secure attribute, cookie prefixes, and nameless cookies. | 118 |
| 5.4 | Top-10 ___Host- cookie names and nameless cookie values from 2022-06-01. | 119 |
| 5.5 | Analyzed Web frameworks, and their respective authentication and CSRF libraries. (⬢ default, ⬢ available, ✓ unaffected, ⚡ vulnerable, — not implemented. (✔ safe (insecure options available), ⚡ vulnerable (secure options available)). | 123 |
| 5.6 | Syntax of processes. | 130 |
| A.1 | Solving time of inv. I.5 for progressively more complex models. | 175 |
| | | 141 |

| | | |
|-----|--|-----|
| A.2 | Comparison of supported Web components in existing models. | 179 |
| B.1 | Considered WPT tests. tal: 24896, WPT Version: d888ebb | 184 |
| C.1 | Web development frameworks from [LKP21] ranked according to GitHub metrics as of April 8, 2022. | 186 |

Bibliography

- [ABC⁺19] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth Schoen, and Brad Warren. Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web. In *CCS*, 2019.
- [ABL⁺10] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *CSF*, 2010.
- [Abu20] Abusix. Abuse Contact Database. <https://www.abusix.com/contactdb>, 2020.
- [AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, 2001.
- [AFG⁺11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *CPP*, 2011.
- [AJPN15] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven Months' Worth of Mistakes: A Longitudinal Study of Typosquatting Abuse. In *NDSS*, 2015.
- [Ang22] Angular. HTTP: Security - XSRF Protection. <https://angular.io/guide/http#security-xsrf-protection>, 2022.
- [Arc] HTTP Archive. The HTTP Archive. <https://httparchive.org/>.
- [ATW⁺20] Eihal Alowaisheq, Siyuan Tang, Zhihao Wang, Fatemah Alharbi, Xiaojing Liao, and XiaoFeng Wang. Zombie Awakening: Stealthy Hijacking of Active Domains Through DNS Hosting Referral. In *CCS*, 2020.
- [Bar92] Hendrik Pieter Barendregt. *Lambda Calculi with Types*. 1992.
- [Bar11a] A. Barth. HTTP State Management Mechanism. RFC 6265, Internet Engineering Task Force, 4 2011.

- [Bar11b] A. Barth. The Web Origin Concept. RFC 6454, IETF, 12 2011.
- [BBC11] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin Cookies: Session Integrity for Web Applications. In *W2SP*, 2011.
- [BDDM14] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *Journal of Computer Security*, 2014.
- [BBM12] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *CSF*. IEEE, 2012.
- [BCF⁺14] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan, and Mauro Tempesta. Provably sound browser-based enforcement of web session integrity. In *CSF*, 2014.
- [BCFK15] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. CookiExt: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 2015.
- [BFH⁺18] Kevin Borgolte, Tobias Fiebig, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates. In *NDSS*, 2018.
- [bHA22] Web Almanac by HTTP Archive. Http archive’s annual state of the web report. <https://almanac.httparchive.org/>, 2022.
- [Bia15] Nick Biasini. Threat Spotlight: Angler Lurking in the Domain Shadows. <http://blogs.cisco.com/security/talos/angler-domain-shadowing>, 2015.
- [BJM08a] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*, 2008.
- [BJM08b] Adam Barth, Collin Jackson, and John C. Mitchell. Securing Frame Communication in Browsers. In *USENIX Security*, 2008.
- [Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW*. IEEE, 2001.
- [Boh12] Aaron Bohannon. *Foundations of Webscript Security*. PhD thesis, University of Pennsylvania, 2012.
- [BP98] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks*, 1998.

- [BP10] Aaron Bohannon and Benjamin C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In John K. Ousterhout, editor, *USENIX Security*, 2010.
- [BRLT19] Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. Extending enumerative function synthesis via smt-driven classification. In *FMCAD*, 2019.
- [BSCS] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre. ProVerif 2.00: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial.
- [BSVL24] Philipp Beer, Marco Squarcina, Lorenzo Veronese, and Martina Lindorfer. Tabbed Out: Subverting the Android Custom Tab Security Model. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024.
- [Bug20] Bugcrowd. Public Bug Bounty List. <https://www.bugcrowd.com/bug-bounty-list/>, 2020.
- [BVV⁺23] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. Web platform threats: Automated detection of web security issues with WPT – artifacts and source code. <https://github.com/SecPriv/web-platform-threats>, 2023.
- [BVV⁺24] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. Web platform threats: Automated detection of web security issues with WPT. In *USENIX Security*, 2024.
- [CDP] Chrome devtools protocol. <https://chromedevtools.github.io/devtools-protocol/>.
- [Cen20] Censys. <https://censys.io/>, 2020.
- [CEWW22] L. Chen, S. Englehardt, M. West, and J. Wilander. Cookies: HTTP State Management Mechanism (IETF Draft. RFC 6265bis, Internet Engineering Task Force, 4 2022.
- [CFG⁺20] Stefano Calzavara, Riccardo Focardi, Niklas Grimm, Matteo Maffei, and Mauro Tempesta. Language-Based Web Session Integrity. In *CSF*. IEEE, 2020.
- [CFST17] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the Web: A Journey into Web Session Security. *ACM Computing Surveys (CSUR)*, 50(1):13:1–13:34, 2017.
- [CH86] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

- [Chra] chrome.declarativenetrequest extension api. <https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/>.
- [Chrb] chrome.webrequest extension api. <https://developer.chrome.com/docs/extensions/reference/webRequest/>.
- [Chrc] Issue 894228: CSP bypass with blob URL. <https://bugs.chromium.org/p/chromium/issues/detail?id=894228>.
- [Chr22a] Chromium Bugs. Issue 1351601: Cookie prefixes bypass via nameless cookies (rfc6265bis). <https://bugs.chromium.org/p/chromium/issues/detail?id=1351601>, 2022.
- [Chr22b] Chromium Bugs. Issue 1354090: post-CVE-2022-2860 security limitations of cookie prefixes and nameless cookies. <https://bugs.chromium.org/p/chromium/issues/detail?id=1354090>, 2022.
- [ci4a] CodeIgniter 4. https://codeigniter.com/user_guide/index.html.
- [ci4b] CodeIgniter Shield. <https://codeigniter4.github.io/shield/>.
- [CJD⁺18] Jianjun Chen, Jian Jiang, Hai-Xin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *USENIX Security*, 2018.
- [CK18] Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 2018.
- [Com] OWASP Community. HttpOnly Cookies. <https://owasp.org/www-community/HttpOnly>.
- [Com20] Common Crawl. Host- and Domain-Level Web Graphs Feb/-Mar/May 2020. <https://commoncrawl.org/2020/06/host-and-domain-level-web-graphs-febmarmay-2020/>, 2020.
- [Con20] Mike Conca. Changes to SameSite Cookie Behavior – A Call to Action for Web Developers. <https://hacks.mozilla.org/2020/08/changes-to-samesite-cookie-behavior/>, 2020.
- [coa] Bug 1459321 - treat loads the result from location.reload() as samesite. https://bugzilla.mozilla.org/show_bug.cgi?id=1459321.
- [coob] [RFC6265bis] Clarify behaviour on page refresh for samesite cookies. <https://github.com/httpwg/http-extensions/issues/628>.

- [cooc] [RFC6265bis] Refactor cookie retrieval algorithm to support non-http apis. <https://github.com/httpwg/http-extensions/pull/1428>.
- [CRB18] Stefano Calzavara, Alvisè Rabitti, and Michele Bugliesi. Semantics-Based Analysis of Content Security Policy Deployment. *ACM Transactions on the Web*, 2018.
- [CRB19] Stefano Calzavara, Alvisè Rabitti, and Michele Bugliesi. Sub-Session Hijacking on the Web: Root Causes and Prevention. *Journal of Computer Security*, 2019.
- [CRRB19] Stefano Calzavara, Alvisè Rabitti, Alessio Ragazzo, and Michele Bugliesi. Testing for Integrity Flaws in Web Sessions. In *ESORICS*, 2019.
- [csu] Express csrf: CSRF token middleware. <https://github.com/expressjs/csrf>.
- [Cza20] Lukasz Czajka. Practical proof search for coq by type inhabitation. In *IJCAR*, 2020.
- [Dai04] Leslie Daigle. RFC3912: WHOIS Protocol Specification, 2004.
- [Dat22] GitHub Advisory Database. ReactPHP’s HTTP server parses encoded cookie names so malicious `__Host-` and `__Secure-` cookies can be sent. <https://github.com/advisories/GHSA-w3w9-vrf5-8mx8>, 2022.
- [Dat23] GitHub Advisory Database. Incorrect parsing of nameless cookies leads to `__Host-` cookies bypass. <https://github.com/pallets/werkzeug/security/advisories/GHSA-px8h-6qyv-m22q>, 2023.
- [Det14] Detectify. Hostile subdomain takeover using heroku/github/desk + more. <https://labs.detectify.com/2014/10/21/hostile-subdomain/>, 2014.
- [DHK⁺22] Quoc Huy Do, Pedram Hosseini, Ralf Küsters, Guido Schmitz, Nils Wenzler, and Tim Würtele. A formal security analysis of the W3C web payment apis: Attacks and verification. In *SE&P*, 2022.
- [dja] Django Framework. <https://www.djangoproject.com/>.
- [DMP96] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 1996.
- [dot] ASP.NET. <https://dot.net>.
- [DR18] Andrej Dudenhefner and Jakob Rehof. A simpler undecidability proof for system F inhabitation. In *TYPES*, 2018.

- [DW] Domenic Denicola and Mike West. Origin Policy. <https://wicg.github.io/origin-policy/>.
- [EdO] EdOverflow. can-i-take-over-xyz. <https://github.com/EdOverflow/can-i-take-over-xyz>.
- [expa] Express. <https://expressjs.com/>.
- [expb] Express Session. <https://github.com/expressjs/session>.
- [fasa] Fastify. <https://www.fastify.io/>.
- [fasb] Fastify csrf-protection. <https://github.com/fastify/csrf-protection>.
- [fasc] Fastify Passport. <https://github.com/fastify/fastify-passport>.
- [FHK19] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. An Extensive Formal Security Analysis of the OpenID Financial-Grade API. In *SE&P*. IEEE, 2019.
- [Fir] Remote protocol (cpd) - firefox. <https://firefox-source-docs.mozilla.org/remote/cdp/>.
- [FKS16] Daniel Fett, Ralf Küsters, and Guido Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *CCS*, 2016.
- [FKS17] Daniel Fett, Ralf Küsters, and Guido Schmitz. The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines. In *CSF*, 2017.
- [flaa] Flask. <https://flask.palletsprojects.com/>.
- [flab] Flask Login. <https://flask-login.readthedocs.io/>.
- [flac] Flask WTF. <https://flask-wtf.readthedocs.io/>.
- [FOS04] The Mozilla Foundation and ASA Opera Software. Position paper for the W3C workshop on web applications and compound documents, 2004.
- [Fre20] FreeDNS. Free DNS Hosting, Dynamic DNS Hosting, Static DNS Hosting, subdomain and domain hosting. <https://freedns.afraid.org/>, 2020.
- [FS20] Edwin Foudil and Yakov Shafranovich. A File Format to Aid in Security Vulnerability Disclosure, 2020.
- [Git22] GitHub. Top programming languages. <https://octoverse.github.com/2022/top-programming-languages>, 2022.

- [Goo] Google. Chrome UX Report. <https://developer.chrome.com/docs/crux/>.
- [GPJV20] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security*, 2020.
- [Gro] Web Hypertext Application Technology Working Group. WHATWG – FAQ.
- [Han] J. Hanson. Passport – Simple, unobtrusive authentication for Node.js. <https://www.passportjs.org/>.
- [HB12] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, 2012.
- [HBdM11] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. μZ -an efficient engine for fixed points with constraints. In *CAV*, 2011.
- [HBSHA19] Phillip Hallam-Baker, Rob Stradling, and Jacob Hoffman-Andrews. RFC8659: DNS Certification Authority Authorization (CAA) Resource Record, 2019.
- [Hel17] Scott Helme. Cross-Site Request Forgery is dead! <https://scotthelme.co.uk/csrf-is-dead/>, 2017.
- [Hel19] S. Helme. CSRF is (really) dead. <https://scotthelme.co.uk/csrf-is-really-dead/>, 2019.
- [HMN15] Charlie Hothersall-Thomas, Sergio Maffei, and Chris Novakovic. Browser-Audit: automated testing of browser security features. In *ISSTA*, 2015.
- [HP21] J. Henkel and B. Pollard. Adding Rank Magnitude to the CrUX Report in BigQuery. <https://developer.chrome.com/blog/crux-rank-magnitude/>, 2021.
- [HTT22] IETF HTTP Working Group, HTTP Extensions. Issue 2229: [rfc6265bis] nameless cookies, client/server inconsistencies #2229. <https://github.com/httpwg/http-extensions/issues/2229>, 2022.
- [IAN] IANA. Root Zone Database. <https://www.iana.org/domains/root/db>.
- [imm] Chrome will disable modifying document.domain to relax the same-origin policy. <https://developer.chrome.com/blog/immutable-document-domain/>.
- [Jac02] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 2002.

- [JK19] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *IMC*. ACM, 2019.
- [Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, 1985.
- [JTL12] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing Browser Security Guarantees through Formal Shim Verification. In *USENIX Security*, 2012.
- [KHF⁺19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *SCP*, 2019.
- [Kit] Eiji Kitamura. Chrome will disable modifying ‘document.domain’ to relax the same-origin policy. <https://developer.chrome.com/blog/immutable-document-domain/>.
- [KKH⁺22] Sunwoo Kim, Young Min Kim, Jaewon Hur, Suhwan Song, Gwangmu Lee, and Byoungyoung Lee. FuzzOrigin: Detecting UXSS vulnerabilities in browsers through origin fuzzing. In *USENIX Security*, 2022.
- [KML⁺17] Panagiotis Kintis, Najmeh Miramirkhani, Charles Lever, Yizheng Chen, Rosa Romero Gómez, Nikolaos Pitropakis, Nick Nikiforakis, and Manos Antonakakis. Hiding in Plain Sight: A Longitudinal Study of Combosquatting Abuse. In *CCS*, 2017.
- [koa] Koa. <https://koajs.com>.
- [koab] Koa CSRF. <https://github.com/koajs/csrf>.
- [koac] Koa Passport. <https://github.com/rkusa/koa-passport>.
- [Kol02] M. Kolšek. Session fixation vulnerability in web-based applications. https://acrossecurity.com/papers/session_fixation.pdf, 2002.
- [KW] Krzysztof Kotowicz and Mike West. Trusted Types. <https://w3c.github.io/webappsec-trusted-types/dist/spec/>.
- [lar] Laravel Framework. <https://laravel.com/>.
- [LDC⁺16] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You’ve Got Vulnerability: Exploring Effective Vulnerability Notifications. In *USENIX Security*, 2016.
- [Lew06] Edward P. Lewis. RFC4592: The Role of Wildcards in the Domain Name System, 2006.

- [LHW16] Daiping Liu, Shuai Hao, and Haining Wang. All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records. In *CCS*, 2016.
- [LKP21] X. Likaj, S. Khodayari, and G. Pellegrino. Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks. In *RAID*. ACM, 2021.
- [LLD⁺17] Daiping Liu, Zhou Li, Kun Du, Haining Wang, Baojun Liu, and Hai-Xin Duan. Don't Let One Rotten Apple Spoil the Whole Barrel: Towards Automated Detection of Shadowed Domains. In *CCS*, 2017.
- [LLHN19] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers. In *NDSS*, 2019.
- [LPP18] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, (POPL), 2018.
- [LSP] Fix policy container construction for blobs. <https://github.com/whatwg/html/pull/6895>.
- [Lun13] R. Lundeen. The Deputies are Still Confused. <https://media.blackhat.com/eu-13/briefings/Lundeen/bh-eu-13-deputies-still-confused-lundeen-wp.pdf>, 2013.
- [LVLP18] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*, 2018.
- [LWN⁺16] Chaz Lever, Robert J. Walls, Yacin Nadji, David Dagon, Patrick D. McDaniel, and Manos Antonakakis. Domain-Z: 28 Registrations Later Measuring the Exploitation of Residual Trust in Domains. In *S&P*, 2016.
- [mc-a] Bug 247197 - upgrade requests in mixed content settings. https://bugzilla.mozilla.org/show_bug.cgi?id=1811787.
- [mc-b] Ship mixed content level 2 upgrading of passive mixed content. https://bugzilla.mozilla.org/show_bug.cgi?id=1811787.
- [McC62] John McCarthy. Towards a mathematical science of computation. In *IFIP*, 1962.
- [Mic22] Microsoft. Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core. <https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery>, 2022.

- [MLS21] Gordon Meiser, Pierre Laperdix, and Ben Stock. Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication. In *ASIA CCS*, 2021.
- [Moc87] Paul Mockapetris. RFC1035: Domain Names - Implementation and Specification, 1987.
- [Moza] Mozilla. Public Suffix List. <https://publicsuffix.org/>.
- [Mozb] Mozilla Developers Network. Cross-Origin Resource Sharing. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#Examples_of_access_control_scenarios.
- [Mozc] Mozilla Developers Network. Document.domain. https://developer.mozilla.org/en-US/docs/Web/API/Document/domain#browser_compatibility.
- [Mozd] Mozilla Developers Network. HTTP response status codes: 302 Found. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302>.
- [Moz22a] Mozilla Bugzilla. Issue 1782561. document.cookie desynchronization after cookie jar overflow. https://bugzilla.mozilla.org/show_bug.cgi?id=1782561, 2022.
- [Moz22b] Mozilla Bugzilla. Issue 1783536. document.cookie in an insecure origin process allows setting an insecure cookie in that process that has the same name as a secure one. https://bugzilla.mozilla.org/show_bug.cgi?id=1783536, 2022.
- [Moz22c] Mozilla Bugzilla. Issue 1783982: Cookie prefixes bypass via nameless cookies (rfc6265bis). https://bugzilla.mozilla.org/show_bug.cgi?id=1783982, 2022.
- [Moz22d] Mozilla. Project Fission. https://wiki.mozilla.org/Project_Fission, 2022.
- [MS07] Marco T. Morazán and Ulrik Pagh Schultz. Optimal lambda lifting in quadratic time. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *IFL*, 2007.
- [nam] [RFC6265bis] Accept nameless cookies. <https://github.com/httpwg/http-extensions/commit/0178223>.
- [Neta] Mozilla Developer Network. Cookie Store API. https://developer.mozilla.org/en-US/docs/Web/API/Cookie_Store_API.
- [Netb] Mozilla Developer Network. Set-Cookie. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>.

- [NIK⁺12] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. 2012.
- [NMY⁺11] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Engineering Secure Software and Systems*. Springer Berlin Heidelberg, 2011.
- [Not19] Mark Nottingham. RFC8615: Well-Known Uniform Resource Identifiers (URIs), 2019.
- [Osb17] Charlie Osborne. Uber Patches Security Flaw Leading to Subdomain Takeover. ZDNet, <https://www.zdnet.com/article/uber-patches-security-flaw-leading-to-subdomain-takeover/>, 2017.
- [Our] Should blob: inherit CSP in addition to origin? <https://github.com/whatwg/html/issues/2593#issuecomment-885083373>.
- [OWAa] OWASP. Cross-site request forgery prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.
- [OWAb] OWASP. Session hijacking attack. https://owasp.org/www-community/attacks/Session_hijacking_attack.
- [OWA20] OWASP. Amass. <https://owasp.org/www-project-amass/>, 2020.
- [PGT⁺19] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *NDSS*, 2019.
- [PHD⁺15] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *ITP*, 2015.
- [PHP] PHP Manual: Using Register Globals. <https://web.archive.org/web/20201205183413/https://www.php.net/manual/en/security.globals.php>.
- [PHP22] PHP Bug Tracker. Issue 81727: cookie integrity vulnerabilities. <https://bugs.php.net/bug.php?id=81727>, 2022.
- [Pol] Policy container explained. <https://github.com/antosart/policy-container-explained>.

- [Por] PortSwigger. Bypassing SameSite cookie restrictions. <https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions>.
- [pot] [RFC6265bis] Inconsistent browser behavior with secure and prefix cookies on localhost. <https://github.com/httpwg/http-extensions/issues/2605>.
- [Pup20] Puppeteer. <https://pptr.dev/>, 2020.
- [Rap20] Rapid7 Labs. Open Data, TCP and UDP scans. <https://opendata.rapid7.com/>, 2020.
- [RBC⁺20] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In *NDSS*, 2020.
- [RBN⁺19] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV*, 2019.
- [Res] Microsoft Research. The Z3 theorem prover. <https://github.com/Z3Prover/z3>.
- [Rey79] John C. Reynolds. Reasoning about arrays. *Commun. ACM*, 1979.
- [RGW⁺19] Richard Roberts, Yaelle Goldschlag, Rachel Walter, Taejoong Chung, Alan Mislove, and Dave Levin. You Are Who You Appear to Be: A Longitudinal Study of Domain Impersonation in TLS Certificates. In *CCS*, 2019.
- [RJ21] Stephen Röttger and Artur Janc. A Spectre proof-of-concept for a Spectre-proof web. <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>, 2021.
- [RMO19] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security*, 2019.
- [RPS23] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. The leaky web: Automated discovery of cross-site information leaks in browsers and the web. In *S&P*. IEEE, 2023.
- [saia] Sails Generate. <https://sailsjs.com/documentation/reference/command-line-interface/sails-generate>.
- [saib] Sails.js. <https://sailsjs.com/>.

- [sam] [RFC6265bis] SameSite=Strict cookie isolation on cross-site windows. <https://github.com/httpwg/http-extensions/issues/2644>.
- [SAVM23] Marco Squarcina, Pedro Adão, Lorenzo Veronese, and Matteo Maffei. Cookie crumbles: Breaking and fixing web session integrity. In *USENIX Security '23*, 2023.
- [SBDL01] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS*, 2001.
- [SBR17] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. On the Content Security Policy Violations due to the Same-Origin Policy. In *WWW*, 2017.
- [SCM21] Marco Squarcina, Stefano Calzavara, and Matteo Maffei. The remote on the local: Exacerbating web attacks via service workers caches. In *WOOT*, 2021.
- [Sec20] Sectigo. Crt.sh: Certificate search. <https://crt.sh/>, 2020.
- [Ser22] Amazon Web Services. Working with AWS Lambda proxy integrations for HTTP APIs. <https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-develop-integrations-lambda.html>, 2022.
- [SJ80] Norihisa Suzuki and David Jefferson. Verification decidability of presburger array programs. *J. ACM*, 1980.
- [SKC20] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *Network and Distributed System Security Symposium (NDSS)*, 2020.
- [SMWL10a] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *SE&P*. IEEE, 2010.
- [SMWL10b] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the Incoherencies in Web Browser Access Control Policies. In *SE&P*, 2010.
- [SNM17] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security*, 2017.
- [Soz07] Matthieu Sozeau. Subset coercions in coq. In *Types for Proofs and Programs*. Springer, 2007.
- [SPL⁺18] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications. In *NDSS*, 2018.

- [spra] Spring. <https://spring.io/>.
- [sprb] Spring Security. <https://spring.io/projects/spring-security>.
- [SPR⁺16] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security*, 2016.
- [SRBS19] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. BakingTimer: Privacy Analysis of Server-Side Request Processing Time. In *ACSAC*. ACM, 2019.
- [SRDB⁺21] Iskander Sanchez-Rola, Matteo Dell’Amico, Davide Balzarotti, Pierre-Antoine Vervier, and Leyla Bilge. Journey to the Center of the Cookie Ecosystem: Unraveling Actors’ Roles and Relationships. In *S&P*. IEEE, 2021.
- [SS13] Sooel Son and Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*, 2013.
- [SS20] Marius Steffens and Ben Stock. PMForce: Systematically Analyzing PostMessage Handlers at Scale. In *CCS 2020*, 2020.
- [SSO] Bypassing samesite restrictions using on-site gadgets. <https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions>.
- [STV⁺21] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web. In *USENIX Security*, 2021.
- [SWW19] Lorenz Schwittmann, Matthäus Wander, and Torben Weis. Domain Impersonation is Feasible: A Study of CA Domain Validation Vulnerabilities. In *EuroS&P*, 2019.
- [syma] Symfony. <https://symfony.com/>.
- [symb] Symfony CSRF. <https://github.com/symfony/security-csrf>.
- [The20] The Chromium Projects. SameSite Updates. <https://www.chromium.org/updates/same-site>, 2020.
- [tor] Tornado Web Server. <https://www.tornadoweb.org/>.
- [TTS] Trusted-types: Restrict to secure contexts. <https://github.com/w3c/webappsec-trusted-types/issues/259#issuecomment-630863753>.

- [Val13] J. Valim. CSRF token fixation attacks in Devise. <https://blog.plataformatec.com.br/2013/08/csrf-token-fixation-attacks-in-devise/>, 2013.
- [VFB⁺23] Lorenzo Veronese, Benjamin Farinier, Pedro Bernardo, Mauro Tempesta, Marco Squarcina, and Matteo Maffei. Webspec: Towards machine-checked analysis of browser security mechanisms. In *SE&P*. IEEE, 2023.
- [VGFSR⁺22] Tom Van Goethem, Gertjan Franken, Iskander Sanchez-Rola, David Dworken, and Wouter Joosen. SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model. In *ASIA CCS*. ACM, 2022.
- [W3Ca] W3C. Content Security Policy Level 3. <https://w3c.github.io/webappsec-csp/>.
- [W3Cb] W3C. File API. <https://www.w3.org/TR/FileAPI/>.
- [W3C09] W3C. Cross-Origin Resource Sharing. <https://www.w3.org/TR/2009/WD-cors-20090317/#cross-origin-request-with-preflight0>, 2009.
- [W3C14] W3C. Cross-Origin Resource Sharing. <https://www.w3.org/TR/cors/#generic-cross-origin-request-algorithms>, 2014.
- [W3C17] W3C. Working Draft: Clear Site Data. <https://www.w3.org/TR/clear-site-data/>, 2017.
- [W3C18] W3C. Content Security Policy Level 3. <https://www.w3.org/TR/CSP3/>, 2018.
- [W3C19] W3C. Service Workers. <https://www.w3.org/TR/service-workers/>, 2019.
- [W3C21] W3C. Secure Contexts. <https://w3c.github.io/webappsec-secure-contexts/>, 2021.
- [W3C23] W3C. Mixed Content. <https://www.w3.org/TR/mixed-content>, 2023.
- [W3T23] W3Techs. Usage statistics of PHP for websites. <https://w3techs.com/technologies/details/pl-php>, 2023.
- [Wal] James Walker. Subdomain Autofill Feature Raises Questions over LastPass Security. <https://portswigger.net/daily-swig/subdomain-autofill-feature-raises-questions-over-lastpass-security>.
- [Weba] Webkit web inspector. <https://webkit.org/web-inspector/>.

- [Webb] WebSpec: Coq proofs and source files. <https://github.com/secpriv/webspec>.
- [webc] web.dev. Schemeful Same-Site. <https://web.dev/schemeful-samesite/>.
- [Wes] Mike West. Cookie Prefixes. <https://tools.ietf.org/html/draft-west-cookie-prefixes-05>.
- [WG16] M. West and M. Goodwin. RFC6265: Same-site Cookies draft-west-first-party-cookies-07, 2016.
- [WHAa] WHATWG. Fetch Standard. <https://fetch.spec.whatwg.org/>.
- [WHAb] WHATWG. HTML - Living standard. <https://html.spec.whatwg.org/>.
- [Wil12] J. Wilander. Advanced CSRF and Stateless Anti-CSRF. https://owasp.org/www-pdf-archive//AppSecEU2012_Wilander.pdf, 2012.
- [Wir11] Wired. Researchers' Typosquatting Stole 20 GB of E-Mail From Fortune 500. <https://www.wired.com/2011/09/doppelganger-domains/>, 2011.
- [WL93] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *SCP*. IEEE, 1993.
- [WLR14] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID*, 2014.
- [WNK⁺23] Seongil Wi, Trung Tin Nguyen, Jihwan Kim, Ben Stock, and Sooel Son. Diffcsp: Finding browser bugs in content security policy enforcement through differential testing. In *NDSS*, 2023.
- [wpta] Samesite cookies are set by cross-site iframe navigations. https://bugzilla.mozilla.org/show_bug.cgi?id=1844827.
- [WPTb] The Web Platform Tests project. <https://web-platform-tests.org/>.
- [WSLJ16] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *CCS*, 2016.
- [wtf] WTForms. <https://wtforms.readthedocs.io/>.
- [WW20] Mike West and John Wilander. RFC6265: Cookies: HTTP State Management Mechanism, 2020.

- [yii] Yii PHP framework. <https://www.yiiframework.com/>.
- [Zal11] M. Zalewski. The tangled web: A guide to securing modern web applications. No Starch Press, 2011.
- [ZJL⁺15] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies Lack Integrity: Real-World Implications. In *USENIX Security*, 2015.
- [ZZS⁺20] Mingming Zhang, Xiaofeng Zheng, Kaiwen Shen, Ziqiao Kong, Chaoyi Lu, Yu Wang, Haixin Duan, Shuang Hao, Baojun Liu, and Min Yang. Talking with Familiar Strangers: An Empirical Study on HTTPS Context Confusion Attacks. In *CCS*, 2020.

Appendix to Chapter 2

A.1 Web Invariants

A.1.1 Cookies

Integrity of `__Host-` Cookies

In the following we give the complete Coq definition of the invariant defined in Section 2.4.1. We encode the invariant as:

```

1 Definition HostInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ rp corr pt sc ctx c_idx cookie _evs cname h,
3     Reachable gb evs st →
4     ((
5       evs = (EvResponse rp corr :: _evs) ∧
6       (rp_hd_set_cookie (rp_headers rp)) = Some cookie ∧
7       (sc_name cookie) = (Host cname) ∧
8       url_host (rp_url rp) = Some h
9     ) ∨ (
10      is_script_in_dom_path gb (st_window st) pt sc ctx ∧
11      evs = (EvScriptSetCookie pt (DOMPath [] DOMTopLevel) c_idx cookie :: _evs) ∧
12      (sc_name cookie) = (Host cname) ∧
13      url_host (wd_location ctx) = Some h
14    )) →
15     (sc_reg_domain cookie) = h.

```

A cookie can be set either via HTTP headers (lines 6-9) or via javascript (lines 12-15), however, when the name of the cookie has the `__Host-` prefix (lines 8 and 14), then the domain that registered the cookie must match (line 9) the URL of the response, or (line 15) the URL of the location of the window in which the script is running.

We can split the two cases in which the a cookie can be set and consider each case separately. When the cookie is set via HTTP headers the encoded invariant is:

```

1 Definition HostInvariantRP (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ rp corr cookie _evs cname h,

```

```

3   Reachable gb evs st →
4   (* A response is setting a cookie *)
5   evs = (EvResponse rp corr :: _evs) →
6   (rp_hd_set_cookie (rp_headers rp)) = Some cookie →
7   (* The cookie prefix is __Host *)
8   (sc_name cookie) = (Host cname) →
9   (* The cookie has been set by the domain of rp *)
10  url_host (rp_url rp) = Some h →
11  (sc_reg_domain cookie) = h.

```

A counterexample of one of the two invariants is also a counterexample of the complete HostInvariant, since the complete invariant is equivalent to requiring both cases (HostInvariantSC, HostInvariantRP) to hold:

```

1  ∀ gb evs st,
2  HostInvariant gb evs st ↔ (HostInvariantRP gb evs st ∧ HostInvariantSC gb evs st).

```

A proof of this equivalence is provided in [Webb].

Confidentiality of HttpOnly cookies

The HttpOnly attribute is designed to make cookies inaccessible to JavaScript both in read and write mode. This corresponds to the following invariant.

Invariant I.2. *Scripts can only access the cookies without the HttpOnly attribute.*

We encode the invariant in our model as follows:

```

1  Definition HttpOnlyInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2  ∀ sc cm c_idx cookie,
3  Reachable gb evs st →
4  (* A script has access to the cookie cm *)
5  Scriptstate gb st sc (SOCookie c_idx cm) →
6  (* The cookie is not httponly *)
7  st_cookiejar st.[c_idx] = Some cookie →
8  cj_http_only cookie = false.

```

Where line 5 specifies that a script `sc` in the page have access to the cookie `cm` that is stored in the cookiejar at index `c_idx`; line 8 requires the cookie to have the HttpOnly flag set to false.

Attack. JavaScript is allowed to perform HTTP requests using various APIs, e.g., XMLHttpRequest and fetch, and programmatically access the contents of the response. In particular, the authors of [SMWL10b] noticed that scripts could read the contents of the Set-Cookie header (through which cookies are set), thus violating the property that should be enforced by the HttpOnly flag.

When we configure our model to allow scripts to access the content of the Set-Cookie header, our toolchain produces a trace (shown in Figure A.1) which shows that a script is able to access a HttpOnly cookie by reading the response headers of a response that contains a Set-Cookie. Modern browsers have fixed the issue by preventing JS access

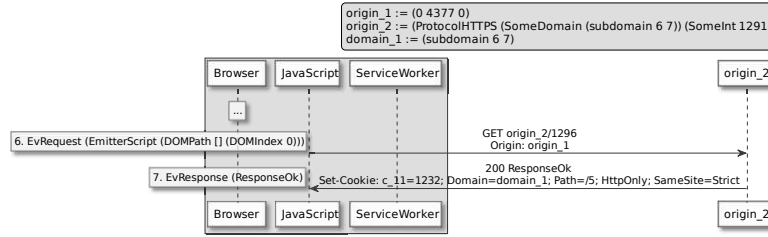


Figure A.1: HttpOnly Inconsistency

to the Set-Cookie header contained in responses. We can configure our model so that Set-Cookie is a *forbidden header* [WHAa] with `c_forbidden_headers (config gb)= true` and verify (up to a finite size) that the invariant holds.

A.1.2 Origin Header

The Origin header was proposed in [BJM08a] as a mechanism that websites can use to protect themselves against CSRF attacks. In particular, browsers populate this HTTP header with the origin that triggered the request being performed and Web servers should validate the header value to block undesired cross-origin requests.

Authenticity of request initiator

According to the proposal for the origin header [BJM08a], the header identifies the origin that initiated the request. If the browser is not able to determine the origin the header value should be null. So, when the Origin header value is different from null, no origin different from what is specified as the header value should be able to generate the request. This corresponds to the following Web invariant.

Invariant I.8. *If a request r includes the header $Origin: o$ (with $o \neq null$), then r was generated by origin o .*

We encode the invariant in our model as follow:

```

1 Definition OriginInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ em rq corr _evs orghd orgsrc,
3     Reachable gb evs st →
4       (* Request with origin header orgd *)
5       evs = (EvRequest em rq corr :: _evs) →
6       rq_hd_origin (rq_headers rq) = Some orghd →
7       (* The source origin is equal to orgd *)
8       is_request_source gb st rq (Some orgsrc) →
9       orgsrc = orghd.

```

where the `is_request_source` predicate holds when `Some orgsrc` is the origin that generated the request `rq`. Note that the predicate needs to take into account redirections: the source of a redirected request is the origin of the server which performed the redirection.

Attack. In [ABL⁺10] the authors reported a vulnerability in the proposed CSRF protection caused by the fact that the header is preserved across cross-origin redirects. This way a POST request to the attacker can be redirected back to the honest server, that accepts it since the Origin header contains the expected value. When we configure our model to reflect the past state of the Web platform that was current at the time of [ABL⁺10] publication, we can rediscover an attack that breaks the invariant on the origin header. In particular, our toolchain produces the following counterexample: (i) The user visits a website hosted on `origin_1` and submits a form towards `origin_2`; (ii) The server on `origin_2` redirects the request back to `origin_1` using HTTP status code 307 to preserve HTTP method and request body; (iii) the browser follows the redirect and produces a new request towards `origin_1`; the request contains the header `Origin: origin_1`, since it preserved upon redirect. As a result, `origin_1` will accept the incoming request since the Origin header contains the expected value, thus voiding the CSRF protection. The output trace is shown in Figure A.2.

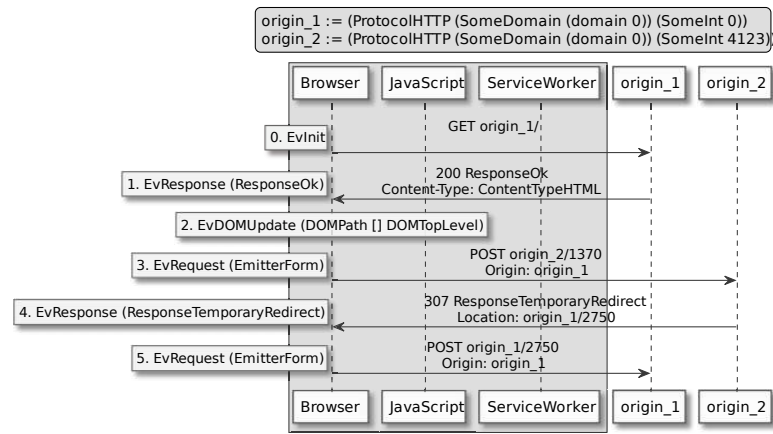


Figure A.2: Origin Header Inconsistency

Modern browsers tackle the issue by setting the header value to `null` in case of a cross-origin redirect, as dictated by the Fetch standard [WHAa, §4.4]. We verify the security of the solution (up to a finite size) by disabling the origin header in cross origin redirects with the `c_origin_header_on_cross_origin_redirect (config gb)= false` configuration option.

A.1.3 Same Origin Policy and CORS

The Same-Origin Policy (SOP) is a security mechanism that restricts the interactions between documents loaded from different origins. The SOP can be relaxed for trusted websites using Cross-Origin Resource Sharing (CORS) [WHAa, §3.2], a protocol that allows responses to specify the origins that are allowed to access their contents.

The CORS protocol distinguishes between simple and non-simple (or *preflighted*) requests depending on the request method, headers and contents [Mozb]. In particular, simple

requests use only the GET, HEAD, POST methods and are allowed to specify a limited sets of headers apart from the ones that are automatically added by the browser; preflighted requests are the requests that do not meet those conditions. Differently from simple requests which are safe to send cross-origin, preflighted requests require the browser to first issue a *pre-flight* request with the OPTIONS method to obtain the authorization to perform the actual request.

Authorization of non-simple request (*i*)

Following the specification for non-simple requests, we can define the relation between pre-flight and non-simple cross-origin requests as an invariant for the Same Origin Policy.

Invariant I.9. *A non-simple cross-origin request must be preceded by a pre-flight request.*

We encode the invariant in our model as follows:

```

1 Definition SOPInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ rq corr em rest,
3     Reachable gb evs st →
4     evs = (EvRequest em rq corr :: rest) →
5     (* The request is a non-simple request *)
6     not (is_cors_simple_request rq) →
7     (* The request is cross origin *)
8     is_cross_origin_request (st_window st) rq →
9     (* There needs to be a preflight request *)
10    Exists (IsEvRequestCORSPreflight rq) rest.
```

where the `IsEvRequestCORSPreflight` holds when an event in the list is a pre-flight request.

Attack. Early drafts of the HTML5 standard added the possibility to use the HTTP methods PUT and DELETE in HTML forms. However, to avoid introducing vulnerabilities in existing websites, the specification requires to use this methods only on same-origin requests. The authors of [ABL⁺10] found that browsers were transparently following cross-origin redirects when using PUT and DELETE. When we configure our model to reflect the past state of the Web platform in which HTML forms are allowed to use those methods, our toolchain is able to find a counterexample to the invariant (see Figure A.3). In particular, when a same-origin PUT (step 3) is redirected to a different origin, the resulting request (step 5) is non-simple and cross-origin. Since requests generated by forms do not trigger a pre-flight, this request breaks the invariant on the Same origin Policy.

The HTTP specification has been modified again to allow only HTTP methods GET and POST in form submissions [WHAb, §4.10.18.6], so this problem does not affect modern browsers. We can disable early HTML5 form methods with `c_earlyhtml5_form_methods (config gb) = false` and verify (up to a finite size) that the invariant holds.

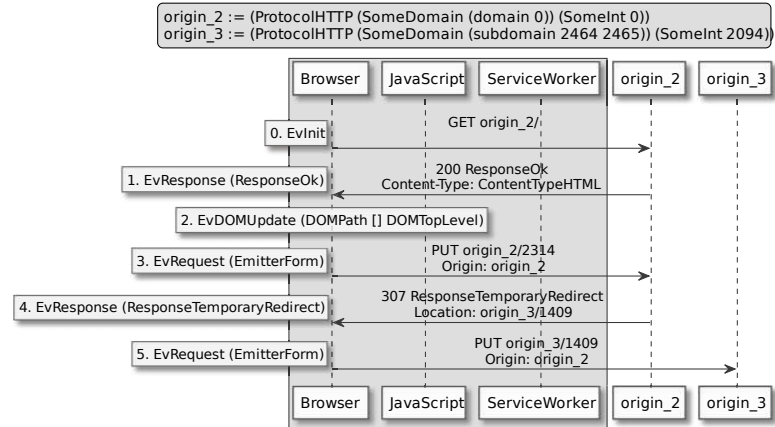


Figure A.3: Authorization of non-simple requests (i): cross-origin redirection of form-generated PUT request

Authorization of non-simple request (ii)

The response to a pre-flight request declares, through the Access-Control-Allow-Origin header, which origins are allowed to perform the cross-origin request. Given that the pre-flight response authorizes an origin to perform potentially harmful cross-origin requests, we should enforce the following invariant.

Invariant I.10. *The authorization to perform a non-simple request towards a certain origin o should come from o itself.*

That we encode in our model as follows:

```

1 Definition CORSInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ em rq corr scr_idx scr_pt rp rp_corr em_idx _evs,
3     Reachable gb evs st →
4     (* Non-simple request made by a script *)
5     evs = (EvRequest em rq corr :: _evs) →
6     em = EmitterScript scr_idx scr_pt ∧ (emitters gb).[em_idx] = em →
7     is_cross_origin_request (st_window st) rq →
8     not (is_cors_simple_request rq) →
9     (* Get CORS preflight response *)
10    is_cors_authorization_response gb st em_idx rq corr rp rp_corr →
11    (* The auth. comes from rq_url *)
12    origin_of_url (rq_url rq) = origin_of_url (rp_url rp).

```

Where `is_cors_authorization_response` (line 10) specifies that `rp` is the response to the CORS pre-flight request that is generated by the request `rq`; and line 12 requires that the origin the request `rq` is directed to must be the same one that generates the authorization response `rp`.

Attack. The original CORS draft allowed browsers to follow cross-origin redirects in responses to pre-flight requests [W3C09]. When we configure our model to follow redirects for pre-flight response, our toolchain produces a counterexample, shown in

Figure A.4, in which after a redirection `origin_3` responds with `Access-Control-Allow-Origin: origin_1` to a request made by `origin_1` towards `origin_2`. Thus, a website running on `origin_2` containing open redirectors might redirect the pre-flight request to a server under the attacker's control, that by returning a CORS header allows the attacker to relax the Same Origin Policy for `origin_2`.

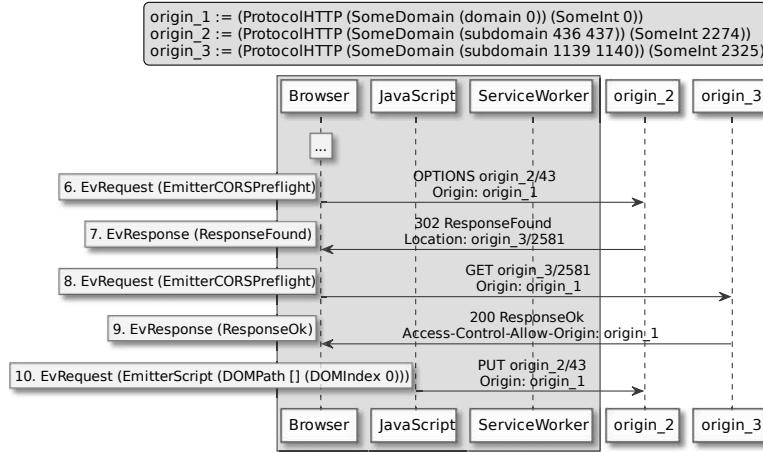


Figure A.4: Authorization of non-simple requests (ii): cross-origin redirection of pre-flight request

The most recent CORS specification (part of the Fetch Standard [WHAa]) specifies that browser should ignore redirects in pre-flight responses. We can verify (up to a finite size) that the invariant holds by configuring our model to ignore pre-flight redirects with `c_redirect_preflight_requests (config gb)= false`.

A.1.4 Content Security Policy

The *Content Security Policy* (CSP) allows Web developers to tighten the security of Web applications by controlling which resources can be loaded and executed by the browser. Originally, the CSP was designed to mitigate content injection vulnerabilities. Subsequently, it was extended to restrict browser navigation (e.g., form action, frame-ancestors) and protect DOM XSS sinks (via `trusted types`). A CSP policy consists of a set of directives and source expressions specifying an allow-list of actions the page is allowed to perform.

Interactions with the SOP

With the `script-src` CSP directive, developers can specify which scripts can be included in a page and thus access the DOM. This corresponds to the following property.

Invariant I.4. *The DOM of a page protected by CSP can be read/modified only by the scripts allowed by the policy.*

We encode the invariant in our model as follows:

```

1 Definition CSPInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ pt sc ctx pt_u src origin tctx tt _evs,
3     Reachable gb evs st →
4     (* A script sc is present in the page *)
5     is_script_in_dom_path gb (st_window st) pt sc ctx →
6     (* The DOM of the toplevel window has been modified by sc *)
7     evs = (EvScriptUpdateHTML pt (DOMPath [] pt_u) tctx :: _evs) →
8     (* The toplevel window is protected by CSP *)
9     rp_hd_csp (dc_headers (wd_document (st_window st))) = Some
10      { | csp_script_src := Some src; csp_trusted_types := tt | } →
11     (* The script sc is allowed by the CSP *)
12     origin_of_url (wd_location (st_window st)) = Some origin →
13     csp_src_match src origin (script_src sc).

```

Where the `csp_src_match` predicate holds when the `src` source expression matches the URL `script_src sc` in a page loaded from origin `origin`.

Attack. By running the query, our toolchain produces a counterexample that corresponds to the CSP violation discovered by the authors of [SBR17]. The complete trace is shown in Figure A.5: (steps 0-2) a page with `Content-Security-Policy: script-src 'none'` is loaded. The `none` value specifies that no script is allowed to be included in this page; (3-5) the page contains a same-origin (`origin_2`) iframe with `script-src origin_3` as CSP, allowing the page loaded in the iframe to (6-8) include scripts from `origin_3`; (9) the script running in the iframe (that was loaded from `origin_3`) can access the DOM of the parent page and modify it, which is allowed by SOP since the two pages come from the same origin. This is particularly dangerous in case the framed page is either compromised or malicious since any attacker-provided script could access the content of the parent page. Similar issues can arise when the framed page is protected by CSP while the parent is not or when the two pages have different origins (but the same site) and domain relaxation is performed [SBR17].

Preventing similar CSP violations could be achieved by having the same CSP policy enforced on all same-origin pages in a site and by disabling domain relaxation (e.g., by removing support for the `document.domain` setter). Using an origin-wide CSP policy can be done manually or via the upcoming Origin Policy [DW] mechanism when it will be supported by major browsers. We can configure our model to apply the same CSP policy to all same-origin pages with the `c_origin_wide_csp (config gb) = true` configuration option and verify that the invariant holds. The Coq proof of the correctness of this solution is available at [Webb].

Integrity of server-provided policies

A service worker [W3C19] is an event-driven worker that acts as a client-side proxy between Web applications and the network. Service workers are intended to enable Web applications to be used even without a network connection. They can intercept and modify network requests towards the origin against which they are registered and all requests triggered by the pages hosted on that origin. Using the Cache API, service workers can be used to store HTTP responses and then serve them even when the network is unreachable.

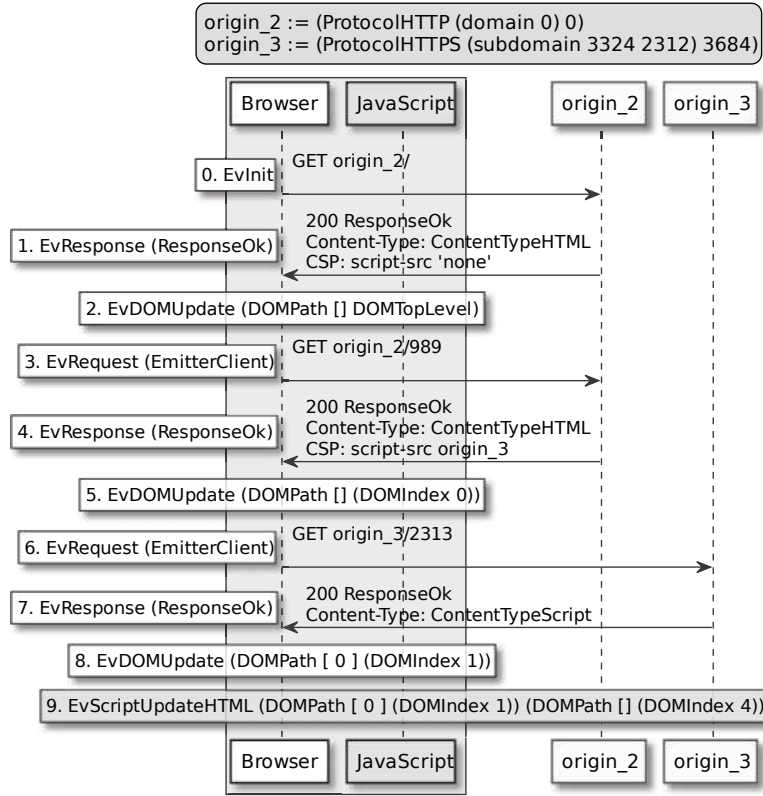


Figure A.5: CSP Inconsistency

Given the position of service workers in the processing of requests and responses, we must ensure that if a response obtained from the network contains a security policy, then such policy cannot be tampered with by the network stack and is correctly enforced by the browser. This corresponds to the following Web invariant.

Invariant I.5. *If a response from the server contains a security policy, then the browser enforces that specific policy.*

We encode the invariant in our model as follows:

```

1 Definition SWInvariant (gb: Global) (evs: list Event) (st: State) : Prop :=
2   ∀ corr rq_idx rp_idx rp em,
3     Reachable gb evs st →
4       (* Get the server response *)
5       is_server_response gb rq_idx rp →
6       (* Get the response that was rendered *)
7       in_history (st_fetch_engine st) corr (em, rq_idx, rp_idx) →
8       (* The CSP of the rendered response is equal to the server one *)
9       rp_hd_csp (rp_headers rp) =
10      rp_hd_csp (rp_headers ((responses gb).[rp_idx])).

```

For every response `rp` that would be generated by the server for a specific request index `rq_idx`, the response that has been rendered by the browser is present in the `ft_history` field of the `FetchEngine`. In particular, the history stores the mapping between requests and responses (`rq_idx`, `rp_idx` at line 7) for every response that is rendered by the browser. The invariant requires that the CSP of the response that is present in the history must be the same as the one that is generated by the server.

Attack. Running the query on WebSpec reveals that it is indeed possible for a service worker to break the invariant by responding to a request with a synthetic response (i.e., created with the `Response` constructor). In particular, when the server-generated response contains a security policy, a service worker could discard the network response and respond to the request with a new possibly unrelated response. This corresponds to an *inattentive* service worker which, due to a programming error, might remove or relax the security policies that are part of the responses the service worker is handling. We can specify that service workers are not allowed to generate synthetic responses using the `c_worker_allow_synthetic_responses (config gb)= false` configuration option. This configuration allow us to model the *cache-first* or *offline-first* pattern, the most popular¹ programming pattern that is used to serve content using service workers. An *offline-first* service worker intercepts all network requests: if a resource is found in the cache, then it is returned to the user before trying to download it; otherwise, if a resource is not found in the cache, the resource is fetched from the network and added to the cache.

When we run the query again in this configuration, WebSpec produces a counterexample (shown in Figure A.6): the invariant is broken once again when a service worker returns a synthetic response that has been added to the cache. Here, however, the synthetic response has been added to the cache by a script running on a page that is same-origin with the service worker. In particular, at step 6, a script running on `origin_2` creates a new response object that does not contain any security header and adds it to the cache. When the browser fetches `origin_2/`, the service worker matches the response that was previously cached by the script and returns it instead of downloading it. So the response rendered by the browser has a different CSP than the original response returned by the server, breaking the invariant. This is a special case of the attack described by Squarcina et al. [SCM21], where an attacker tampers with cached responses to strip or weaken the CSP served to the user. As the authors pointed out, this issue can be prevented by making the Cache API inaccessible to scripts running in the page context. We can verify that the invariant holds by restricting the Cache API to workers only, using the `c_script_update_cache (config gb)= false` configuration option. The security proof of this fix is available online [Webb].

¹https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline_Service_workers

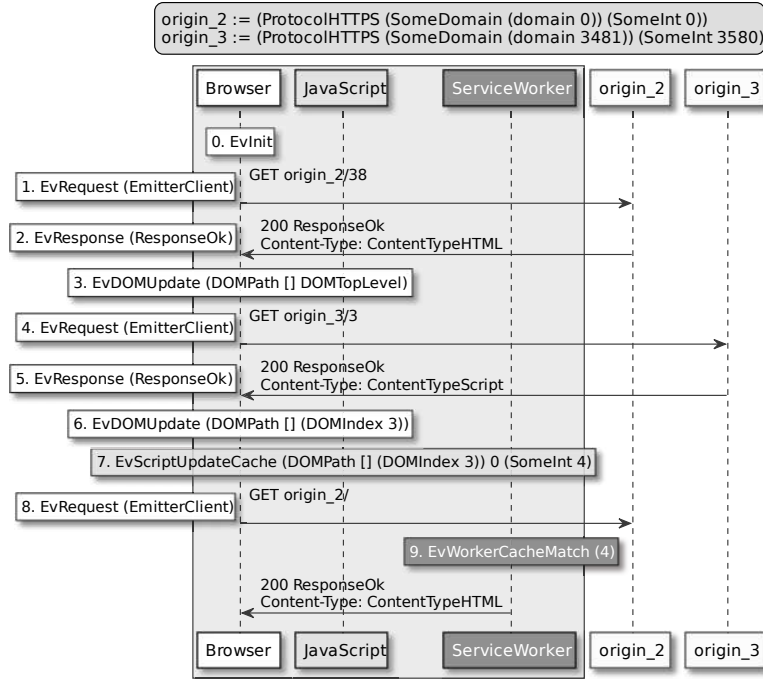


Figure A.6: Service Workers Cache Inconsistency

A.2 Verification of Security Properties: Preventing Cache API Access From Other Browsing Contexts

Any script running on a page that is same origin with a service worker is free to use the Cache API to manipulate the cached entries that the service worker could later serve to the user. The authors of [SCM21] propose disabling the Cache API from other browsing contexts as a solution to address the attacks they discovered. We will use our model to show that the invariant we define in Appendix A.1.4 holds if we allow only service workers to access the cache.

Assuming that service workers do not create synthetic responses, we can prove the stronger property that every response that is rendered by the browser had been generated by the server. That is, there is no possibility for the network stack of the browser to tamper with the response before rendering it, so the rendered responses are either received from the network or are clones of the responses that have been previously received by the browser. We encode the property as the following theorem:

```

1 Theorem
2 script_update_cache_disabled_implies_no_tampering:
3   ∀ gb,
4     c_worker_allow_synthetic_responses (config gb) = false →
5     c_script_update_cache (config gb) = false →
6   ∀ evs st corr rq_idx rp_idx rp em,
7     Reachable gb evs st →
8     is_server_response gb rq_idx rp →

```

```

9      (ft_history (st_fetch_engine st)).[corr] = Some (em, rq_idx, rp_idx) →
10     ((responses gb).[rp_idx]) = rp.

```

For each browser in which service workers cannot create synthetic responses and only service workers can access the cache (lines 4-5), for every reachable state, the responses in the history are always received from the server (line 10). The history field of the `FetchEngine` stores the mapping between request and response (indexes) for the responses that have been rendered by the browser. The assertion at line 10 implies that the mapping between requests and responses defined by the server (modeled as the `is_server_response` predicate) is the same as the one that is used by the browser: we cannot have, for a specific request index `rq_idx`, a mismatch between the server-defined response and the rendered response. This property could easily be violated if we allow workers to generate synthetic responses: when receiving a request identified by `rq_idx`, the service worker is free to choose a response which does not match the one returned by the `is_server_response` predicate.

Proving the theorem requires an additional helper lemma, which specifies that when our assumptions apply, for every reachable state that is processing the response to a remote request, i.e., the request and response URLs are not local scheme URLs, we can only have two possible configurations: (i) if there is a cached response `rp` that matches the current request, then `rp` is a response previously received from the network; (ii) alternatively, when no cached responses match the current request, the response comes from the network. We encode the lemma as follows:

```

1 Lemma
2 cache_or_ft_response_implies_server_response:
3   ∀ gb,
4     c_worker_allow_synthetic_responses (config gb) = false →
5     c_script_update_cache (config gb) = false →
6     ∀ evs st rq rq_idx rp rp_idx,
7       Reachable gb evs st →
8       rq = (requests gb).[rq_idx] →
9       rp = (responses gb).[rp_idx] →
10      not (is_local_scheme (rq_url rq)) →
11      not (is_local_scheme (rp_url rp)) →
12      (
13        ((wk_cache (st_service_worker st)).[rq_idx]) = Some rp_idx →
14        server_responses gb.[rq_idx] = rp_idx
15      ) ∧ (
16        ft_request (st_fetch_engine st) = rq →
17        ft_response (st_fetch_engine st) = Some rp →
18        server_responses gb.[rq_idx] = rp_idx
19      ).

```

Where the `server_responses` array in `Global` models the mapping between requests and responses sent by the server. In particular, the `is_server_response` predicate we use in our main theorem is defined in terms of the `server_responses` array.

To prove (ii) we just show that a state in which `ft_response` is not null is a state which just received a response from the server. To prove (i) we have to show that the only way to add a response to the cache is to first receive it from the server. This is

the case because of the two assumptions we made: the service worker cannot create or tamper with the network responses since is not allowed to create synthetic responses, so the only responses that it can add to the cache are the ones received from the server; scripts running on browsing contexts other than workers cannot modify the cache as our configuration option only allows service workers to access the Cache API. The proof of our main theorem follows from the application of the helper lemma.

Our main theorem implies that the invariant of Appendix A.1.4 holds when service workers do not tamper with the responses before caching and only workers are allowed to access the cache, thus proving the correctness of the solution proposed in [SCM21].

A.3 Compiler

WebSpec includes a compiler that aims to find inhabitants of inductive types, a problem which is known to be undecidable for CIC, the logic of Coq [DR18]. To this end, the compiler translates terms in a fragment of CIC into CHC logic, i.e., first-order logic with fixed-points expressed in terms of Constrained Horn Clauses, hence discharging the undecidability of the problem to CHC solvers [HBdM11, HB12]. In the following, we give an overview of how our compiler performs this translation.

A.3.1 Considered CIC Fragment

Contrary to related work [CK18, Cza20], our compiler does not perform a shallow embedding into *untyped* first-order logic, but instead performs a type-preserving translation into CHC logic, i.e., *typed* first-order logic with fixed-point. If, on the one hand, this allows us to leverage all the power of CHC solvers, this comes, on the other hand, at the price of restrictions on the fragment of the logic of Coq we consider.

The considered fragment of the logic of Coq we consider is CIC without dependent types (1), and where inductive type annotations and constructor arguments are restricted to ground variables (2). We also require inductive type parameters to be instantiated when the compiler is called. Before discussing these limitations, note that the resulting logic is still extremely expressive as it contains System F_ω , the higher-order polymorphic lambda calculus. This also means that the inhabitation problem is still undecidable on this fragment [DR18].

The reason for the restriction on inductive type annotations and constructor arguments (2) is twofold. The first reason is that CHC solvers do not perform type equation resolution, and therefore introducing symbolic type variables is forbidden. It is possible to circumvent this issue by performing a shallow embedding of types, however this would likely come at a significant cost in resolution time. The second reason is similar to the first, but for functions. However in this case, we expect this restriction to be relaxed in the future thanks to recent progress in function synthesis [BRLT19, RBN⁺19].

The restriction on dependent types (1) could also be circumvented by shallow embedding, but again at a high cost in resolution time. Instead, upcoming development of WebSpec

aims to relax this restriction so that dependent types are allowed in inductive types. This relaxation will cover a significant number of practical cases, like the famous example where the type of an array includes a program expression giving the size of that array.

A.3.2 Compilation Pipeline

In order to translate the support fragment of CIC to CHC logic, our compiler performs the following steps:

Term-Type-Kind Hierarchy From a syntactic point of view, types cannot be distinguished from terms in CIC. Because CHC does not permit such intricacy, we have to build a strict term-type-kind stratified hierarchy [Bar92], where kinds are defined as $k := \text{Prop} \mid \text{Set} \mid k \rightarrow k$. This stratification is done by recursive exploration, starting from the inductive type on which the compiler is called, and following CIC typing rules² to deduce to which stratum each syntactic term belongs. We rely on Coq type-checking to ensure that connections between terms, types and kinds are sound. As a side effect, this stratification makes a clear distinction between types and proposition or between terms and proofs, which will ease subsequent steps.

Partial Application In CIC, any term can be partially applied. This includes functions of course, but also inductive types, constructors, or type definitions. Such flexibility is not allowed in CHC, and therefore all partial applications have to be removed. This is done by systematically performing η -expansion [DMP96] on every term that could be applied.

Lambda Abstraction We also have to removed lambda abstractions, both those which are present in the original CIC terms and those which were introduced by η -expansion. To this end, we perform β -reduction wherever possible and remove remaining lambda-abstractions by lambda-lifting [Joh85, MS07].

Polymorphism and Higher-Order Thanks to the previous steps, all functions are now defined at top-level and totally applied. Therefore we can now remove the use of polymorphism and higher-order simply by specialization: For every application of a function (resp. an inductive type) to a type or a function argument, we generate a specialized version of the function (resp. the inductive type) where the type or function parameter is replaced by the argument.

Constructor Constraints Constructors of inductive types in CIC can contain terms with arbitrary constraints, while CHC only supports simple algebraic datatypes. Therefore, we split every non-simple inductive type into a simple inductive type of kind `Set` and an inductive type of kind `Prop` which encapsulates these constraints.

Once these steps are done, the rest of the compilation is straightforward. Simple inductive types of kind `Set` are mapped to CHC algebraic datatypes, inductive types of kind `Prop`

²<https://coq.inria.fr/refman/language/cic.html>

Table A.1: Solving time of inv. I.5 for progressively more complex models.

| # | Features | | | | | | | | Solving Time | |
|---|----------|--------|-------|------|-----|-----|------|-----|--------------|----------|
| | base | cookie | redir | cors | lsc | ref | pmsg | lst | Baseline | w/ Lemma |
| 1 | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | 71m | 13s |
| 2 | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | 3h 34m | 58s |
| 3 | ● | ● | ● | ○ | ○ | ○ | ○ | ○ | 3h 25m | 1m 7s |
| 4 | ● | ● | ● | ● | ○ | ○ | ○ | ○ | 2h 14m | 48s |
| 5 | ● | ● | ● | ● | ● | ○ | ○ | ○ | 6h 54m | 2m 11s |
| 6 | ● | ● | ● | ● | ● | ● | ○ | ○ | 9h 32m | 1m 52s |
| 7 | ● | ● | ● | ● | ● | ● | ● | ○ | 13h 20m | 2m 7s |
| 8 | ● | ● | ● | ● | ● | ● | ● | ● | 15h 46m | 3m |

base: Core Browser Functionality, CSP and Service Workers; cookie: Cookies;
redir: HTTP Redirections; cors: CORS Protocol and Headers; lsc: Local Schemes;
ref: Referer and Referrer Policy; pmsg: Post Message; lst: Local Storage

are mapped to relations, while CIC terms, types and proposition are mapped to CHC terms, sorts, and formulas.

A.4 Scalability

In this section, we report on the result of the experimental evaluation of the scalability of our browser model. In particular, we measure how the addition of individual Web components affects the performance of WebSpec counterexample finding pipeline, and show how lemmas (Section 2.6.1) are the most effective tool for improving the solving time. We focus, as our main case study, on the *Integrity of server-provided policies* Web invariant (Appendix A.1.4), since the counterexample found by WebSpec requires the browser model to only support service workers, the cache API, and the CSP header, outside of the core browser features (e.g., requests, responses, DOM, etc).

Starting from the features listed in Table A.2, we identify 10 modules, each representing a Web platform feature and refactor our model to be configurable w.r.t. the included components. With this modification our model is composed of a (core) core set of browser functionality on top of which we are able to automatically include or exclude (cookies) the Cookie and Set-Cookie headers, the cookie jar, and the document.cookie JavaScript API; (redir) HTTP redirections and response codes; (cors) the CORS protocol and its request and response headers; (csp) the Content-Security-Policy headers and rules, including the script-src and trusted-types directives; (sw) service Workers and the JavaScript Cache API; (lsc) support for local scheme URLs, the URL.createObjectURL API, and the inheritance rules for CSP; (ref) the Referer request header and the referrer policy mechanism; (pmsg) the Web messaging API (window.postMessage); (lst) the local storage API (window.localStorage).

Table A.1 reports the time required by WebSpec to find the counterexample for our case study on 8 incrementally more complex versions of the browser model. For each version, we include one additional features and run the query twice, measuring the running time with or without the inclusion of lemmas. The table clearly confirm the intuition that the

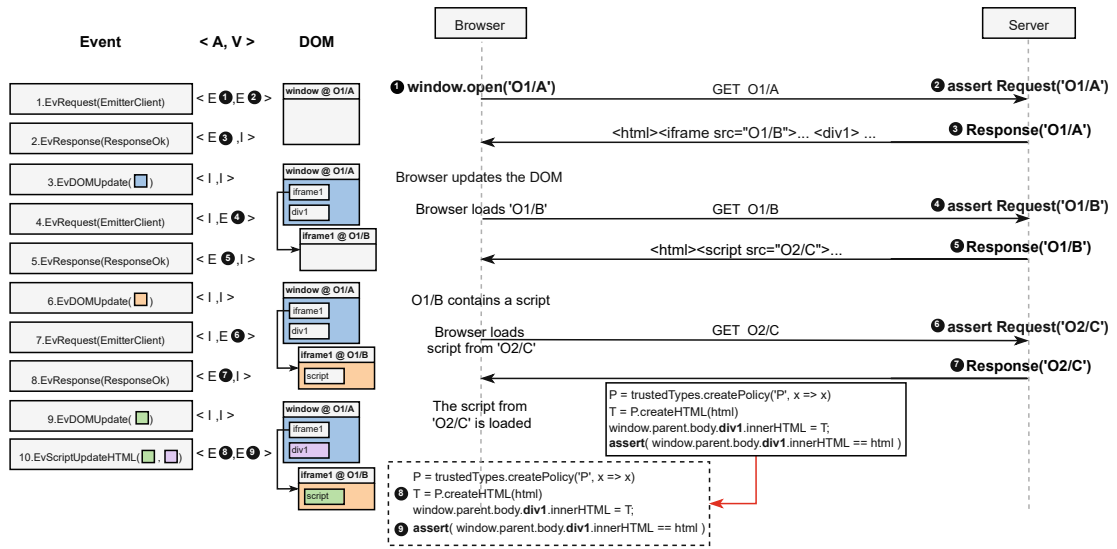


Figure A.7: WPT test corresponding to the Trusted Types attack trace (Figure 2.5)

solving time increases with the addition of new features, as the base model requires less than one hour, compared to the 16 hours of the complete model. This increase however is not predictable, as the addition of a feature may benefit the solver by introducing constraints which limit the search space, as for example in line #4, where the addition of CORS improves the solving time by one third. In all versions of the model, lemmas offer a substantial improvement of the counterexample finding pipeline, never exceeding the 3 minutes of solving time. In our case study, the lemma which is first applied by the solver is `script_state` (Section 2.6.1), which comprises the first 5 events of the 9 present in the complete trace. This shows that limiting the number of steps the solver needs to consider brings a more noticeable improvement than the simplification obtained by the removal of Web components.

A.5 Verifier Example

In Figure A.7, we provide an overview of how the attack trace shown in Figure 2.5 is translated into a WPT test, executable on real browsers. Each event (on the left) is assigned a *SAV* tuple. Notice that we omit the *Setup* element of the tuples since none of the events in the trace requires it.

Each WPT test generated by the verifier has at least two components: the *test launcher* and the *server*. The test launcher is the main WPT test file and consists of an HTML page containing a script that starts the actual test. The server component of a test is a Python script that handles requests (for all origins) and their corresponding responses. For simplicity, we omit the test launcher from the DOM column of Figure A.7.

The first event in the trace is an `EvRequest` to the URL `O1/A`, which is assigned an

explicit action ❶ and an explicit verification ❷. These are translated, respectively, to a `window.open` call, performed by the test launcher, and a server-side assertion to ensure that the request was performed. The `window.open` call creates a new browser window, represented in our example by the first element in the DOM column. The second event (`EvResponse`) is mapped to the explicit action ❸ of responding to the request, and an implicit verification. The verification of this action is implicit, since the response is known before runtime and generated by the test server. The third event in the trace (`EvDOMUpdate`) is mapped to an implicit action and an implicit verification, since updating the DOM after an HTTP response is internal browser behavior that cannot be explicitly tested. An incorrect DOM update, however, would cause all subsequent assertions to fail. Since the HTTP response from action ❸ returns an HTML page containing an `iframe` (`iframe1`) with source `01/B`, the browser is expected to load the subresource from this URL via a GET request. This request corresponds to event 4 (`EvRequest`) whose action is thus implicit. However, the verification of this action is necessary and explicitly done through an assertion at the server-side. The next two events (5 `EvResponse` and 6 `EvDOMUpdate`) are similar to events 2 and 3, differing only in the content of the HTTP response, this time containing an HTML page that includes a script with source `02/C`. The browser is then expected to fetch the script from `02/C` (events 7, 8, and 9). At this stage, the script is loaded in `iframe1` and ready to execute. Script events are always mapped to explicit actions and explicit verifications. In this example, event 10 (`EvScriptUpdateHTML`) is translated to JavaScript code that tries to modify (via `innerHTML`) a target element in the parent page ❸, and performs a WPT assertion on the content of the target element ❹. Since the `innerHTML` sink in the parent page requires a valid Trusted Type, the script first creates a loose policy and then updates the target element with a `TrustedHTML` created with such policy. Note that the JavaScript code that is executed for event 10 is received as the response body of event 8. Since WebSpec does not model JavaScript code but only the API calls that scripts can perform (Section 2.2.5), the verifier generates the response body (i.e., the code) for each script depending on the actions and verifications that the script needs to perform in the subsequent events.

If the test is successful, i.e., all assertions pass, the script in `iframe1` is able to modify an element in the parent page that is protected by Trusted Types, thus confirming the presence of the attack on real browsers.

A.6 Completeness

Table A.2 provides an overview of the features supported by the models discussed in Section 2.7. In particular, here we focus exclusively on Web components implemented in Web browsers, since this is ultimately the goal of our work, but models like WIM [DHK⁺22], WebSpi [BBDM14] and Alloy [ABL⁺10] additionally implement a variety of other features that are needed to model other parts of the Web ecosystem.

One of the main differences among the different proposals lies in the way JavaScript is

modeled. WIM and Featherweight Firefox model the small-step semantics of (a subset of) JavaScript: this is of fundamental importance, e.g., in WIM, since the model has been used to verify the security of Web protocols and it is necessary to define the precise semantics of scripts used by the parties involved in the protocol run. In WebSpec, similarly to the Alloy model of Akhawe et al. [ABL⁺10], we are only interested to the API calls that a script can perform. For this reason, we do not specify the exact behavior of a script, rather we assume that a script can call the supported APIs in any arbitrary way, using any data in its knowledge as parameters to these calls.

From the perspective of features support, WebSpec and WIM are the two most complete models available so far. As mentioned in Section 2.7, WebSpec does not support HSTS, HTTP basic authentication and the Web Payment API, since we abstract away from the network and from the specific implementation of Web servers. On the other hand, we support a variety of features that are missing in WIM browsers, such as CORS, cookie attributes like Domain, Path and SameSite, the `__Host-` prefix, CSP, the Cache API, interception of request of service workers, the `document.cookie` API, which play a prominent role for many of the attacks reported in this chapter. As shown in the table, there are also some minor differences concerning modeled URL components, type of supported HTTP redirects, status codes and headers, and functionality of service workers.

Concerning the other models, WebSpec essentially supports all the features implemented by them. Currently we only support the `Origin` and `Access-Control-Allow-Origin` HTTP headers for CORS, while the Alloy model supports all of them³: although the headers supported in WebSpec are sufficient to implement the fundamental CORS functionalities, the remaining ones allow a more careful treatment of CORS and we plan to implement them as future work.

³For space reasons, in Table A.2 we let ACA stand for `Access-Control-Allow`, AC for `Access-Control` and ACR for `Access-Control-Request`.

Table A.2: Comparison of supported Web components in existing models.

| Web Components | | WebSpec | WIM [DHK ⁺ 22] | WebSpi [BBDM14] | Alloy [ABL ⁺ 10] | FF [Boh12] |
|---------------------|--|-----------------------------------|--------------------------------------|-----------------|---|-----------------|
| URLs | Scheme | ● | ● | ● | ● | ● |
| | HTTP(S) | ● | ● | ● | ● | ● |
| | Pseudo-protocols | data:, blob: | - | - | - | about: |
| | Host | ● | ○ | ○ | ○ | ○ |
| | Port | ● | ○ | ○ | ○ | ○ |
| | Path | ● | ● | ● | ● | ● |
| | Parameters | ○ | ● | ● | ● | ● |
| | Fragment | ○ | ● | ○ | ○ | ○ |
| | JS API URL.createObjectURL | ● | ○ | ○ | ○ | ○ |
| HTTP | Request methods | ● | ● | ● | ● | ● |
| | GET | ● | ● | ● | ● | ○ |
| | POST | ● | ● | ● | ● | ○ |
| | Others | PUT, DELETE, OPTIONS | PUT, DELETE, OPTIONS, TRACE, CONNECT | - | PUT, DELETE, OPTIONS | - |
| | Response codes | | | | | |
| | Redirection | 302, 307 | 303, 307 | 302 | 302, 303, 307 | - |
| | Others | 200, 204 | 101, 200 | 200 | 200, 401 | 200 |
| | Headers (not fitting the categories below) | | | | | |
| | Referer | ● | ● | ○ | ○ | ○ |
| | ReferrerPolicy | ● | ● | ○ | ○ | ○ |
| | Directive origin | ● | ● | ○ | ○ | ○ |
| | Directive no-referrer | ● | ● | ○ | ○ | ○ |
| | Directive unsafe-url | ● | ● | ○ | ○ | ○ |
| | Authorization | ○ | ○ | ○ | ○ | ○ |
| | Content-Type | ● | ○ | ○ | ○ | ○ |
| | Location | ○ | ● | ○ | ○ | ○ |
| | Strict-Transport-Security | ○ | ● | ○ | ○ | ○ |
| | WWW-Authenticate | ○ | ○ | ○ | ● | ○ |
| Cookies | HTTP headers | ● | ● | ● | ● | ● |
| | Cookie | ● | ● | ● | ● | ● |
| | Set-Cookie | ● | ● | ● | ● | ● |
| | Attributes | | | | | |
| | Domain | ● | ○ | ○ | ● | ● |
| | Path | ● | ○ | ○ | ● | ○ |
| | Secure | ● | ● | ○ | ● | ○ |
| | HttpOnly | ● | ○ | ○ | ○ | ○ |
| | SameSite | ● | ○ | ○ | ○ | ○ |
| | __Secure- Prefix | ● | ○ | ○ | ○ | ○ |
| | __Host- Prefix | ● | ○ | ○ | ○ | ○ |
| | JS API document.cookie | ● | ● | ● | ○ | ○ |
| Windows | SOP for cookies | ● | ● | ● | ● | ● |
| | Multiple tabs | ○ | ● | ○ | ○ | ○ |
| | Framing support | ● | ● | ○ | ○ | ○ |
| | Cross-window communication (postMessage API) | ● | ● | ○ | ○ | ○ |
| | JS API window.location | ● | ● | ○ | ○ | ○ |
| | JS API window.history | ○ | ● | ○ | ○ | ○ |
| DOM | JS API window.close | ○ | ● | ○ | ○ | ○ |
| | Supported elements | <script>, <iframe>, <form>, | <script>, <iframe>, <form> | - | <form> | <script>, <div> |
| | JS API for DOM manipulation | ● | ○ | ○ | ○ | ● |
| | JS API document.domain | ● | ○ | ○ | ○ | ○ |
| XHR / Fetch API | SOP for DOM access | ● | ● | ○ | ○ | ● |
| | SOP for XHR / fetch requests | ● | ● | ● | ● | ● |
| | Sending requests via JavaScript | ● | ● | ● | ○ | ● |
| | Reading responses via JavaScript | ● | ● | ○ | ○ | ○ |
| | Forbidden response headers (Set-Cookie) | ● | ● | ○ | ○ | ○ |
| CORS | Request types | | | | | |
| | Simple requests | ● | ○ | ○ | ● | ○ |
| | Non-simple requests (preflight) | ● | ○ | ○ | ● | ○ |
| | HTTP headers | | | | | |
| | Origin | ● | ● | ● | ● | ○ |
| | AC-Allow-Origin | ● | ○ | ○ | ● | ○ |
| | Others | - | - | - | ACA-Method, ACA-Headers, ACA-Credentials, AC-Max-Age, ACR-Method, ACR-Headers | - |
| CSP / Trusted Types | CSP directives | | | | | |
| | script-src | ● | ○ | ○ | ○ | ○ |
| | trusted-types | ● | ○ | ○ | ○ | ○ |
| | require-trusted-types-for | ● | ○ | ○ | ○ | ○ |
| | CSP Inheritance | ● | ○ | ○ | ○ | ○ |
| | Trusted types | | | | | |
| | Create trusted types (policy.createHTML) | ● | ○ | ○ | ○ | ○ |
| Service Workers | Secure context restriction | ● | ○ | ○ | ○ | ○ |
| | Interception of requests (evt.respondWith) | ● | ○ | ○ | ○ | ○ |
| | Access to the cache API | ● | ○ | ○ | ○ | ○ |
| | Messaging with other windows | ○ | ● | ○ | ○ | ○ |
| | Opening new windows (Clients.windowOpen) | ○ | ● | ○ | ○ | ○ |
| Storage APIs | Cache API | | | | | |
| | Caches.put | ● | ○ | ○ | ○ | ○ |
| | Caches.match | ● | ○ | ○ | ○ | ○ |
| | Secure context restriction | ● | ○ | ○ | ○ | ○ |
| | Local storage | | | | | |
| | localStorage.getItem | ● | ● | ○ | ○ | ○ |
| Web Payment APIs | localStorage.setItem | ● | ● | ○ | ○ | ○ |
| | | ○ | ● | ○ | ○ | ○ |

Appendix to Chapter 3

B.1 Encoding Known Web Invariants

We report in the following our encoding in first-order logic of the 3 invariants which were previously defined in the literature. For each invariant, we provide the natural language version of the property and its encoding in our model.

B.1.1 Integrity of Secure Cookies

The RFC dictates that it should not be possible to set cookies with the `Secure` attribute from insecure channels [CEWW22, §5.5]. This invariant has been previously formalized as part of the WebSpec framework [VFB⁺23] as follows.

Invariant (I.1). *Cookies with the `Secure` attribute can only be set over secure channels.*

The invariant is encoded in our model as follows:

$$\begin{aligned}
 \text{SECURE-COOKIES-INVARIANT}(tr) := & \\
 & t_2 > t_1 \wedge \\
 & \mathbf{net_response}(_, url, \{set_cookie_headers\}, _)@_{tr}t_1 \wedge \\
 & set_cookie \in set_cookie_headers \wedge \\
 & name \neq "=" \wedge value \in split_cookie(set_cookie) \wedge \\
 & "Secure" \in split_cookie(set_cookie) \wedge \\
 & \mathbf{cookie_jar_set}(name, value, \{Secure=true\}, false)@_{tr}t_2 \Rightarrow \\
 & (url_proto(url, "wss") \vee url_proto(url, "https"))
 \end{aligned}$$

For every network response at time t_1 that leads to a cookie being set in the cookie jar (at time t_2) that has the `Secure` attribute set to `true`, then the protocol of the response url is either `https` or `wss` (i.e., it is a secure channel).

B.1.2 Confidentiality of HttpOnly cookies

The HttpOnly cookie attribute informs browsers that accesses to cookies with this attribute set to true by non-HTTP APIs, i.e., `document.cookie`, should not be allowed. This property was formalized in the literature [VFB⁺23] as:

Invariant (I.2). *Scripts can only access cookies without the HttpOnly attribute.*

We encode the invariant as:

$$\begin{aligned} \text{HTTP-ONLY-INVARIANT}(tr) := & \\ & t_2 > t_1 \wedge \\ & \text{cookie-jar-set}(name, value, \{http\text{-only}, secure, domain, path\})@_{tr}t_1 \wedge \\ & \text{js-get-cookie}(ctx, cookies)@_{tr}t_2 \wedge \\ & name \neq "" \wedge value \in \text{split-cookie}(cookies) \wedge \\ & \text{cookie-match}(path, domain, secure, ctx\text{-location}(ctx)) \Rightarrow \\ & http\text{-only} = false \end{aligned}$$

For every access to `document.cookie` in the domain *domain* at time t_2 that successfully returns a cookie previously stored in the cookie jar for the same domain (at time t_1) then the cookie's HttpOnly attribute has the value *false*.

B.1.3 Integrity of __Host- cookies

Browsers should enforce that cookies with a name prefix of `__Host-` are set with an empty *domain* attribute, making these cookies host-only. Effectively, these cookies can only be set by responses to the domain that created them or by scripts running in that domain. Veronese et al. [VFB⁺23] discuss this property of the `__Host-` prefix and propose the following natural language formalization:

Invariant (I.3). *A `__Host-` cookie set for domain *d* can only be set by *d* or by scripts included in pages on *d*.*

We encode the invariant in our model as:

$$\begin{aligned} \text{HOST-INVARIANT}(tr) := & \\ & t_2 > t_1 \wedge \\ & (\text{net-response}(_, url, \{set\text{-cookie-headers}\}, _)@_{tr}t_1 \wedge \\ & \quad set\text{-cookie} \in set\text{-cookie-headers} \wedge \\ & \quad \text{"__Host-"} \neq cname \neq "" \wedge cvalue \in \text{split-cookie}(set\text{-cookie}) \wedge \\ & \quad url\text{-domain}(url, host) = _) \vee \\ & (\text{js-set-cookie}(ctx, set\text{-cookie}, _)@_{tr}t_1 \wedge \\ & \quad \text{"__Host-"} \neq cname \neq "" \wedge cvalue \in \text{split-cookie}(set\text{-cookie}) \wedge \\ & \quad url\text{-domain}(ctx\text{-location}(ctx), host) = _) \\ & \text{cookie-jar-set}(\text{"__Host-"} \neq cname, cvalue, \{domain\}, false)@_{tr}t_2 \Rightarrow \\ & \quad domain = host \end{aligned}$$

For every network response or access to `Document.cookie` property at t_1 that causes a `cookie-jar-set` event at t_2 which sets a `__Host-`-prefixed cookie, the effective domain of the cookie must be equal to the domain of the *url* of the network response or to the browsing context where the access to `Document.cookie` was performed.

B.2 Test Selection

Table B.1 reports the considered tests for our evaluation. In particular, we execute all `testharness.js` tests from the d888ebb version of WPT (Apr 2023).

B. APPENDIX TO CHAPTER 3

| | | | | | |
|---------------------------|------|---------------------------------|----|------------------------------|---|
| html | 6404 | compute-pressure | 30 | webvr | 7 |
| referrer-policy | 1301 | web-bundle | 29 | remote-playback | 7 |
| content-security-policy | 821 | focus | 29 | pointerlock | 7 |
| fetch | 754 | domparsing | 29 | mediasession | 7 |
| dom | 473 | soft-navigation-heuristics | 28 | mediacapture-fromelement | 7 |
| IndexedDB | 454 | cors | 27 | keyboard-lock | 7 |
| svg | 448 | payment-request | 26 | fledge | 7 |
| xhr | 391 | shape-detection | 25 | x-frame-options | 6 |
| navigation-api | 375 | webrtc-encoded-transform | 24 | webrtc-stats | 6 |
| workers | 321 | credential-management | 24 | shared-storage | 6 |
| service-workers | 296 | animation-worklet | 24 | gamepad | 6 |
| websockets | 276 | reporting | 23 | file-system-access | 6 |
| streams | 251 | mediacapture-image | 23 | close-watcher | 6 |
| webaudio | 247 | import-maps | 23 | badging | 6 |
| wasm | 246 | domxpath | 23 | webrtc-svc | 5 |
| bluetooth | 230 | worklets | 22 | wai-aria | 5 |
| encoding | 215 | orientation-event | 21 | push-api | 5 |
| upgrade-insecure-requests | 197 | inert | 20 | delegated-link | 5 |
| shadow-dom | 169 | requestidlecallback | 19 | content-index | 5 |
| webrtc | 168 | longtask-timing | 19 | clear-site-data | 5 |
| mixed-content | 163 | visual-viewport | 18 | webrtc-identity | 4 |
| webmessaging | 154 | storage-access-api | 18 | vibration | 4 |
| mathml | 140 | long-animation-frame | 18 | ua-client-hints | 4 |
| webxr | 137 | hr-time | 18 | proximity | 4 |
| custom-elements | 132 | screen-wake-lock | 17 | payment-method-basic-card | 4 |
| pointerevents | 124 | quirks | 17 | mimesniff | 4 |
| speculation-rules | 123 | notifications | 17 | merchant-validation | 4 |
| resource-timing | 122 | mediacapture-record | 17 | lifecycle | 4 |
| WebCryptoAPI | 119 | js-self-profiling | 17 | device-memory | 4 |
| web-animations | 119 | battery-status | 17 | virtual-keyboard | 3 |
| scheduler | 108 | urlpattern | 16 | trust-tokens | 3 |
| encrypted-media | 106 | orientation-sensor | 16 | top-level-storage-access-api | 3 |
| client-hints | 104 | measure-memory | 16 | timing-entrytypes-registry | 3 |
| scroll-animations | 102 | geolocation-API | 16 | screen-details | 3 |
| eventsourcing | 100 | screen-orientation | 15 | periodic-background-sync | 3 |
| editing | 98 | old-tests | 15 | parakeet | 3 |
| infrastructure | 91 | browsing-topics | 15 | netinfo | 3 |
| trusted-types | 88 | beacon | 15 | mst-content-hint | 3 |
| FileAPI | 87 | web-share | 14 | generic-sensor | 3 |
| layout-instability | 81 | resize-observer | 14 | autoplay-policy-detection | 3 |
| media-source | 78 | input-events | 14 | webrtc-priority | 2 |
| permissions-policy | 76 | imagebitmap-renderingcontext | 14 | webhid | 2 |
| performance-timeline | 76 | background-fetch | 14 | savedata | 2 |
| encoding-detection | 75 | secure-payment-confirmation | 13 | png | 2 |
| web-locks | 73 | presentation-api | 13 | permissions-revoke | 2 |
| webcodecs | 73 | picture-in-picture | 13 | permissions-request | 2 |
| webvtt | 71 | payment-handler | 13 | managed | 2 |
| fullscreen | 70 | console | 13 | intervention-reporting | 2 |
| intersection-observer | 69 | scroll-to-text-fragment | 12 | installedapp | 2 |
| cookies | 69 | is-input-pending | 12 | html-media-capture | 2 |
| selection | 63 | font-access | 12 | direct-sockets | 2 |
| user-timing | 62 | accelerometer | 12 | deprecation-reporting | 2 |
| largest-contentful-paint | 61 | web-nfc | 11 | density-size-correction | 2 |
| signed-exchange | 60 | speech-api | 11 | background-sync | 2 |
| cookie-store | 60 | page-visibility | 11 | window-placement | 1 |
| compression | 59 | network-error-logging | 11 | webrtc-ice | 1 |
| serial | 58 | idle-detection | 11 | web-otp | 1 |
| webidl | 55 | geolocation-sensor | 11 | webmidi | 1 |
| url | 54 | forced-colors-mode | 11 | webdriver | 1 |
| event-timing | 54 | server-timing | 10 | subresource-integrity | 1 |
| paint-timing | 53 | screen-capture | 10 | private-click-measurement | 1 |
| navigation-timing | 53 | sanitizer-api | 10 | payment-method-id | 1 |
| mediacapture-streams | 51 | pending-beacon | 10 | page-lifecycle | 1 |
| webnn | 50 | mediacapture-insertable-streams | 10 | media-playback-quality | 1 |
| preload | 50 | media-capabilities | 10 | mediacapture-region | 1 |
| webusb | 49 | magnetometer | 10 | mediacapture-handle | 1 |
| webstorage | 49 | gyroscope | 10 | mediacapture-extensions | 1 |
| feature-policy | 49 | compat | 10 | input-device-capabilities | 1 |
| fs | 48 | audio-output | 10 | eyedropper | 1 |
| loading | 47 | ambient-light | 10 | entries-api | 1 |
| clipboard-apis | 47 | webrtc-extensions | 9 | ecmascript | 1 |
| element-timing | 46 | touch-events | 9 | custom-state-pseudo-class | 1 |
| uievents | 43 | permissions | 9 | contenteditable | 1 |
| portals | 42 | webgl | 8 | content-dpr | 1 |
| webtransport | 37 | video-rvfc | 8 | contacts | 1 |
| webauthn | 36 | subapps | 8 | apng | 1 |
| js | 35 | secure-contexts | 8 | acid | 1 |
| document-policy | 33 | keyboard-map | 8 | acname | 1 |
| storage | 32 | document-picture-in-picture | 8 | | |

Table B.1: Considered WPT tests.
Total: 24896, WPT Version: d888ebb

Appendix to Chapter 5

C.1 Web Framework Analysis

Table C.1 lists the entire pool of Web frameworks considered for this study. We restricted the analysis to the top 10 frameworks according to the GitHub metrics *watch*, *fork*, and *stars*, obtaining the final set of 13 frameworks.

| Framework | Language | GH Watch | GH Fork | GH Star |
|---------------|----------|-------------|--------------|--------------|
| ASP.NET MVC | C# | 75 | 329 | 739 |
| ASP.NET Core | C# | 1.4k | 7.7k | 27.8k |
| Service Stack | C# | 515 | 1.6k | 5k |
| Nancy | C# | 438 | 1.5k | 7.2k |
| Spring | Java | 3.4k | 33.3k | 47.1k |
| Play | Java | 683 | 4k | 12.1k |
| Spark | Java | 413 | 1.6k | 9.3k |
| Vert.x-web | Java | 79 | 470 | 955 |
| Vaadin | Java | 53 | 59 | 361 |
| Dropwizard | Java | 398 | 3.4k | 8.2k |
| Blade | Java | 302 | 1.1k | 5.6k |
| ZK | Java | 46 | 169 | 350 |
| Apache Struts | Java | 124 | 737 | 1.1k |
| Apache Wicket | Java | 61 | 354 | 616 |
| Express | JS | 1.8k | 9.6k | 56.6k |
| Meteor | JS | 1.6k | 5.2k | 42.9k |
| Koa | JS | 847 | 3.2k | 32.5k |
| Hapi | JS | 422 | 1.4k | 13.8k |
| Sails | JS | 667 | 2k | 22.2k |
| Fastify | JS | 281 | 1.7k | 22.7k |
| ThinkJS | JS | 268 | 643 | 5.3k |
| Total.js | JS | 218 | 459 | 4.1k |
| AdonisJS | JS | 229 | 579 | 12.3k |
| Laravel | PHP | 4.6k | 22.4k | 69.3k |
| Symfony | PHP | 1.2k | 8.6k | 26.7k |
| Slim | PHP | 525 | 1.9k | 11.3k |
| CakePHP | PHP | 573 | 3.5k | 8.5k |
| Zend/Laminas | PHP | 18 | 56 | 1.4k |
| CodeIgniter | PHP | 1.6k | 7.8k | 18.2k |
| FuelPHP | PHP | 107 | 287 | 1.4k |
| Yii2 | PHP | 1.1k | 7k | 13.9k |
| Phalcon | PHP | 658 | 1.9k | 10.6k |
| Li3 | PHP | 91 | 247 | 1.2k |
| CodeIgniter4 | PHP | 278 | 1.6k | 4.2k |
| Flask | Python | 2.2k | 15k | 58.5k |
| Django | Python | 2.3k | 26.9k | 63.3k |
| Tornado | Python | 1k | 5.4k | 20.5k |
| Bottle | Python | 320 | 1.4k | 7.6k |
| Pyramid | Python | 160 | 878 | 3.7k |
| Falcon | Python | 273 | 872 | 8.7k |
| Zope | Python | 91 | 99 | 288 |
| Masonite | Python | 57 | 104 | 1.7k |
| TurboGears2 | Python | 32 | 76 | 777 |
| Web2py | Python | 220 | 866 | 2k |

Table C.1: Web development frameworks from [LKP21] ranked according to GitHub metrics as of April 8, 2022.