Saarland University

Department of Computer Science

Security Testing at Scale: Studying Emerging Client-side Vulnerabilities in the Modern Web

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Soheil Khodayari

Saarbrücken, 2024

Tag des Kolloquiums:	28. August 2024
Dekan:	Prof. Dr. Roland Speicher

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Martina Maggio
Berichterstattende:	Dr. Giancarlo Pellegrino
	Prof. Bernd Finkbeiner, PhD
	Prof. Dr. Martin Johns
Akademischer Mitarbeiter:	Dr. Bhupendra Acharya

Zusammenfassung

Die rasante Entwicklung der Client-seitigen Technologien in jüngster Zeit hat neue Varianten traditioneller Sicherheitsprobleme hervorgebracht, die sich nun ausschließlich auf Client-seitige JavaScript-Programme beziehen. Wir haben wenig bis gar kein Wissen über diese neuen Bedrohungen, und explorative Sicherheitsevaluierungen von JavaScript-basierten Webanwendungen werden durch den Mangel an zuverlässigen und skalierbaren Testverfahren behindert. In dieser Arbeit gehen wir diese Herausforderungen an, indem wir JAW vorstellen, ein statisch-dynamisches Framework zur Untersuchung client-seitiger Schwachstellen im großen Maßstab, wobei wir uns besonders auf client-seitiges Request Hijacking und DOM Clobbering konzentrieren und deren Muster, Verbreitung und Auswirkungen in freier Wildbahn untersuchen. Wir instanziieren JAW auf über einer halben Million Seiten von 10K Top-Websites und verarbeiten insgesamt über 56B Zeilen Code, was zeigt, dass diese neuen Varianten im Web allgegenwärtig sind. Wir demonstrieren die Auswirkungen dieser Schwachstellen durch die Konstruktion von Proof-of-Concept-Exploits, die die Ausführung von beliebigem Code, das Entweichen von Informationen, offene Umleitungen und CSRF auch gegen beliebte Websites ermöglichen. Schließlich überprüfen und bewerten wir die Annahme und Wirksamkeit bestehender Gegenmaßnahmen gegen diese Angriffe, einschließlich Eingabevalidierung und browserbasierte Lösungen wie SameSite-Cookies und Content Security Policy.

Abstract

The recent rapid evolution of client-side technologies have introduced new variants of traditional security issues that now manifest exclusively on client-side JavaScript programs. We have little-to-no knowledge of these new emerging threats, and exploratory security evaluations of JavaScript-based web applications are impeded by the scarcity of reliable and scalable testing techniques. In this thesis, we address these challenges by presenting JAW, an open-source, static-dynamic framework to study client-side vulnerabilities at scale, focusing particularly on client-side request hijacking and DOM Clobbering vulnerabilities where we investigate their patterns, prevalence, and impact in the wild. We instantiate JAW on over half a million pages of top 10K sites, processing over 56B lines of code in total, showing that these new variants are ubiquitous on the Web. We demonstrate the impact of these vulnerabilities by constructing proof-of-concept exploits, making it possible to mount arbitrary code execution, information leakage, open redirections and CSRF also against popular websites that were not reachable through the traditional attack vectors. Finally, we review and evaluate the adoption and efficacy of existing countermeasures against these attacks, including input validation and browser-based solutions like SameSite cookies and Content Security Policy.

Background of this Dissertation

This dissertation is based on the following four papers discussed below [P2, P1, P4, P3]. All four papers have been accepted and published at top peer-reviewed conferences in the field of IT security. The author of this thesis contributed to all these research projects as the leader and main author. However, in this section we highlight parts that were conducted by others.

Chapter 3 is based on our work [P1] published at USENIX Security 2021, where we presented a new system to automatically detect and study vulnerabilities in client-side JavaScript programs, focusing on a new variant of Cross-Site Request Forgery (CSRF) attacks known as client-side CSRF [1]. Giancarlo Pellegrino commented and contributed to the writing of the paper in his role as the author's supervisor.

In Chapter 4, we extend our study of client-side CSRF, covering new browser APIs and request types that can be hijacked in our detection tool, and present a large-scale evaluation of the threat landscape posed by request hijacking vulnerabilities in the wild. This work is based on our paper accepted at IEEE S&P 2024 [P2]. Thomas Barber implemented the patched version of Firefox to enable dynamic taint tracking of JavaScript request-sending instructions, which is why we do not discuss implementation details for this part in the thesis. Both Giancarlo Pellegrino and Thomas Barber contributed to the writing of the paper.

We present our work published at IEEE S&P 2023 [P3] in Chapter 5, where we explore the applicability of our vulnerability detection system to code-less injection attacks, focusing on DOM Clobbering. Our study covers three dimensions: a systematic study of the attack surface, a large-scale measurement of vulnerable websites, and a comprehensive evaluation of existing defenses. Giancarlo Pellegrino provided guidance in the writing process of the paper.

Finally, Chapter 6 is based on our publication at IEEE S&P 2022 [P4], where we shift our focus from attacks to defenses. Specifically, we conduct a systematic and comprehensive study of SameSite cookie policies, presenting a longitudinal evaluation of their adoption and efficacy in the wild. Giancarlo Pellegrino guided the paper writing in his capacity as the supervisor.

[P1] Khodayari, S. and Pellegrino, G. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In: *USENIX Security Symposium.* 2021.

[P2] Khodayari, S., Barber, T., and Pellegrino, G. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web. In: *IEEE S&P Symposium.* 2024.

[P3] Khodayari, S. and Pellegrino, G. It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses. In: *IEEE S&P Symposium.* 2023.

[P4] Khodayari, S. and Pellegrino, G. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In: *IEEE S&P Symposium.* 2022.

Further Contributions of the Author

In addition to these primary works, the author of this thesis was also able to contribute to two additional publications [S1, S2], of which [S2] was an extension of the work conducted for author's master thesis after joining CISPA.

[S1] Likaj, X., Khodayari, S., and Pellegrino, G. Where We Stand (or Fall): An analysis of CSRF defenses in Web Frameworks. In: *International Symposium on Research in Attacks, Intrusions and Defenses.* 2021.

[S2] Sudhodanan, A., Khodayari, S., and Caballero, J. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In: *Network and Distributed Systems Security Symposium.* 2020.

Acknowledgments

This thesis has evolved into a captivating adventure, and the credit for making it both achievable and enjoyable goes to the incredible individuals I have crossed paths with. Your contributions have turned this scholarly journey into a delightful and enriching experience.

Firstly, a heartfelt thanks to my exceptional advisor, Giancarlo Pellegrino, for being an awesome mentor and for his continued support and patience that have been invaluable throughout the years. I am extremely grateful to Juan Caballero and Avinash Sudhodanan for sparking my interest in Web security and helping me grow my technical skills in my early days of research during my master's studies. I would also like to extend my thanks to the reviewers of this thesis and my defense committee, Martin Johns, Bernd Finkbeiner, Martina Maggio, and Bhupendra Acharya, for agreeing to review my work and for being there at the final stage of this journey.

I also want to give a big shoutout to my awesome lab mates, Giada Stivala, Andrea Mengascini, Gianluca De Stefano, Aleksei Stafeev, Sepehr Mirzaei, Lorenzo Cazzaro and Anthony Gavazzi, who have made my academic journey memorable with insightful discussions, shared successes, and the everyday camaraderie that added joy to our daily office routine.

In addition, I want to thank my peers at CISPA, Marius Steffens, Hamed Rasifard, Sanam Lyastani, Faezeh Nasrabadi, Jannis Rautenstrauch, Aurore Fass, Carolyn Guthoff, Alexander Ponticello, Riccardo Zanotto, Lea Gröber, Simon Anell, Masudul Hasan Masud Bhuiyan, Pouya Narimani, Sebastian Roth, Shubham Agarwal, Florian Hantke, Saleh Soudijani, Matteo Leonelli, and Giacomo Santato, for the wonderful time spent together, whether it was during conferences, at lunch time, or at social events. Furthermore, I would like to express my gratitude to CISPA faculties, Ben Stock, Katharina Krombholz, and Cristian-Alexandru Staicu, for our fruitful discussions, Zoom meetings during COVID that helped me retain my sanity, and the little moments during lunch that created an enjoyable working atmosphere. I am deeply thankful to everyone I had the privilege of meeting at CISPA, as you all contributed to creating such an unforgettable experience.

A second round of applause also goes to my co-authors: Giancarlo Pellegrino, Thomas Barber, Juan Caballero, Avinash Sudhodanan, and Xhelal Likaj, for their collaborative efforts.

Lastly, a heartfelt thank you to my family, whose unwavering support and love have been instrumental in my success throughout my PhD.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Client-side technologies are evolving at a rapid pace. Since the advent of Web 2.0, an increasing number of online services, from email to collaborative document editing, are accessible directly through web browsers, ensuring a seamless experience across various devices and operating systems. Thanks to these advancements, more and more web applications can offer dynamic and interactive features without the need for frequent page reloads, as processing tasks are offloaded to the client side, leading to faster response times and improved scalability.

Unfortunately, these numerous benefits have not been without a cost. The growing complexity of client-side programs has given rise to new variations of known security issues, now exclusively manifesting on the client side (e.g., [1–4]). Simultaneously, the continuous introduction of new JavaScript APIs have expanded the threat landscape for Web attacks. For example, recently introduced APIs such as the Push API [5] for push notifications and the Beacon API [6] for asynchronous requests have opened up new avenues for request forgery attacks. Even worse, these attacks are only the tip of the iceberg. Recent advancements in client-side technologies have introduced unforeseen interactions between client-side JavaScript programs and their execution environment, rising unexpected security problems like mutation-based Cross-Site Scripting (XSS) [7], script gadgets [8] and DOM Clobbering [10, 9]. Obviously, understanding and addressing these issues is imperative to fortify our digital infrastructure. However, traditional security testing methodologies are struggling to keep pace with the evolving complexities of these new emerging threats.

Detecting and studying new client-side vulnerabilities on the Web platform is not an easy task, as it requires the collection and analysis of hundreds or even thousands of webpages per real web applications. Unfortunately, exploratory security evaluations of JavaScript-based web applications are primarily impeded by the scarcity of reliable and scalable testing techniques. Firstly, there is no canonical representation for JavaScript code. Secondly, JavaScript programs operate in an event-driven manner, necessitating models that can effectively capture and incorporate this characteristic into the canonical representation. Thirdly, relying solely on static analysis is often not sufficient due to the dynamic nature of JavaScript programs [11–13] and their execution environment [14], prompting the need for hybrid static-dynamic analysis techniques. Finally, a significant portion of code on webpages comprises JavaScript libraries, and subjecting them to repeated analyses results in inefficient models poorly suitable for detecting vulnerabilities.

In this thesis, we address these challenges by proposing *Hybrid Property Graphs* (HPGs), a canonical representation for client-side JavaScript programs that captures both static and dynamic program behaviors. Inspired by prior work [15], we employ property graphs for model representation and use graph traversals to find security-sensitive program instructions that use data values from attacker-controllable inputs, such as instructions that create and send network requests. We implement our approach into JAW, an open-source security testing framework for the client-side of web applications at scale. Starting from a seed URL, JAW monitors the program execution, collects web resources, and instantiate an HPG for each webpage. Beyond vulnerability detection, HPGs offer the capability to query program properties, enabling security analysts to interactively explore security-related program characteristics, such as secure and insecure code patterns. Having developed the JAW framework, we use it as the vehicle for our large-scale evaluation of emerging client-side Web vulnerability classes, particularly focusing on vulnerable program behaviors, prevalence of vulnerabilities, and their impact. Our large-scale measurements enable up-to-date risk estimation, supporting higher readiness level for risk shifts in the Web ecosystem.

The first two parts of this thesis center around Cross-Site Request Forgery (CSRF) [17, 1, 16] attacks, where an attacker tricks a user's web browser into performing unwanted actions on a trusted website where the user is authenticated, such as changing account settings [18], making illicit financial transactions [19], or performing other sensitive operations on behalf of the victim [21, 20]. In this thesis, we study new variants of request forgery attacks that abuse the client-side code. First, we focus on attacks that hijack asynchronous request-sending APIs [23, 22], known as client-side CSRF [1]. We evaluated the prevalence of client-side CSRF vulnerabilities among all (i.e., 106) web applications from the Bitnami catalog [24], covering over 228M lines of JavaScript code, and discovered 12,701 forgeable client-side requests affecting 87 web applications in total. Then, we explore the threats originating from the hijacking of other client-side request-sending APIs. Particularly, we systematize request hijacking vulnerabilities and the resulting attacks by reviewing browser API capabilities and Web specifications, identifying 10 distinct vulnerability variants, including seven new ones. Then, we instantiate JAW on the top of the Tranco top 10K sites, performing, to our knowledge, the first investigation into the prevalence of request hijacking flaws in the wild. Our study uncovers that request hijacking vulnerabilities are ubiquitous, affecting 9.6% of the top 10K sites. We demonstrated the impact of these vulnerabilities by constructing 67 proof-of-concept exploits across 49 sites, making it possible to mount arbitrary code execution, information leakage, open redirections and CSRF also against popular websites like Microsoft Azure, Reddit, Indeed and Starz.

In the third part of this thesis, we demonstrate the applicability of JAW to other vulnerability classes, i.e., we study attacks that abuse the execution environment of client-side JavaScript programs to achieve code execution. Specifically, we focus on a namespace confusion vulnerability known as DOM Clobbering [25, 10, 9, 26], where attackers confuse a web application by injecting HTML elements whose name matches the name of sensitive JavaScript variables, ultimately overshadowing their value. Starting with a dynamic analysis of 19 different mobile and desktop browsers, we systematized DOM Clobbering attacks, uncovering 31.4K distinct markups that use five different techniques to unexpectedly overwrite JavaScript variables in at least one browser. Then, we used our systematization to identify and characterize program instructions that can be overwritten by DOM Clobbering, and used it to extend JAW, enabling it to detect clobberable data flows in webpages. We used JAW to assess the prevalence of DOM Clobbering vulnerabilities on Tranco top 5K sites, identifying a total of 9,467 vulnerable data flows across 491 affected sites.

Finally, our analysis extends to the assessment of existing defenses, emphasizing the critical need for effective and compatible mitigation strategies against these evolving threats. For example, we examine the robustness of HTML sanitizers and Content Security Policy [27] against DOM Clobbering. Furthermore, we review and evaluate the adoption and efficacy of existing countermeasures against request forgery attacks, including input validation and browser-based solutions. For example, the `SameSite` cookie attribute [28] allows developers to choose from three cookie policies (None, Lax, and Strict), depending on whether cookies should be attached to different cross-site request contexts or not. Recently, chromium-based browsers restricted cookies' scope to a same-site context by default, known as the Lax policy [29, 30]. To study this change, we conducted a set of large-scale, longitudinal, both automated and manual measurements of the Alexa top 1K, 10K, 100K, and 500K sites across the main rollout dates of the SameSite policies, covering both SameSite usage and cross-site functionality breakage caused by the new default policy. Also, we performed an extensive evaluation of threats against the new Lax-by-default policy's effectiveness, looking at the adequacy of the coverage provided by the Lax policy and bypass caused by website developers' mistakes. Our study observes a significant mismatch between the request contexts protected by Lax and the ones actually used by websites in the wild, making it possible to perform cross-site attacks also against popular websites such as Tumblr, Twitch, SoundCloud, Mailchimp, and Pixiv. Even when using Lax or Strict policies, much of their effectiveness depends on developers' awareness of SameSite policies' implications, who could introduce vulnerabilities or inconsistent policies, leading to SameSite policy bypasses.

1.1 Problem Statement

In today's dynamic and interconnected digital landscape, Web security stands as a paramount concern. The exponential growth of web applications [31] and the increasing complexity of online platforms have given rise to a myriad of security challenges (e.g., [33, 36, 35, 32, 34, 37, 3, 4, S2]). As organizations and individuals alike become more reliant on web-based technologies, the vulnerabilities associated with these systems become a focal point for malicious actors seeking unauthorized access, data breaches, and service disruptions.

This problem statement chapter explores the web of challenges faced in the context of reusable and scalable security testing for web applications, focusing on client-side JavaScript programs. By unraveling the complexities of these challenges, we pave the way for innovative solutions and strategies that can safeguard our online infrastructure. In this chapter, we set the stage for a comprehensive exploration of the problems, providing context and clarity to the issues that demand our attention in the pursuit of a secure and resilient Web.

1.1.1 Automatic Detection of Client-side CSRF (RQ1)

Client-side Cross-Site Request Forgery (client-side CSRF) is a new breed of CSRF vulnerabilities affecting modern web applications. Like the more traditional CSRF, with a brief visit to a malicious URL, an adversary can trick the victim's browser into sending an authenticated security-sensitive HTTP request on the user's behalf towards a target website without user's consent or awareness.

In the traditional CSRF, the vulnerable component is the server-side program, which cannot distinguish whether the incoming authenticated request was performed intentionally, also known as the *confused deputy problem* [39, 38]. CSRF is typically solved by adding a pseudo-random unpredictable request parameter, preventing forgery (see, e.g., [17]), or by changing the default browsers' behavior and avoiding the inclusion of HTTP cookies in cross-site requests (see, e.g., [40, 41]). In the client-side CSRF, the vulnerable component is the JavaScript program instead, which allows an attacker to generate arbitrary requests by modifying the input parameters of the JavaScript program. As opposed to the traditional CSRF, existing anti-CSRF countermeasures (see, e.g., [17, 40, 41]) are not sufficient to protect web applications from client-side CSRF attacks, making it critical to detect these class of vulnerabilities.

Client-side CSRF is very new—with the first instance affecting Facebook in 2018 [1]—and we have little-to-no knowledge of the vulnerable behaviors, the severity of this new flaw, and the exploitation landscape. Studying new vulnerabilities is a challenging task, requiring the analysis of hundreds of webpages per real web applications. However, such analyses are largely hindered by the scarcity of reusable and scalable tools suitable for the detection and analysis of vulnerable JavaScript behaviors. In this thesis, we address these challenges by proposing a new technique and a corresponding tool for security testing of client-side JavaScript programs *at scale*. In the following, we break down our research question in well-defined challenges.

RQ1.1: Static Representational Models. JavaScript programs are incredibly challenging to be analyzed via static analysis. For example, prior work have proposed inter-procedural control flow graphs [43, 42], data flow dependency graphs [44, 45], type analyzers [47, 46, 48], and points-to analysis [50, 49]. Unfortunately, these approaches provide ad-hoc representation of programs, each focusing on an individual aspect that is alone not sufficient to study client-side CSRF. Recently, we have seen new ideas unifying static representations with *code property graphs* (CPGs) [51, 15]. However, these new ideas are not tailored to JavaScript's nuances, such as the asynchronous events [45], or the execution environment [14]. To date, there are no models for JavaScript that can provide a canonical representation to conduct both detection and exploratory analysis of the code.

To address this challenge, we propose Hybrid Property Graphs (HPGs), a uniform canonical representation for JavaScript source code, similarly as code property graphs for C/C++ [15] and PHP [51]. We describe our approach in §3.1.

RQ1.2: Vulnerability-specific Analysis Tools. Over the past years, there have been a plethora of approaches to detect vulnerabilities in client-side JavaScript programs. To date, these approaches have been mainly applied to XSS [32, 52, 53, 37, 54], or logic and validation vulnerabilities [61, 60, 55, 57–59, S2, 56], resulting in tools that are rather tightly *coupled* with the specific analysis of the vulnerability. Thus, researchers seeking to study new client-side vulnerabilities like client-side CSRF are forced to reimplement those approaches rediscovering tweaks and pitfalls.

We address this challenge by defining HPGs and developing JAW, enabling us to perform a variety of security tasks, i.e., detection and exploratory analyses of client-side vulnerabilities like client-side CSRF and DOM Clobbering. We believe that decoupling the code representation (the graph) from the analysis (traversals) renders JAW more suitable for reuse, similarly to other CPG-based approaches [51, 15]. We demonstrate this as a part of our measurements and security analyses for various vulnerability classes (Cf. Sections 1.1.2 to 1.1.4).

RQ1.3: Event-based Transfer of Control. Existing unified representations such as CPGs [51, 15] assume that the transfer of control happens *only* via function calls, an assumption no longer valid for JavaScript. In JavaScript, the transfer of control happens also via events which either originate from the environment, e.g., mouse events, or are user-defined, as shown in Listing 2.1. When an event is dispatched, one or more registered functions are executed, which can change the state of the program, register new handlers, and fire new events. Representing the transfer of control via event handlers is fundamental for the analysis of JavaScript programs. As we will show in Sections 3.1 and 3.2, HPGs in our approach capture JavaScript nuances such as event-based transfer of control by proposing the Event Registration, Dispatch and Dependency Graph (ERDDG).

RQ1.4: Dynamic Web Execution Environment. JavaScript programs rely on many dynamic behaviors that make it challenging to study them via pure static analysis. A typical example is the dynamic code loading [12]. In essence, JavaScript programs can be streamed to the user's web browser, just like other resources. Thus, contrary to the assumption in most static analysis approaches, the entire JavaScript code may not be available for the analysis [11]. Another example is the interaction between JavaScript and the DOM tree. Consider, for example, two variables containing the same DOM tree node; however, the content of one variable is fetched via `document.querySelector("input")` and the other by `document.form[0].input`. In such a case, it is often important to determine whether the two variables point to the same object (i.e., point-to analysis). However, it can be considerably hard to determine this by looking at the source code, as DOM trees are often generated by the same program. As we will show in §3.1.2, HPGs capture the dynamics of the web execution environment of client-side JavaScript programs via both snapshots of the DOM environment and traces of JavaScript events.

RQ1.5: Shared Third-party Code. Most modern web applications include at least one third-party JavaScript library [62], such as jQuery [63], to benefit from their powerful abstractions over the low-level browser APIs. Detection of client-side CSRF requires the ability to determine when the program performs HTTP requests, also when the program delegates low-level network operations to libraries. Similarly, library functions can be part of the data flows of a program.

To date, existing approaches are highly inefficient as they include the source code of libraries in the analysis. We observe that external libraries account for 60.55% of the total JavaScript lines of code of each web page[1], thus requiring existing techniques to re-process the same code even when visiting a new page of the same web application. An alternative approach consists of creating hand-crafted models of libraries (see, e.g., [64]). While such an approach is effective

[1]We calculated the fraction of library lines of code over the testbed web applications of §3.3.1 using the crawler and the configuration of the data collection phase of §3.2.1.

when modeling low-level browser APIs, it does not scale well to external libraries. First, external libraries are updated more frequently than browser APIs and second, there are many alternative libraries that a JavaScript program can use [65]. However, as we will show in Sections 3.1 and 3.2, JAW can generate reusable symbolic models of external libraries, that will be used as proxy in our HPGs.

1.1.2 Studying Request Hijacking Vulnerabilities in the Wild (RQ2)

Request forgery attacks have been one of the most critical threats to web applications since the early days of the Web, where attackers trick victims' browsers into making authenticated, security-sensitive HTTP requests [17, 67, 66, 16, 20]. The fundamental vulnerability enabling these attacks is the inability of the server-side component to distinguish unintentional from intentional requests (i.e., the confused deputy flaw [39, 38]), allowing maliciously-forged requests to cause a persistent state change of the web application, such as resetting passwords [18, 19] or deleting data from databases [68]. The recent rapid evolution of client-side technologies has introduced more subtle variants of request forgery vulnerabilities where attackers no longer rely on the confused deputy flaw but instead exploit insufficient input validation vulnerabilities in the client-side JavaScript program to hijack outgoing requests. The research community has only recently started exploring these vulnerabilities, mainly focusing on client-side CSRF [1, 69], for which we propose a corresponding detection and analysis technique as a part of RQ1 (see, i.e., [P1]). Unfortunately, client-side CSRF is only one instance of the larger issue of request hijacking in web applications, as other types of outgoing HTTP requests exist within JavaScript programs that attackers can hijack, which, to date, are largely unexplored.

Client-side request hijacking vulnerabilities occur when a JavaScript program uses attacker-controllable inputs, such as URL parameters, to send network requests. A closer look at prior work [1, 69] reveals that they primarily focus on *asynchronous* requests generated via the `XMLHttpRequest` [22] and `fetch` [23] APIs, missing other types of outgoing requests and APIs that a JavaScript program can use, e.g., push notifications, web sockets, and server-sent events, including the `sendBeacon` API [6], which accounts for over 35.3% of API calls for asynchronous requests[2]. As a result, we still lack a comprehensive exploration and understanding of this threat on the Web.

To address this research question, we start from the answers of RQ1 in §1.1.1 for the analysis of JavaScript programs and the client-side CSRF vulnerability, and provide a component-centric security analysis of requests induced in modern browsers and their potential to be hijacked by cross-site adversaries. Particularly, we focus on the following research questions:

RQ2.1: Browser Capabilities and Attack Systematization. The recently proposed client-side CSRF vulnerability [1, 69] allows attackers to generate arbitrary HTTP requests by manipulating JavaScript program input parameters. However, client-side CSRF is just one instance of the broader issue of request hijacking in client-side code, i.e., JavaScript programs can perform different types of requests (e.g., asynchronous vs top-level requests or socket connections) using numerous APIs (e.g., `XMLHttpRequest` vs `sendBeacon`), which presents a diverse threat landscape. In this thesis, we take a step back and study client-side request hijacking vulnerabilities. First, we look at various browser methods and APIs for sending requests, and label each with specific capabilities (e.g., accept `javascript` URIs, allow setting the request body, etc). Then, we review existing literature and conduct a comprehensive threat modeling analysis, systematically assessing the security risks that emerge when an attacker can manipulate various fields of request-sending APIs. We address this RQ in §4.1.

RQ2.2: Detection, Prevalence, and Impact. Despite being aware of client-side CSRF since 2018 [1], we still lack a clear understanding of its prevalence and severity across the Web

[2]We calculated the API usage over Tranco top 10K sites (see §4.1.3) using the data collection setting detailed in §4.2.1 and §4.3.

on a large-scale. Unsurprisingly and by extension, we have little-to-no information about the overall impact and pervasiveness of the broader issue of request hijacking in real websites. In this thesis, we aim to fill this gap by quantifying the prevalence of request hijacking in the wild, identifying vulnerable behaviours, and investigating their impact to gain insights into the underlying issues and factors that affect the security posture of web applications. We address this RQ in Sections 4.2 and 4.3.

RQ2.3: Defenses and Effectiveness. While numerous research efforts studied request forgery countermeasures (e.g., [17, 66, P4, 70, 72, 71, 19]), their focus has been only the traditional request forgery attacks that abuse the confused deputy flaw, and hence, we still lack a comprehensive understanding of the protective coverage of various defenses mechanisms against client-side variants of the request hijacks. As the final part of this chapter, we systematically assess existing defenses and their efficacy leveraging data collected from the previous answers. In particular, we measure the efficacy and adoption of browser-based policies, such as CSP [27], COOP [73] and COEP [74], and examine the discovered vulnerabilities to uncover insecure input validation patterns and practices adopted by developers. We address this RQ in §4.4.

1.1.3 Understanding DOM Clobbering Attacks and Defenses (RQ3)

The third part of this thesis demonstrates the applicability of JAW to code-less injection attacks. Arbitrary client-side code execution has been one of the major threats against web applications since the early days, traditionally achieved by injecting JavaScript code into vulnerable pages, e.g., Cross-Site Scripting (XSS) attacks [81, 77, 76, 7, 78, 32, 75, 79, 80, 54, 82]. However, over the past 20 years, the growth of Web technology has introduced unforeseen interactions between JavaScript programs and the execution environment that can result in execution of arbitrary code without injecting JavaScript but only by injecting seemingly harmless HTML markups. The research community has only recently begun studying the security of these interactions, mainly focusing on small code fragments called script gadgets [8] that react to injected HTML markups and transform it into code. Yet, script gadgets are only the tip of the iceberg, and other complex interactions exist that attackers can abuse to hijack the program execution. However, these interactions have received little attention in research thus far.

DOM Clobbering is a vulnerability that originates from a naming collision between JavaScript variables and named HTML markups, where browsers replace pre-existing content of an undefined variable with an HTML element when the variable name and the element's `name` (or `id`) attribute match. Developers unaware of such behavior may use the content of undefined variables for sensitive operations, such as URLs for fetching remote content, and attackers can exploit it by injecting markups with colliding names. In this thesis, we systematically study the various DOM Clobbering attack techniques in modern browsers, the severity and prevalence of DOM Clobbering vulnerabilities, and the effectiveness of existing defenses. In particular, we answer the following research questions:

RQ3.1: DOM Clobbering Attack Techniques. When looking at the evolution of DOM Clobbering attack markups, we observe a consistent complexity growth, starting from a single HTML element [83] that can overwrite a variable, evolving with pairs of HTML tags [10, 26] that clobber properties of objects (2013-2015), and then advancing into a wide variety of browser-specific combinations of different HTML tags and attributes that can not only overwrite variables, but also native DOM objects (2015-2018) [84, 87, 85, 86], nested object properties, and loop elements (2018-2022) [88, 89, 9]. Despite the growth of markups' complexity, the exploration of the attack surface has not been conducted systematically, and to date, many of the possible combinations of tags, attributes, markup relationships and possible JavaScript object manipulations are not considered. In this thesis, we intend to fill this gap and exhaustively explore such an attack surface by generating clobbering markups and testing modern mobile and desktop browsers automatically. We answer this RQ in §5.1.

RQ3.2: DOM Clobbering Patterns and Prevalence. While the existence of DOM Clobbering is known for more than a decade [10, 83], we still do not have a measurement about the prevalence, impact, and code patterns of this vulnerability. In this thesis, we intend to quantify the prevalence of DOM Clobbering in the wild, identify vulnerable behaviours, and examine their impact to shed some light on possible causes and factors hampering web applications' security. Finally, we intend to review developers' mistakes and identify vulnerable and secure coding patterns that can fix those issues. We answer this RQ in §5.2.

RQ3.3: Mitigations and Kill Switch. As a final question, we look at the defenses, their effectiveness, and cost-benefit, leveraging the data generated and collected from the previous answers, i.e., DOM Clobbering markups, vulnerability prevalence, and developer mistakes. In particular, we intend to evaluate the cost-benefit trade-off resulting from disabling named property accesses [90, 91] in browsers and thoroughly assess existing solutions such as HTML sanitization [78], Content-Security Policy (CSP) [88, 92] and freezing object properties [93] against DOM Clobbering. We address this RQ in §5.3.

1.1.4 Studying the Effectiveness of SameSite Policies (RQ4)

In the last part of this thesis, we build upon the insights obtained from our examination of preceding research questions—namely, the prevalent existence of cross-site vulnerabilities such as client-side request forgery vulnerabilities on the Web, and shift our emphasis from attacks to defenses. Specifically, we focus on the recently proposed Lax-by-default SameSite cookie policy [95, 94], studying its adoption and effectiveness against cross-site attacks.

Despite the recent surge of attention toward SameSite cookies, little has been done to understand how they are used by application developers (see, e.g., [94]) and the hurdles when using them in practice together with different web application functionalities. This thesis takes the first step in this direction and explores the security and effectiveness of SameSite cookies by quantifying their usage in the wild, and also by systematizing known and introducing new web attacks that can circumvent SameSite cookies, and ultimately compromise web applications' security. Specifically, we answer the following research questions:

RQ4.1: Trend Analysis of SameSite Cookie Usage. The main benefit of the `SameSite` attribute is the new default cookie policy, which can disrupt existing websites. To help developers transition to the new default policy, Google introduced three gradual changes to Chrome, introduced in 2016, 2019, and 2020 [94]. First, in April 2016, Chrome 51 introduced support for the new attribute without modifying the default policy. Later, in September 2019, Chrome 77 started showing console warning messages for cookies without the `SameSite` attribute. The final step of this transition took place in 2020, with Chrome 80. Specifically, in February 2020, Chrome set Lax+POST the new default policy. However, shortly after, Google rolled it back (April 2020) to ease developers' transition to the new policy in light of the COVID-19 pandemic. The Lax-by-default was then restored in July 2020 with Chrome 84 [95, 94].

When looking at the SameSite rollout timeline, one of the first questions we intend to address is understanding the long rollout approach's effectiveness by quantitatively measuring how website developers adapted to the upcoming new policies across the main rollout milestones. In particular, we intend to quantify the websites that picked one of the three pre-defined SameSite policies and those that rely on the new default behavior. Also, as supporting the `SameSite` attribute requires modifying the server-side component's code, developers may make mistakes and use non-existing policies (see, i.e., [96]), inadvertently resulting in deploying a different policy than the intended one. We answer these questions in §6.1.

RQ4.2: Functionality Breakage. Starting from July 2020, the new default policy for cookies without the `SameSite` attribute is Lax [95, 94]. Changing the default policy for cookies can interfere with cross-site communications between web services. For example, services such as Single Sign-On (see, i.e., [99, 97, 100, 98]) rely on asynchronous authenticated requests to exchange authentication tokens. Another example is Oracle APEX, a web application that can run inside iframes (see, i.e., [101]). According to the Lax policy, browsers will not include cookies in requests originated in iframes, preventing Oracle APEX from sending authenticated requests. Despite this anecdotal evidence, we lack a comprehensive overview of the websites whose cross-site functionalities may no longer work due to the new default policy. This thesis intends to fill this gap, by identifying types of affected functionality and providing a first quantification of the affected websites. We answer these questions in §6.2.

RQ4.3: Lax Adequacy and Threats. The radical change introduced by the `SameSite` attribute is that browsers no longer include cookies to all cross-site requests by default, but only to those originating from predefined lists of same-site contexts. These lists capture those contexts typically used in cross-site attacks. For example, the Lax policy forbids including cookies in cross-site POST submissions as they are typically used in CSRF attacks. However, prior works (e.g., [21, 102, S2]) suggest that the Lax policy's coverage may not be adequate. Developers do not strictly obey the distinction between safe and unsafe HTTP methods and implement state-changing or state-leaking operations via, for example, GET requests.

One of the questions that we intend to answer in this thesis is the adequacy of the new same-site policies and their effectiveness. In particular, we want to focus on the tension between the contexts protected by the new policies and the contexts used by existing websites to implement security-sensitive operations. Second, non-academic security reports (see, e.g., [103–105]) have shown that, in certain cases, implementation mistakes can cause a bypass of the protection offered by the new SameSite policy. For example, web applications may not distinguish between different HTTP methods when processing incoming HTTP requests, allowing adversaries to forge protected requests, such as POST, by changing the HTTP method to non-protected methods, such as GET [104, 105]. However, it is somewhat unclear whether these vulnerabilities are outliers or widespread security problems. In this thesis, we intend to provide a comprehensive evaluation of threats against web applications that rely on the same-site policies and determine their severity by looking for their prevalence in the wild. We answer these questions in §6.3.

RQ4.4: Browser Inconsistencies and Web Frameworks' Defaults. Our last research question investigates how consistent different (i) web browsers and (ii) web frameworks apply SameSite cookies on cross-site requests, and what are the divergent aspects among them. For example, the default policy in Chrome is Lax, whereas Firefox enforces the None policy. Even when browsers enforce a default Lax policy, web frameworks' APIs may downgrade it to None by default.

1.2 Contributions

In this section, we provide an overview of our contributions for each of the research questions outlined in §1.1.

RQ1: Automatic Detection of Client-side CSRF. In this thesis, we perform the first systematic study of client-side CSRF, a new variant of CSRF affecting the client-side JavaScript program, and present a taxonomy of forgeable requests considering two features, i.e., request fields, and the type of manipulation. Furthermore, we present *hybrid property graphs*, a single and coherent representation for the client-side of web applications, capturing both static and dynamic program behaviors. We implemented our approach into a multi-purpose and reusable framework, JAW, that detects client-side CSRF by instantiating a HPG for each web page, starting from a single seed URL.

In order to assess the efficacy and practicality of our approach and to study client-side CSRF vulnerabilities, we evaluated JAW with over 228M lines of JavaScript code in 106 popular applications from the Bitnami catalog [24], identifying 12,701 forgeable requests affecting 87 applications, out of which we created working exploits for 203 requests of seven applications.

Finally, we publicly release the source code of JAW[3] to support the future research effort to study vulnerable behaviors of JavaScript programs.

RQ2: Studying Request Hijacking Vulnerabilities in the Wild. We undertake, to the best of our knowledge, the first evaluation of client-side request hijacking vulnerabilities in the wild, covering three main aspects: a systematic exploration of the attack surface, a measurement of vulnerable websites, and a thorough review and evaluation of request hijacking defenses. Starting from a comprehensive survey of browser API capabilities, we systematically examine potential attacks when attackers manipulate one or more inputs of request-sending APIs, covering various types of sensitive requests in modern browsers.

Then, we propose JAW-v2, a client-side request hijacking detection tool that uses a combination of hybrid program analysis [P1] and in-browser dynamic taint tracking [107, 106] for the discovery of potentially-vulnerable data flows and dynamic analysis with API instrumentation [108] for the automated vulnerability verification. We instantiate JAW-v2 against the Tranco top 10K websites to quantify the prevalence and impact of client-side request hijacking in the wild, processing over 32.4B lines of JavaScript code across 11.5M scripts and 339K webpages. Finally, we identify and evaluate defenses, covering built-in countermeasures offered by browsers and custom defenses implemented by applications at code-level. In particular, we assess the efficacy and adoption of browser policies like Content Security Policy (CSP) [109, 27] and Cross-Origin Opener Policy (COOP) [73], and examine the client-side code to identify insecure input validation practices adopted by developers against request hijacking attacks.

Our results show that the attack surface of client-side request hijacking vulnerabilities is large, with a total of 10 different variants across six request types, of which seven variants are previously unknown, notably hijacking requests of push notifications, window navigations, EventSource, and WebSockets. Furthermore, client-side request hijacking data flows are ubiquitous, affecting 9.6% of the Tranco top 10K websites, with a total of 202K instances across 17.9K webpages. Of these, the new vulnerability types and variants constitute a significant fraction (36.1%), with over 73.3K instances. To demonstrate the significance of these vulnerabilities, we created 67 proof-of-concept exploits in 49 sites, including popular ones like Microsoft Azure, Indeed, Starz, Google DoubleClick, TP-Link, and Reddit, leading to critical consequences such as arbitrary code execution, CSRF, information leakage and open redirections. Finally, the analysis of existing countermeasures suggest that each can only mitigate a fraction of attacks. For example, CSP cannot mitigate over 41% of the information leakage and XSS exploitations of the request hijacking, and COOP and COEP cannot mitigate over 93% and 94.7% of the total request hijacks, respectively. Our results show that developers can fix request hijacking vulnerabilities at code level, and we identify eight insecure input validation patterns to avoid.

RQ3: Understanding DOM Clobbering Attacks and Defenses. We conduct the first comprehensive and systematic study of DOM Clobbering, covering vulnerability, attack techniques, detection, prevalence, impact, and defenses. Starting from a comprehensive survey of prior DOM Clobbering vulnerabilities, we systematically generate candidate DOM Clobbering markups, and automatically test desktop and mobile browsers against them, covering all known HTML tags and attributes–including custom ones–and markup relationships.

Then, we propose JAW-v3, a DOM Clobbering detection tool that amalgamates static program analysis, i.e., [P1], for the discovery of potentially-vulnerable data flows, with forced execution, i.e., [110], for the automated vulnerability verification, leveraging the generated DOM Clobbering markups. We instantiate JAW-v3 against the Tranco top 5K websites to quantify the prevalence and impact of DOM Clobbering vulnerabilities, processing, in total, over 24.6B lines of JavaScript code across 18.3M scripts and 205.6K webpages. Finally, we identify, review, and evaluate defenses, covering existing countermeasures and secure code patterns. In particular, we first precisely measure the cost-benefit trade-off of browser-level countermeasures and thoroughly

[3]https://soheilkhodayari.github.io/JAW

test HTML sanitizers. Then, we review the vulnerable code discovered by JAW-v3, identify common developer mistakes, and distill a list of secure coding patterns.

Our findings underscore the extensive attack surface associated with DOM Clobbering vulnerabilities, with only 481 out of 31,432 generated DOM Clobbering markups are currently known, and the remainings are either previously-unknown instances (148) or variants of known cases (30,803). When grouping markups by browser behaviors, we observe ten different behavioral groups, showing that while most of the attacks are shared across browsers, many others work with specific browsers only. In addition, our experiments discovered 114 new native browser APIs that these markups clobber in at least one browser, including security-sensitive APIs like cache storage [111] and trusted types [112].

Second, DOM Clobbering vulnerabilities are quite widespread, affecting 9.8% of the top 5K websites, including popular sites like GitHub, Fandom, Trello, Vimeo, TripAdvisor, WikiBooks and AliExpress, leading to severe consequences such as arbitrary code execution, client-side CSRF [P1], and open redirections [114, 113].

Third, when looking at the browser-level defenses, disabling named property accesses can cause more breakage, i.e., 2,561 websites, than benefits, i.e., 491 vulnerable websites, with a cost-benefit ratio of 5.2:1 websites. In the absence of a browser-level fix, developers need to be particularly careful when choosing a countermeasure, as they balance protection with usability. For example, 55% of the most popular HTML sanitizers across the five most used web languages are vulnerable to at least one of the 31.4K clobbering markups by default. The remaining 45% sanitizers remove named properties, i.e., `id` and `name` attributes, which may interfere with the DOM manipulation operations. Also, our results show that CSP is insufficient because 85% of the discovered vulnerabilities can cause code execution without manipulating the `src` attribute. Finally, our results show that developers can fix vulnerabilities at the code level, and we identify eight distinct vulnerable code patterns to avoid and propose four secure patterns to fix them.

RQ4: Studying the Effectiveness of SameSite Policies. As the final part of this thesis, we perform, to our knowledge, the first security evaluation of the SameSite cookie policy, systematically studying the trend of its usage from June 2019 to March 2021 on the top 500K Alexa sites. First, we study the impact of the new default SameSite cookie policy, and present an overview of the number of affected services and websites by analyzing top 500 Alexa websites.

Then, we comprehensively review the threats against SameSite cookies, and identify seven known and propose three new threats inspired by prior work. We quantify the impact and prevalence of vulnerabilities of each threat in the wild by designing large-scale experiments. Furthermore, we conduct a behavioural analysis of 14 popular web browsers, identifying seven divergent ways on how browsers enforce the SameSite cookie policy. Finally, we analyze top five web frameworks of top five programming languages, and show that 24% of the frameworks offer APIs that, by default, downgrade the new default Lax protection offered by browsers.

Overall, this thesis provides the following insights about the current state of the `SameSite` attribute. As expected, after a rapid increment of sites using one of the SameSite policies around the enforcement dates, we now observe a rather moderate, yet steady, growth. Second, about 19% of the functionalities implemented via cross-site requests without an explicit SameSite policy does not work after the Lax-by-default enforcement, of which the vast majority are requests for advertisement online services. Third, uncustomizable, pre-packaged policies like the SameSite policy are making it particularly challenging for a significant fraction of websites to benefit from their protections without substantially revisiting websites' designs and implementations. For example, while the Lax policy can considerably reduce the attack surface of cross-site exploitations, we observed a significant mismatch between the cross-site request contexts covered by Lax and the ones used by websites in the wild, making it possible to perform cross-site attacks. Such a mismatch may suggest that a user-customizable, per-contexts same-site policy could be more beneficial for these websites. Fourth, even when using Lax or Strict policies, much of their effectiveness depends on developers who may introduce inconsistent or conflicting policies, leading to SameSite policy bypasses. Finally, we observed that popular mobile and desktop browsers exhibit inconsistent behaviors when processing and enforcing SameSite policies and handling exceptional cases.

1.3 Thesis Outline

This thesis is structured into eight chapters. Chapter 2 provides essential background information for a comprehensive understanding of this work. Chapter 3 introduces a security testing framework for automated detection and interactive exploration of client-side JavaScript vulnerabilities, particularly focusing on client-side CSRF. In Chapters 4 and 5, we apply and extend our analysis framework to the problem of client-side request hijacking and DOM Clobbering vulnerabilities, studying their prevalence and impact in the wild as well as the efficacy of existing countermeasures. Drawing from our insights into these attacks and defense mechanisms, Chapter 6 studies the adoption and effectiveness of SameSite cookie policies, especially in countering request forgery attacks. Chapter 7 discusses the related work, highlighting prior achievements and comparing this work with existing literature. Finally, Chapter 8 concludes this thesis and discusses the broader implications of our findings.

2
Technical Background

In this chapter, we present the foundational knowledge necessary to understand this work. We begin by presenting essential Web technologies such as HTML, JavaScript, and security-sensitive browser APIs, which are often exploited in the context of client-side Web attacks (§2.1), and follow up with a discusssion of the central security concept of the Web— the Same-Origin Policy (SOP). Then, we describe fundamental security problems for Web applications that bear relevance to this thesis (§2.2), such as Cross-Site Leaks (§2.2.1), Cross-Site Request Forgery or CSRF (Sections 2.2.2 and 2.2.3), and DOM Clobbering vulnerabilities (§2.2.4). Finally, we introduce SameSite-by-default cookies, an emerging security countermeasure with the potential to alleviate common variants of cross-site attacks such as CSRF and conclude this chapter (§2.3).

2.1 Foundational Concepts

In this section, we introduce the foundational concepts for the Web technology.

2.1.1 Hypertext Markup Language

Hypertext Markup Language (HTML) is a platform-independent language [115] that web browsers use to interpret and display Web content to users. It serves as the prevailing de-facto standard for resources accessible through the Web, and is standardized [116] and maintained by W3C [117] and WHATWG [118]. At its core, HTML uses a set of predefined tags to define the structure, layout, and presentation of text, images, links, and other multimedia elements on a webpage, serving as the building blocks for constructing interactive web applications. For instance, developers can use HTML anchor links to reference other webpages, or display forms to users, which they can fill with information and then send back to the application by clicking a submit button.

As the demand for interactive web applications continues to grow, HTML technology has and is undergoing a continuous evolution to cater to these ever-expanding needs. For example, a notable advancement is the incorporation of scripting languages like JavaScript and style sheets [119], which facilitated the creation of dynamic user experiences and functionalities such as form validation, real-time data updates, and interactive visual effects, by adding dynamic logic to the otherwise static client-side code [120]. Similarly, the recent HTML's built-in support for audio and video embedding, for which developers needed to rely on external plugins like Flash [121, 122] in the past, further underscores HTML's ongoing journey to accommodate contemporary Web development demands.

The Evolving Attack Landscape of HTML. The rapid integration of new features into HTML, such as the incorporation of JavaScript, has undeniably ushered in a plethora of conveniences and functionalities, enhancing the overall Web experience for both developers and users. However, this progression has not been without its compromises: an expanded attack surface that exposes web applications to a heightened risk of security issues, such as code injection vulnerabilities [76, 8, 32, 83, 54, 3].

2.1.2 JavaScript

JavaScript is an event-driven programming language for building modern and interactive web applications. It allows websites to respond to user interactions in real time, manipulate the content on the page, and create interactive features like animations, forms, and multimedia elements. Initially developed to bring client-side scripting to Netscape Navigator [123], JavaScript has evolved into a general-purpose programming language over the past years. For example, the advent of `Node.js` [125, 124] enabled JavaScript to be used not only as a client-side technology, but also for server-side programs and mobile applications.

Dynamic Features and Testability Challenges. JavaScript is an inherently dynamic language. It has prototype-based inheritance, dynamic property lookups and coercions, allowing variables to change types and values at runtime, as it follows the ECMAScript specification [126]. While this dynamism is advantageous for creating dynamic and interactive web content, they pose a considerable challenge when it comes to testing [127], particularly through methods like static analysis [53, 15, 128]. Static analysis tools rely on the ability to analyze code without executing it. As a result, accurately identifying potential vulnerabilities becomes a complex task, demanding specialized techniques that can navigate the intricate nuances of JavaScript's dynamic nature (e.g., [129, 11, 12, 14, 50]).

2.1.3 Document Object Model

The Document Object Model (DOM) [130] is a programming interface for web documents. It provides a tree-structured representation of the webpages, and allows programmatic access and manipulation to the tree with JavaScript. For example, the HTML DOM is accessible using the `Document` interface [131]. DOM APIs [132] provide access to various sensitive browser features such as tabs and windows, local storage mechanisms, and browser history, creating an appealing attack surface for hackers (e.g., [78, 8, 32, 54]).

2.1.4 Same-Origin Policy

Modern web browsers can simultaneously load and display resources from multiple websites. Whether through multiple open tabs or embedded frames from different sites, these resources can interact with one another. However, allowing unrestricted interaction can have critical security implications. For example, if a malicious attacker compromises a script, the compromised script could divulge the entire contents of a user's browser. The Same-Origin Policy (SOP) [134, 133] prevents this from happening by blocking read access to resources loaded from a different origin.

In general, SOP is a fundamental security mechanism that controls how documents and resources (e.g., scripts) on one Web *origin* can interact with documents and resources on another *origin*, with the *origin* being a combination of protocol, domain, and port. This way, a document hosted on `evil.com` cannot access the contents of `benign.com` because they belong to two different domains or origins, e.g., consider the case where users open `evil.com` and `benign.com` webpages in two different browser tabs.

2.1.5 HTTP Cookies

HTTP cookies are an intrinsic element of the Web that facilitates the synchronization of the state between clients and servers, who should communicate over the stateless HTTP protocol [135]. A cookie is a key-value pair along with a number of attributes that control when and where the cookie is used. Examples of such attributes are the expiration date, or the `Secure` attribute [136] which restricts the cookie to be sent exclusively over HTTPS. However, these benefits do not come without a trade-off: cookies are automatically transmitted to servers for all requests, including those made by third-party websites, enabling a number of Web attacks (e.g., [17, 137, 16, 102, 20]). The remainder of this chapter present these attacks and discusses countermeasures against them.

2.2 Vulnerabilities and Attacks

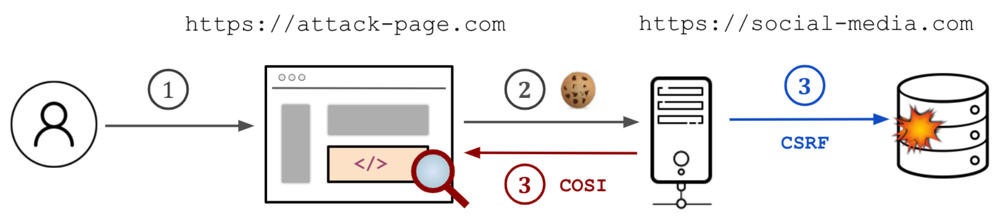
In this section, we provide the essential background information required to comprehend the vulnerabilities and attacks examined in this thesis.

2.2.1 Cross-Site Attacks

Cross-Site (XS) attacks are a family of web attacks where attackers lure users into visiting a malicious web page that tricks the user's web browser to send authenticated cross-site HTTP requests to a vulnerable target website. One of the first instances of cross-site attacks is Cross-Site Request Forgery (CSRF) [17, 21, 138, P1, 16, 20], where attackers leverage cross-site requests to perform security-sensitive, server-side state-changing operations, such as user credential reset (see, e.g., [139–141]) or money transfers [19], without user's consent or awareness. CSRF vulnerabilities are caused by the confused deputy problem [39, 38] where the server-side program cannot distinguish between unintentional and intentional requests.

Malicious cross-site requests can also target the users making these requests by leaking sensitive information about user's login status [142, 144, 143], account type [S2], age range [145], or user's identity (i.e., deanonymization attack) [102, S2], bypassing the Same-Origin Policy. These attacks are often called cross-site information leakage (XS-Leaks) or Cross-Origin State Inference (COSI) attacks [147, 35, 137, 148, 102, S2, 145, 149, 146]. COSI attacks abuse the fundamental concept of composability on the Web, which allows webpages to interact with one another, and exploit browser side channels to leak HTTP responses through this interaction.

Figure 2.1: Example of a COSI and CSRF attack.



Threat Model. CSRF and COSI attacks have a similar two-phases attack pattern: *preparation* and *attack*. Figure 2.1 exemplifies the threat model of these attacks. In the preparation step, the attacker prepares a malicious webpage referring to resources from the target site. These can be, for example, a hidden, self-submitting HTML form to reset the user password at the target site or a JavaScript file hosted by the target site. During the attack phase, the user visits the attack page (step 1). As a result of the included resource, the user's browser sends a cross-site request to the target website, and the browser automatically attaches the user's authenticated session cookies to this request (step 2). The target website receives and processes the request (step 3). In the case of CSRF, the server will perform the requested operation, e.g., by resetting the user password with an attacker-controlled one. In case of a COSI attack, the attack page uses browser side-channels to leak sensitive information about the user. For example, consider a cross-origin HTTP request that returns a 200 response code when the user is logged in and a 404 otherwise.

2.2.2 Client-Side CSRF

Client-side CSRF is a new category of CSRF vulnerability where the adversary can trick the client-side JavaScript program to send a forged HTTP request to a vulnerable target site by manipulating the program's input parameters. In a client-side CSRF attack, the attacker lures a victim into clicking a malicious URL that belongs to an attacker-controlled web page or an honest but vulnerable web site, which in turn causes a security-relevant state change of the target site. These vulnerabilities originate when the JavaScript program uses attacker-controlled inputs, such as the URL, for the generation of outgoing HTTP requests.

Listing 2.1: Example client-side CSRF vulnerability derived from SuiteCRM.

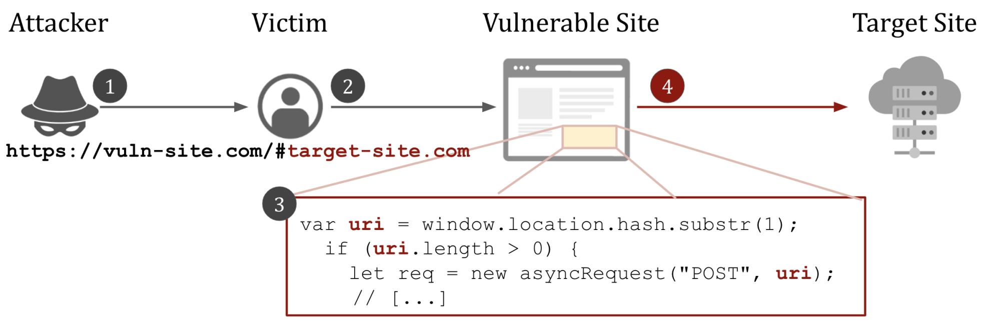
```
1   var i = document.querySelector('input');
2   async function h(e){
3    var uri = window.location.hash.substr(1);
4    if (uri.length > 0) {
5      let req = new asyncRequest("POST", uri);
6      // Add Synchronizer Token
7      req.initHeader('X-CSRF-TOKEN', token);
8      var price = await req.send();
9      i.value = price;}}
10   i.addEventListener('loadInvoice', h);
...
14   function showInvoicePrice(input_id) {
15    document.getElementById(input_id).dispatchEvent(new CustomEvent('loadInvoice', {}));}
16   showInvoicePrice('input');
```

Similarly to the classical CSRF (Cf. §2.2.1), client-side CSRF can be exploited to perform security-sensitive actions on the server-side and compromise the database integrity. Successful CSRF attacks can lead to remote code execution [150, 16], illicit money transfers [16, 19], or impersonation and identity riding [139, 21, 1, 140, 152, 151], to name only a few instances.

Vulnerability. Listing 2.1 exemplifies a vulnerable script–based on a real vulnerability that we discovered in SuiteCRM–that fetches a shopping invoice with an HTTP request during the page load. First, the program fetches an HTML input field with id `input` (line 1), and then defines an event handler `h` that is responsible for retrieving the price of the invoice with an asynchronous request and populating the `input` with the price (lines 2-9). For asynchronous requests, the function `h` uses YUI library [153], that provides a wrapper `asyncRequest` for the low-level `XMLHttpRequest` browser API. Then, the function `h` is registered as a handler for a custom event called `loadInvoice`. This event is dispatched by the function `showInvoicePrice` (lines 14-16). The vulnerability occurs (in lines 3-5) when the JavaScript program uses URL fragments to store the server-side endpoint for the HTTP request, an input that can be modified by the attacker.

Attack. Figure 2.2 shows an example of attack exploiting the client-side CSRF vulnerabilities of Listing 2.1. First, the attacker prepares a URL of the vulnerable site, by inserting the URL of the target site as URL fragment (step 1). Then, the victim is lured into visiting the vulnerable URL (step 2), as it belongs to an application that the user trusts. Upon completion of the page load (step 3), the JavaScript code will extract a URL from the URL fragment, and send an asynchronous HTTP request towards the target site, which in turn causes a security-relevant state change on the target server.

Figure 2.2: Example of client-side CSRF attack.



```
var uri = window.location.hash.substr(1);
  if (uri.length > 0) {
    let req = new asyncRequest("POST", uri);
    // [...]
```

Threat Model. The overall goal of an attacker is forging client-side HTTP requests by

Listing 2.2: Example request hijacking vulnerability in Microsoft Azure.

```
var params = (new URL(window.location)).searchParams;
var t = params.get("request");
if(t != null && t.length){
    // post message to opener
    opener && opener.postMessage("reauthPopupOpened", t);
    // listen for signal
    window.onmessage = function(){
        if (event.origin !== opener.origin) return;
        if (event.data === "sendRequest"){
            // top-level navigation request
            document.location.assign(t);}
    }}
```

manipulating various JavaScript input sources (e.g., see [32]). In this work, we consider the URL, window name, document referrer, postMessages, web storage, HTML attributes, and cookies, each requiring different attacker capabilities. Manipulating the URL, window name, referrer and postMessages require an attacker able to forge a URL or control a malicious web page. For example, a *web attacker* can craft a malicious URL, belonging to the origin of the honest but vulnerable web site, that when visited by a victim leads to automatic submission of an HTTP request by the JavaScript program of the target site. Alternatively, a web attacker can control a malicious page and use browser APIs to trick the vulnerable JavaScript of the target page to send HTTP requests. For example, a web attacker can use `window.open()` [154] to open the target URL in a new window, send postMessages [37] to the opened window, or set the window name through `window.name` API [155]. Furthermore, a web attacker can manipulate `document.referrer` leveraging the URL of the attacker-controlled web page.

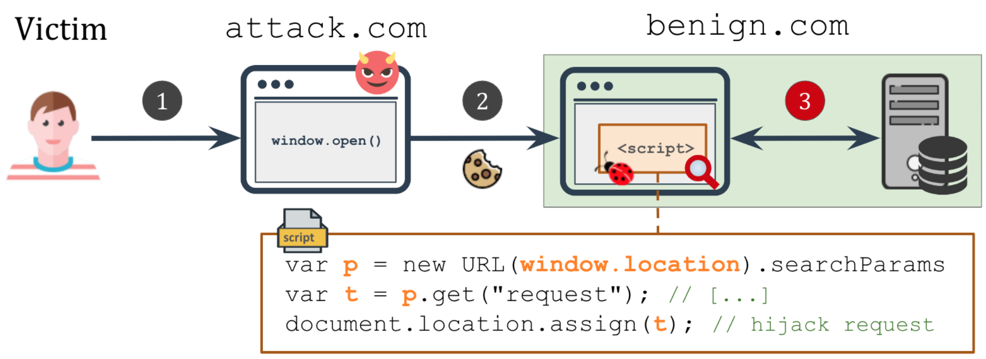
For web storage and HTML attributes, the attacker needs to add ad-hoc data items in the web storage or DOM tree. A web attacker could achieve that assuming the web application offers such functionalities (e.g., by HTTP requests). Similarly, a web attacker with a knowledge of an XSS exploit can manipulate the web storage or DOM tree. Finally, modifying cookies may require a powerful attacker such as a *network attacker*. This attacker can implant a persistent client-side CSRF payload in the victim's browser by modifying cookies (e.g., see [157, 54, 156]), which can lie dormant, and exploited later on to attack a victim. We observe that all attacks performed by the web attacker can be performed by a network attacker too.

Existing Defenses are Ineffective. Over the past years, the community proposed several defenses against CSRF (e.g., [17, 138, 66, 158, 159, 72]). Recently, browser vendors proposed to introduce a stricter same-site cookies policy [40, 41, 29], by marking all cookies as `SameSite=Lax` by default [30]. Unfortunately, existing mechanisms cannot offer a complete protection against client-side CSRF attacks, e.g., when synchronizer tokens [17, 138] or custom HTTP headers [17, 20] are used, the JavaScript program will include them in the outgoing requests as shown in line 7 of Listing 2.1. Also, if the browser or the web site is using the same-site policy for cookies, JavaScript web pages, once loaded, can perform preliminar same-site requests to determine whether a pre-established user session exists, circumventing the same-site policy.

2.2.3 Client-Side Request Hijacking

Client-side request hijacking vulnerabilities arise when attackers can trick the client-side JavaScript program into manipulating request-sending APIs with attacker-controlled inputs. The recently proposed client-side CSRF vulnerability [1, P1, 69] is a prominent example of such request hijacking, where attackers manipulate `XMLHttpRequest` [22] or `fetch` [23] API parameters, and trigger sensitive actions without user awareness and intention. However, other types of client-side request hijacking also exist.

Figure 2.3: Example request hijacking attack.



```
var p = new URL(window.location).searchParams
var t = p.get("request"); // [...]
document.location.assign(t); // hijack request
```

Listing 2.2 shows a real example of a request hijacking vulnerability that we discovered in Microsoft Azure (disclosed and patched), where attackers can hijack a top-level HTTP request. In more detail, the code first retrieves a query parameter value from the URL (lines 1-2), and checks that it is not empty (line 3). Then, it sends a postMessage to its `opener` webpage, and waits to receive back the `sendRequest` signal (lines 5-9). Finally, it triggers an HTTP request for navigation by changing the document location to the query parameter value (line 11). The vulnerability originates in the assignment in line 11 because attackers can control the value of query parameters and, ultimately, pick the URL of their choosing for the navigation request. Here, the distinctive characteristic of `location.assign()` as a top-level request introduces additional security risks for cross-site requests, because unlike `XMLHttpRequests` that are constrained by SameSite cookies [P4] and Same-Origin Policy [160], top-level requests including `location.assign()` are not, bypassing existing countermeasures.

Threat Model. In this thesis, we consider a *web* attacker [161, 17] who abuses inputs such as URL parameters, window name, document referrer, and postMessages, which is in line with prior work in the area of client-side vulnerabilities and defenses [17, P1, P4, 8, 32, 3]. Figure 2.3 shows an example attack scenario exploiting the vulnerability in Listing 2.2. First, the attacker prepares a malicious page and lures the victim into visiting it (step 1). The attack page uses the `window.open()` API [154] to open the vulnerable webpage in a new window (step 2), where it injects an attack payload in the query parameter `request` (say attack model A). Alternatively, the attacker can share the malicious URL with victims (instead of using browser APIs) and entice them to click on it, triggering a top-level navigation as shown in [P1] (say attack model B). When the page is loaded completely (step 3), the JavaScript code extracts the payload from the query parameter, and triggers a top-level HTTP request towards the payload value, enabling attackers to hijack the original request. Unfortunately, because this request is top-level, browsers will attach cookies to it, circumventing the SameSite policy [28, P4]. In particular, in attack model A, the `SameSite=Lax` policy (default in Chromium-based browsers) attaches cookies to `window.open()` requests but `SameSite=Strict` policy can mitigate that. However, in scenario B, even `SameSite=Strict` is not sufficient, as cookies are always attached to same-site requests. Consequently, the attacker obtains CSRF by sending arbitrary requests to any security-sensitive endpoint, resulting in compromise of database integrity (e.g., deleting VMs, and changing user settings in Azure). Note that cross-origin policies like CORS (i.e., `access-control-*` headers [162]) allow a server to restrict access of any origin other than its own, thus are ineffective against CSRF exploitations that abuse the same-origin requests.

Then, in this thesis, we consider other types of URLs that are not based on the `http` scheme. For example, the `location.assign()` API also accepts URLs with the `javascript` scheme, which enables attackers to escalate request API hijacking to arbitrary client-side code execution

Listing 2.3: Example of DOM Clobbering vulnerability where named properties overshadow JavaScript variables.

```
1  var s = document.createElement('script');
2  let config =  window.globalConfig || {href: 'script.js'};
3  s.src = config.href;
4  document.body.appendChild(s);
```

Listing 2.4: Example of DOM Clobbering vulnerability where named properties overshadow native DOM APIs.

```
1  var s = document.createElement('script');
2  let b = document.documentElement.getAttribute('baseURI');
3  s.src = b + '/script.js';
4  document.body.appendChild(s);
```

if there is no or improper input validation, e.g., by injecting `javascript:alert(document-.cookie)` in the query parameter `request` in Listing 2.2 Accordingly, as this example highlights, hijacking a request API can have a wide range of consequences, including cross-site request forgery, client-side code execution, open redirection, and sensitive information leakage–to name only a few examples. As we will show in §4.4, these types of request hijacking attacks could be mitigated by constraining request APIs with *opt-in* security policies, e.g., using the CSP `connect-src` directive [163].

2.2.4 DOM Clobbering

DOM Clobbering vulnerabilities originate from a naming collision between JavaScript variables and named HTML markups, i.e., markups with an `id` or `name` attribute [164, 25, 26]. When an undefined variable [166, 165] and an HTML markup have the same name, the browser replaces the pre-existing content of the variable with the DOM object mirroring the markup type. Listing 2.3 shows a snippet of vulnerable code, which loads a script whose URL is stored in a global configuration object, i.e., `window.globalConfig`. In more details, the code first creates a `script` tag (line 1), and then, it retrieves the global configuration object and stores it in a local variable `config` (line 2). If the configuration object does not exist, it uses a minimal default configuration, i.e., `{href: 'script.js'}` (line 2). Then, the program sets the `src` attribute of the newly created script tag to the `href` property of the configuration object (line 3) and appends the new script to the DOM tree (line 4).

The vulnerability originates in the assignment in line 2 because attackers can control the value of `window.globalConfig`, and ultimately, pick the script `src` value of their choosing by injecting an HTML tag with `id="globalConfig"`, e.g., `<a id="globalConfig" href="malicious.js">`. When parsing such a markup code, the browser maps the anchor tag element to the `window.globalConfig` property as mandated by the *named property access* rule of the HTML specifications (see [169, 168, 167]). The escalation to arbitrary code execution happens in line 3, when the code reads the `href` property of the object `window.globalConfig`, which no longer contains the object with the global configuration but it contains the attacker-controlled anchor tag whose `href` property value is `malicious.js`.

Attackers can abuse named property accesses in other ways, where instead of overwriting variables by HTML nodes, they can overshadow browser APIs. Listing 2.4 illustrates an example of such an attack. Similarly to Listing 2.3, this code also dynamically creates and loads a script. Instead of fetching the URL from a global configuration object, the code intends to use the `baseURI` attribute of the main HTML tag via the `document.documentElement` API (line 2). An attacker can manipulate the content of `src` in line 3 by shadowing the native property `document.documentElement` using an attacker-injected node in the DOM tree [170], e.g., a `form` element with `name="documentElement"` and the custom property `baseURI="malicious.js"`. When parsing the form tag, the browser maps the property

Table 2.1: Contexts where the three `SameSite` policies apply. We use $^{(*)}$ for the Lax+POST exceptional policy, and ✓ to show contexts where cookies are included in the cross-site HTTP request.

None	Lax	Strict	Context		Example
✓	✓	-	Anchor	GET	`<a href=`u`>`
✓	✓	-	Form	GET	`<form method=GET action=`u`>`
✓	✓	-	Link prerender	GET	`<link rel=prerender href=`u`/>`
✓	✓	-	Link prefetch	GET	`<link rel=prefetch href=`u`/>`
✓	✓	-	win.open()	GET	`window.open(u)`
✓	✓	-	win.location	GET	`window.location.href=`u
✓	✓$^{(*)}$	-	Form	POST	`<form method=POST action=`u`>`
✓	-	-	Iframe	GET	`<iframe src=`u`>`
✓	-	-	Object	GET	`<object data=`u`>`
✓	-	-	Embed	GET	`<embed src=`u`>`
✓	-	-	Image	GET	`<img src=`u`>`
✓	-	-	Script	GET	`<script src=`u`>`
✓	-	-	StyleSheet	GET	`<link rel=stylesheet href=`u`>`
✓	✓$^{(*)}$	-	Async Reqs.	Any	`xmlhttp.open("POST", `u`)`

`document.documentElement` to the JavaScript object representing the form tag (an instance of the `HTMLFormElement` class) which has a function called `getAttribute` which returns the value of the attribute `baseURI`, i.e., the string `malicious.js`.

Threat Model. In a DOM Clobbering attack, the attacker needs to insert an ad-hoc HTML payload into a target, vulnerable webpage. A *web* attacker [161, 17] can achieve that, e.g., adding a preview of a post to the client-side webpage by leveraging the URL parameters. Another example is the case where the attacker can implant a persistent DOM Clobbering payload in the target webpage, which can lie dormant, and exploited later on to attack a victim, e.g., adding persistent comments in the UI through Gmail's dynamic email feature [171] which allows including HTML content [88], or user-generated Markdown descriptions in code repositories that are turned into HTML content [172, 173]. Finally, a more powerful web attacker (e.g., [8, 54]) who is aware of a markup injection vulnerability can manipulate the DOM tree.

2.3 Same-Site Policies

A distinctive feature of XS attacks (Cf. §2.2.1) is that browsers include existing valid cookies in all outgoing requests, regardless of the context of the requests (e.g., user click on anchor tags or asynchronous HTTP fetch operation) or the origin of the page performing the requests (e.g., same-site or cross-site requests). An effective solution to XS attacks is limiting the scope of cookies by defining the context in which browsers can include cookies.

Limiting the scope of cookies to prevent XS attacks is not a new idea. For example, in 2006, Johns et al. proposed Request Rodeo [66], an HTTP proxy for browsers that can detect and remove cookies from cross-site requests that the user did not initiate when interacting with the webpage. More recently, in 2016, Google revamped a similar idea by introducing in Chrome a new cookie attribute [174], the `SameSite` attribute for the `Set-Cookie` HTTP response header (see RFC 6265bis [28]), allowing developers to pick one out of three pre-defined cookie policies, namely, the None, Lax, and Strict policies [28, 175].

SameSite Cookies. The `SameSite` attribute introduces three pre-defined cookie policies. The None policy specifies that cookies are attached to *all* outgoing requests, including cross-site ones. This policy corresponds to the default policy before the introduction of the SameSite attribute. At the other end of the spectrum, we have the Strict policy that stipulates that cookies are restricted to the same-site only, i.e., cookies are never attached to any cross-site request. Finally, the Lax policy is the new *default* policy for cookies, and it defines the contexts where browsers can include cookies for cross-site requests. For example, browsers include cookies to same-site requests and top-level navigation requests (e.g., clicking on an anchor link) with safe HTTP methods [176]. However, browsers will not include cookies to cross-site requests with unsafe HTTP methods. Table 2.1 summarizes the context where the same-site policies apply.

New Default Policy and Exception. Starting from July 2020, Google Chrome set the new default policy to Lax [95], meaning that when the `SameSite` attribute is missing, the browser will enforce the Lax policy. Other browser vendors adopted [178, 177] or are planning to adopt [41] the same new default policy.

Unfortunately, enforcing a default Lax policy may break web application functionalities that rely on cross-site communications and cookies. For example, web-based Single Sign-On (SSO) implementations, such as OpenIDConnect or SAML SSO implementations, optimize user experience by avoiding user re-authentication when valid authenticated session cookies are included in the cross-site requests. Such requests are often implemented via POST asynchronous requests or HTML POST forms. The new default Lax policy forbids browsers from sending cookies with these requests, breaking these functionalities. To counter that, Chrome introduced an exception to the Lax policy—called `Lax+POST`—where, for all cookies without the `SameSite` attribute, Chrome applies the None policy only for the first two minutes after the cookie is set. Then, Chrome switches to the Lax policy. Table 2.1 lists the contexts covered by SameSite cookies.

3 Automatic Detection of Client-side CSRF Vulnerabilities

In this chapter, we present an automatic technique to detect and study client-side CSRF vulnerabilities, addressing RQ1 of §1.1. Client-side CSRF is a new variant of CSRF vulnerabilities that enables attackers to generate arbitrary requests by modifying the input parameters of JavaScript programs (Cf. §2.2.2).

In general, studying client-side CSRF vulnerabilities in JavaScript-based web applications is not an easy task. First, there is no canonical representation for JavaScript code. Second, JavaScript programs are event-driven, and we need models that capture and incorporate this aspect into the canonical representation. Third, pure static analysis is typically not sufficiently accurate due to the dynamic nature of JavaScript programs [11–13], and their execution environment [14], calling for hybrid static-dynamic analysis techniques. Finally, JavaScript libraries constitute a noteworthy fraction of code across web pages, and analyzing them repeatedly leads to inefficient models poorly suitable for detecting vulnerabilities.

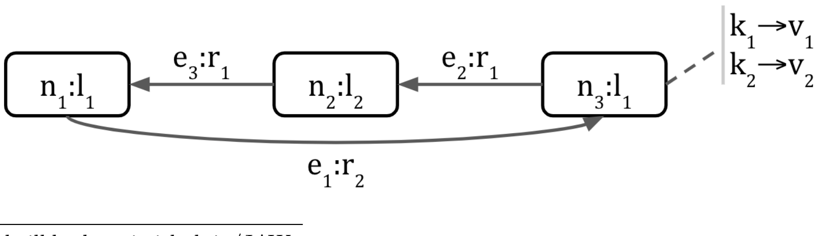
In this chapter, we address these challenges by proposing *hybrid property graphs* (HPGs), a coherent, graph-based representation for client-side JavaScript programs, capturing both static and dynamic program behaviors. Inspired by prior work [15], we use property graphs for the model representation and declarative graph traversals to identify security-sensitive HTTP requests that consume data values from attacker-controllable sources (§3.1). Also, we present JAW, a framework that detects client-side CSRF by automatically collecting web resources, monitoring program execution and instantiating a HPG for each web page, starting from a single seed URL (§3.2). We release the source code of JAW[1] to support the future research effort to study vulnerable behaviors of JavaScript programs.

Finally, we instantiated JAW against *all* (i.e., 106) web applications of the Bitnami catalog [179] to detect and study client-side CSRF, covering, in total, over 228M lines of JavaScript code over 4,836 web pages (§3.3). Overall, our approach uncovers 12,701 forgeable client-side requests affecting 87 web applications. For 203 forgeable requests, we successfully created client-side CSRF exploits against seven web applications that can execute arbitrary server-side state-changing operations or enable cross-site scripting and SQL injection, that are not reachable via the classical attack vectors. We analyzed the forgeable requests and identified 25 distinct patterns, highlighting the fields that can be manipulated and the type of manipulation.

3.1 Hybrid Property Graph

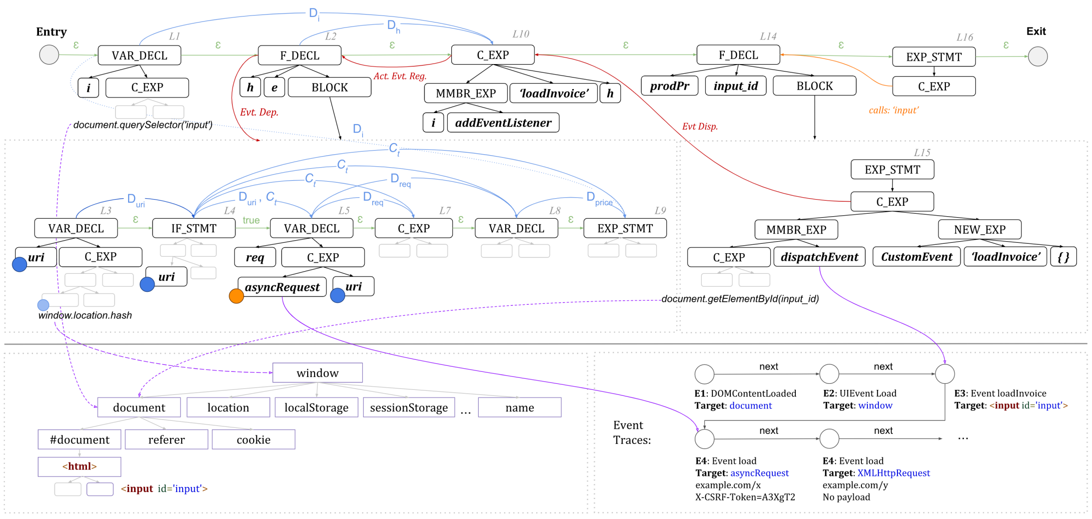
This section introduces hybrid property graphs (HPGs). A HPG comprises of the *code representation* and *state values*. The code representation unifies multiple representations of a JavaScript program whereas the state values are a collection of concrete values observed during the execution of the program. We use a *labeled property graph* to model both, in which nodes and edges can have labels and a set of key-value properties. The example below shows a graph where l_i is the node label and r_j is the relationship label. Nodes and edges can store data by using properties, a key-value map.

Figure 3.2: Example of labeled property graph



[1]https://soheilkhodayari.github.io/JAW

Figure 3.1: HPG for the running example in Listing 2.1. The top part depicts the code representation, including the AST (black edges), CFG (green edges), IPCG (orange edges), PDG (blue edges), ERDDG (red edges), and the semantic types (blue and orange filled circles representing `WIN.LOC` and `REQ` types, respectively). Note that not all nodes and edges are shown for brevity. Edges connected to dotted boxes reflect that the edge is connected to each node within the box. The bottom part demonstrates the dynamic state values to augment the static model. Arrows between the two parts represent the link between the two models.



In the rest of this section, we present how we map the code representation and state values into a graph (Sections 3.1.1 and 3.1.2), and show how we can instantiate and query such a graph to study client-side CSRF vulnerabilities (§3.2.3).

3.1.1 Code Representation

The code representation models the JavaScript source code and builds on the concept of code property graph (CPG) which combines three representations for C programs, i.e., abstract syntax tree, control flow graph, and program dependence graph [15]. Later, the same idea has been adapted to study PHP programs [51], extending CPGs with call graphs. HPGs further extend CPGs with the event registration, dispatch, and dependency graph and the concept of semantic types.

Abstract Syntax Tree (AST). An AST is an ordered tree encoding the hierarchical decomposition of a program to its syntactical constructs. In an AST, terminal nodes represent operands (e.g., identifiers), and non-terminal nodes correspond to operators (e.g., assignments). In Figure 3.1, AST nodes are represented with rounded boxes. Terminal nodes are in bold-italic, whereas non-terminal nodes are all capitals. AST edges connect AST nodes to each other following the production rules of the grammar of the language, e.g., in line 10 of Listing 2.1, `i.addEventListener('loadInvoice', h)` is a call expression (CALL_EXP) with three children, the member expression (MMBR_EXP) `i.addEventListener`, the literal `'load-Invoice'` and an identifier `h`. AST nodes are core nodes of the code representation, providing the building blocks for the rest of the presented models.

Control Flow Graph (CFG). A CFG describes the order in which program instructions are executed and the conditions required to transfer the flow of control to a particular path of execution. In Figure 3.1, CFG is modeled with edges (in green) between non-terminal AST nodes. There are two types of CFG edges: conditional (from predicates and labeled with *true* or *false*) and unconditional (labeled with ϵ). A CFG of a function starts with a *entry* node and ends with a *exit* node, marking the boundaries of the function scope. These fragmented intra-procedural flows are connected to each other by inter-procedural *call* edges, as discussed next.

Inter-Procedural Call Graph (IPCG). An IPCG allows inter-procedural static analysis of JavaScript programs. It associates with each call site in a program the set of functions that may be invoked from that site. For example, the expression `showInvoicePrice('input')` of line 16 in Listing 2.1 calls for the execution of the function `showInvoicePrice` of line 14. We integrate the IPCG in our code representation with directed *call* edges, e.g., see the orange edge between the C_EXP AST node and the F_DECL AST node in Figure 3.1.

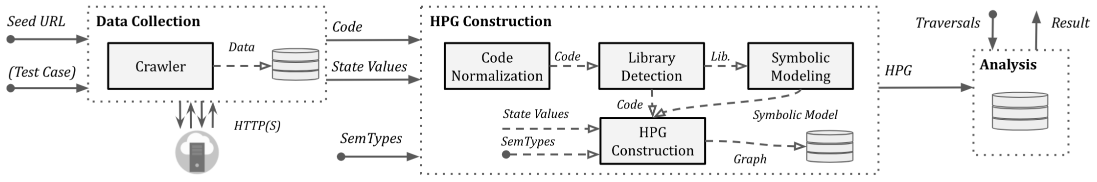
Program Dependence Graph (PDG). The value of a variable depends on a series of statements and predicates, and a PDG [180] models these dependencies. The nodes of a PDG are non-terminal AST nodes, and edges denote a *data*, or *control* dependency. A data dependency edge specifies that a variable, say x, *defined* at the source node is afterwards *used* at the destination node, labeled with D_x. For example, in Figure 3.1, variable `uri` is declared in line 3 (by VAR_DECL), and used in line 4 (in IF_STMT), and thus a PDG edge (in blue) connects them together. A control dependency edge reflects that the execution of the destination statement depends on a predicate, and is labeled by C_t, or C_f corresponding to the true, or false condition, e.g., the execution of the CALL_EXP in line 7 depends on the IF_STMT predicate in line 4.

Event Registration, Dispatch and Dependency Graph (ERDDG). The ERDDG intends to model the event-driven execution paradigm of JavaScript programs and the subtle dependencies between event handlers. In an ERDDG, nodes are non-terminal AST nodes, and we model execution and dependencies with three types of edges. The first edge models the registration of an event, e.g., line 10 in Listing 2.1 registers `h` as the handler for the custom event `loadInvoice`. We represent the registration of an event with an edge of type *registration* between the node C_EXP (i.e., the call site for `addEventListener`) and the node F_DECL (i.e., the statement where the function `h` is defined). The second edge models the dispatch of events. For example, line 15 in Listing 2.1 calls the browser API `dispatchEvent` to schedule the execution of the handler of the `loadInvoice` event type. We model the transfer of control with an edge of type *dispatch*. See, for example, the edge (in red) between the C_EXP node of line 15 and the C_EXP registering the handler in Figure 3.1. The last edge models dependencies between statements and events. We implement the dependency with an edge between the AST node for the handler's declaration and the AST nodes of the handler's statements. Figure 3.1 shows such an edge from the F_DECL node of line 2 and the body of the function.

Semantic Types. The detection of client-side CSRF requires identifying statements that send HTTP requests, and that consume data values from pre-defined sources. We model the properties of statements via semantic types. A semantic type is a pre-defined string assigned to program elements. Then, types are propagated throughout the code, following the calculation of a program, e.g., we can assign the type `WIN.LOC` to `window.location` and propagate it to other nodes, following PDG, CFG, IPCG, and ERDDG edges. In Figure 3.1, we use a blue filled circle for the type `WIN.LOC` that is propagated following the D_{uri} PDG edge, i.e., the term *uri* of line 3, 4, and 5. Semantic types can also be assigned to functions to specify their behavior abstractly. For example, we can use the string `REQ` for all browser APIs that allow JavaScript programs to send HTTP requests, such as `fetch`, or `XMLHttpRequest`. HPGs model semantic types as properties of the AST node.

Symbolic Modeling. When analyzing the source code of a program, we need to take into account the behaviors of third-party libraries. We extract a symbolic model from each library and use it as a proxy for the analysis of the application code. In this work, the symbolic model is an assignment of semantic types to libraries' functions and object properties. For example, in Figure 3.1, we can use the semantic type `REQ` (represented with an orange filled circle) for the `asyncRequest` term, and abstract away its actual code. Also, to reconstruct the data flow of

Figure 3.3: Architecture of JAW.



programs that use library functions, we define two semantic types modeling intra-procedural input-output dependencies of library functions. We use the semantic type $o \leftarrow i$ for functions whose input data values flow to the return value and the type $o \sim i$ for functions whose output is conditioned on the input value (e.g., by an IF_STMT). As we will show in §3.2, the symbolic modeling of libraries is performed automatically by JAW, who creates a mapping between the library elements and a list of semantic types.

3.1.2 State Values

JavaScript programs feature dynamic behaviors that are challenging to analyze via static analysis. As such, we augment HPGs to include concrete data values collected at run-time, and link them to the counterpart code representation.

Event Traces. To capture the possible set of fired events that are not modeled due to the limitations of the static analysis [12], or auto-triggered events, we augment the static model with dynamic traces of events. Event traces are a sequence of concrete incidents observed during the execution of a web page. For example, the `load` event or a network event for the response of a HTTP request. We use the trace of events fired upon the page load to activate additional *registration* edges in our ERDDG graph when possible. As shown in Figure 3.1, the nodes of the graph for event traces represent concrete events observed at run-time, and edges denote their ordering.

Environment Properties. Environment properties are attributes of the global `window` and `document` objects. The execution path of a JavaScript program and the values of variables may differ based on the values of the environment properties. We enrich HPGs by creating a graph of concrete values for the properties observed dynamically. We also store a snapshot of the HTML DOM tree [181]. If the value of a variable is obtained from a DOM API, the actual value can be resolved from the tree. We use the DOM tree to locate the objects that a DOM API is referencing. For example, to determine if an event dispatch is targeting a handler, we can check if the dispatch and registration is done on the same DOM object. We create a node for each environment property, and store concrete values as properties of the node. As depicted in Figure 3.1, we connect these nodes by edges representing a property ownership, or a parent-child relationship.

3.2 JAW: Javascript Analysis frameWork

In this section, we present JAW, a framework to study client-side CSRF vulnerabilities using HPGs. Starting from a seed URL of a web site, JAW visits web pages using a JavaScript-enabled web crawler to collect the web resources. During the visit, JAW also collects run-time state values. Then, given a list of user-defined semantic types and their mapping to JavaScript language tokens, JAW constructs the HPG. The construction has two phases. First, JAW identifies external JavaScript used by the program and processes it in isolation to extract a symbolic model. Then, it constructs the graph of the rest of the JavaScript code, and link elements of the

JavaScript program to the state values. Finally, JAW analyzes client-side CSRF by executing queries on the HPG (§3.2.3). Figure 3.3 shows an overview of the JAW's architecture.

3.2.1 Data Collection

The data collection module performs two tasks: crawling to discover URLs from different user states, and collecting the JavaScript code and state values for each web page found.

Input. The input of the data collection module is a seed URL of the web application under test, and, optionally, test cases to pass the user login, e.g., as scripted Selenium tasks [182] or via trace recording [183, 184].

Crawler. We developed a crawler that uses a headless instance of Chrome [185] controlled via Selenium [182]. Starting from the seed URL, the crawler visits the web application to collect web resources and run-time execution data. It follows the iterative deepening depth-first search strategy, and terminates when no other URLs are found, or when its allocated time budget runs out (default is 24h). Optionally, if provided as input, it executes test cases before the crawling session.

JavaScript Code and State Values. When visiting each page, the crawler stores the web resources and state values every $t_i = 10$ seconds for $m = 2$ times (configurable parameters). The crawler collects the HTML page, JavaScript program, fired events, HTTP requests and responses, and the JavaScript properties explicitly shown in (bottom left of) Figure 3.1 for each t_i interval. While JavaScript properties are extracted via the Selenium interface, we developed a Chrome extension for our crawler that resorts to function hooking to intercept calls to the `addEventListener` for collecting events and to the `chrome.webRequest` API to intercept the network traffic.

3.2.2 Graph Construction

JavaScript code and state values collected are next used to build a HPG. The built graph is imported into a Neo4j [186] database allowing for fine-grained, declarative path traversals to detect and study client-side CSRF. This section delineates technical details for constructing HPGs.

Code Normalization. As the first step, JAW creates a normalized JavaScript program by concatenating code segments inside the script tags and HTML attributes, preserving the execution order of program segments. When combining inline code, JAW replaces inline event handler registration with `addEventListener` API.

Library Detection. To identify libraries, we use Library Detector [187], a tool that searches for known library signatures inside the execution environment (e.g., global variables). We used it as a bundled script injected by Selenium [183]. Library Detector has a series of pre-defined checks (i.e., usage indicator functions) for each JavaScript library that it supports. It searches for known library signatures inside the execution environment by appling the usage indicator functions. For example, global variables set on the `Window` object by a library are an indicator of the usage of that library. It returns the list of libraries used in the web page. At the time of writing this paper, Library Detector provides support for the detection of 114 different library scripts, including JQuery, React, Angular, and Prototype.

HPG Construction. JAW constructs HPGs as follows. First, a graph is created for the symbolic modeling of each detected library. This step is skipped if a symbolic model for the library already exists. Then, it creates a graph for the program under analysis. Regardless the use of the graph, the rules to construct a HPG do not change, as presented next.

1. AST—JAW uses Esprima [188], a standard-compliant ECMAScript [126] parser to generate the AST of the normalized source code. The output of Esprima is a JSON representation of the AST. In this representation, a node is a key-value dictionary with a `type` property (e.g., VAR_DECL) and edges are represented with ad-hoc dictionary keys. We mapped the JSON output to AST nodes and AST edges of our graph.

2. CFG— We extensively reviewed open-source CFG generators, such as escontrol [189], styx [190], or ast-flow-graph [191], and selected Esgraph [192] because of its popularity, and compliance with Esprima. Starting from an AST, Esgraph generates a CFG where nodes are AST nodes for statements or declarations, and an edge is labeled with `true` or `false`, for a conditional branch, or ϵ for a node of the same basic block.

3. PDG—JAW uses dujs [193], a def-use analysis library based on Esgraph. We modified dujs to add support for global variables, closures, and anonymous function calls. The output of dujs is a list of def-use relationships for each variable v between the AST edges, that JAW import as data dependence edges D_v in our HPG. For the control dependence edges, JAW calculates post-dominator trees [194] from the CFG, one for each statement s. Then, JAW maps each edge of the tree to C_t or C_f for the true or false branch, respectively.

4. IPCG—JAW generates the IPCG as follows. During the construction of the AST and CFG, JAW keeps track of all function definitions and call sites. Then, it associates a call site to the function definition(s) it may invoke. There are five types of call expressions in JavaScript: function calls on the global object (e.g., `foo()`), property calls (e.g., `a.foo()`, or `a['foo']()`), constructor calls (e.g., `new Foo()`), invocations via the `call()` [195], and `apply()` method [196]. For all cases, the actual function definition name may be aliased. We resolve the pointers using our PDG, and connect the call edge accordingly. If the value of the pointer is conditioned, we connect an edge to each respective function definition.

5. ERDDG—For the generation of the ERDDG, JAW keeps track of event dispatches and handler registrations during the creation of the AST and the CFG. For each event handler found, JAW creates a registration edge that connects the top-level AST node (i.e., CFG node) to the handler function, and a dependency edge connecting the handler function to statements of the body. To associate each event dispatch to a registration site, we check if they target the same DOM element. For this, we resolve the pointer on which the event is dispatched, and the pointer on which the handler is registered leveraging our PDG, and check if they refer to the same variable declaration or different variables with verbatim or semantically same values. We use the DOM snapshot to check if two different DOM queries can semantically target the same element. For example, an element can be queried with its *id*, or alternatively its *name* attribute. Once we determine that the pointers reference the same element, we connect an edge between the dispatch and registration sites.

6. Semantic Types and Propagation— The input for this step is a mapping T between a semantic type t and a signature for AST node σ, e.g., we map the type `WIN.LOC` to the JavaScript property `window.location`. For each pair $(t,\sigma) \in T$, JAW stores each type t to the AST node that is matching the signature σ. Then, JAW propagates the type t through the HPG.

Algorithm 1 propagates forward a type t from a node n to other nodes. First, the function `propagateLeft` assigns the type t to the variable v on the left-hand side (e.g., of an assignment), if any, and returns it. Then, the function `propagateByPDG` propagates t following PDG edges and returns the visited paths P. Then, for each node n_t at the end of the path $p_i \in P$, we distinguish three cases. The first case is that n_t is a function call that is modeled by the special semantic types assigned during the symbolic modeling. If so, we taint the output variable o, and recursively call `propagateForward` for o. Second, n_t is a call expression having an IPCG edge. In this case, we taint the parameter c on the function definition corresponding to the argument tainted on the call site, and call `propagateForward` for c. Then, we check if the last tainted node from the context of the function definition is a tainted return statement. If so, we call `propagateForward` for the variable v_{left} on the call site that holds the returned result. Third, n_t is an event dispatch expression that passes tainted data. In this case, we jump the dispatch

Algorithm 1: Forward semantic type propagation

```
inputs  : Node $n$ with a variable having semantic type $t$.
outputs : Propagates semantic types and returns the last tainted node.
function propagateForward($n$, $t$):
    $v \leftarrow$ propagateLeft($n$, $t$) // taint left-hand side
    $n_t \leftarrow n$ // last tainted node
    $P \leftarrow$ propagateByPDG($n$, $v$, $t$) // tainted PDG paths
    for $p_i \in P$ do
        $n_t \leftarrow p_i[p_i.length - 1]$ // last CFG-level tainted node
        $v_t \leftarrow$ getRightHandSideTaintedVariable($n_t$, $t$)
        if hasSymbolicFunctionCall($n_t$) and hasSemanticType($n_t$, "o<-i") then
            $o \leftarrow$ propagateLeft($n_t$, $t$)
            propagateForward($o$, $t$) // recursion
        end
        if hasCallExpressionWithCallArgOfType($n_t$, $t$) then
            $c \leftarrow$ traverseCallEdge($n_t$, $v_t$, $t$) // call def param
            $ret \leftarrow$ propagateForward($c$, $t$) // returned variable
            if isRetStmt($ret$) and hasSemanticType($ret$, $t$) then
                $v_{left} \leftarrow$ propagateLeft($n_t$, $t$)
                if $v_{left}$ is not $null$ then
                    propagateForward($v_{left}$, $t$) // recursion
                end
            end
        end
        if hasDispatchEdgeWithArgOfType($n_t$, $t$) then
            $e \leftarrow$ traverseDispatchAndRegistrationEdges($n_t$, $v_t$, $t$) // handler param
            propagateForward($e$, $t$)
        end
    end
    return $n_t$ // last tainted node
```

and registration edges, taint the corresponding event variable, and call `propagateForward` for the variable. This process terminates when none of the above criteria holds.

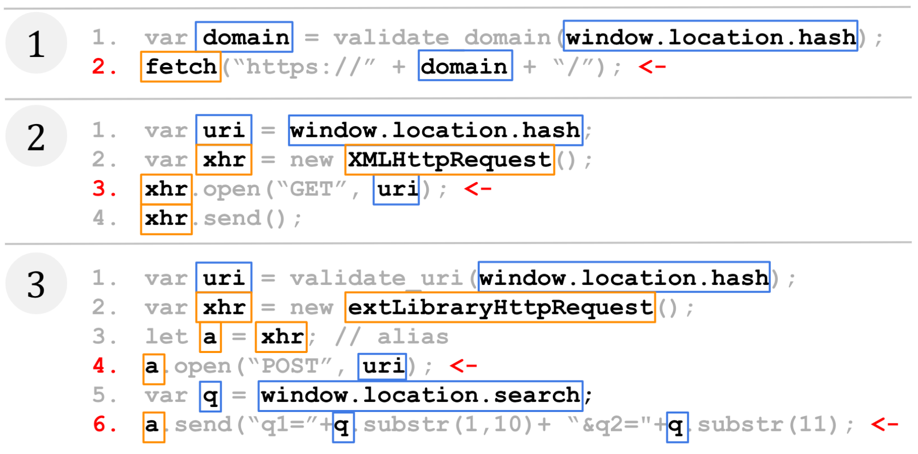
JAW performs the semantic type propagation when creating both the HPG for the symbolic modeling of a library and the HPG of the rest of the code. When creating the HPG for the rest of the code, the semantic type mapping T includes the mapping created during the symbolic modeling. Table 3.1 summarizes the list of semantic types supported by JAW. We can use one semantic type for each of the injection points where the attacker can input data. Semantic types can also be assigned to functions to specify their behavior abstractly, e.g., functions that delegate the dispatch of events or the HTTP requests to low-level browser APIs, as discussed in more detail next.

Symbolic Modeling. The output of this step is a mapping of semantic types and AST nodes, which is used during the construction of a HPG for the program under analysis. Symbolic modeling starts with the construction of a HPG from the library source code. Then, after the propagation of the semantic types, JAW searches for function definitions with intra-procedural input-output relationships. More specifically, JAW identifies all non-anonymous function expressions with at least one input parameter, and track the value of its return statement(s), if any, through a backward program slicing approach. At a high level, we start from where a value is returned, flow through where it is modified, and end at where it is generated leveraging the PDG, CFG, IPCG, and ERDDG graphs. If the returned variable, say o, has a PDG *control* dependency to a function input, say i, we assign the type $o \sim i$ to the function. If we establish a PDG *data* dependency, we mark it with $o \leftarrow i$. Finally, JAW selects all function expression and object property nodes with at least one semantic type, that will be used in the HPG construction of the JavaScript code.

Table 3.1: List of semantic types supported by JAW. Types are assigned to constructs representing input sources of a web application, functions that send HTTP requests, dispatch or register events, and functions with inputs/outputs.

Descr.	Type	Example of use
Window URL	WIN.LOC	window.location.hash
Cookie	DOM.COOKIES	doc.cookie
localStorage	LOCAL-STORAGE	doc.localStorage
sessionStorage	SESSION-STORAGE	doc.sessionStorage
postMessage	POST-MESSAGE	addEventListener(evt, h)
Window Name	WIN.NAME	window.name
Document Referrer	DOC.REFERRER	doc.referrer
DOM Attribute	DOM.READ	doc.getElementById('x').value
Client-Side Request	REQ	XMLHttpRequest
Event Dispatch	E-DISPATCH	el.triggerHandler(evt)
Handler Registration	E-REGISTER	el.on(evt, h)
Func. I/O	$o \leftarrow i$	function(i){return $o = g(i)$;}
Func. I/O	$o \sim i$	function(i){if($cond(i)$) return o;}

Figure 3.4: Examples of vulnerable code. Orange and blue boxes represent `REQ` and `WIN.LOC` semantic types, respectively.



1

```
var domain = validate_domain(window.location.hash);
fetch("https://" + domain + "/"); <-
```

2

```
var uri = window.location.hash;
var xhr = new XMLHttpRequest();
xhr.open("GET", uri); <-
xhr.send();
```

3

```
var uri = validate_uri(window.location.hash);
var xhr = new extLibraryHttpRequest();
let a = xhr; // alias
a.open("POST", uri); <-
var q = window.location.search;
a.send("q1="+q.substr(1,10)+ "&q2="+q.substr(11); <-
```

3.2.3 Analysis of Client-side CSRF with HPGs

Given a HPG as described in Sections 3.1.1 and 3.1.2, we now use it to detect and study client-side CSRF. We say that a JavaScript program is vulnerable to client-side CSRF when (i) there is a data flow from an attacker-controlled input to a parameter of an outgoing HTTP request *req*, and (ii) *req* is submitted on the page load.

We model both conditions using graph traversals, i.e., queries to retrieve information from HPGs. In our work, we define graph traversals using the declarative Cypher query language [197], but in this paper we exemplify Cypher syntax with set notation and predicate logic while retaining the declarative approach. A query Q contains all nodes n of HPG for which a predicate p (i.e., a graph pattern) is true, i.e., $Q = \{n : p(n)\}$. We use predicates to define a property of a node. For example, we use the predicate $hasChild(n,\ c)$ to say that a node n has an AST child c. Another example of predicate is $hasSemType(n,\ t)$, which denotes a node n with a semantic type t. Predicates can be combined to define more complex queries, e.g., via logical operators.

Detection of Client-side CSRF. The first condition for client-side CSRF vulnerability is the presence of attacker-controlled input parameters for outgoing requests. Figure 3.4 shows different instances of vulnerable code taken from real examples, where by construction, we assigned the `WIN.LOC` and `REQ` semantic types to AST nodes, which are shown as blue and orange boxes, respectively. For all three cases of Figure 3.4, the goal is to identify the lines of code having both orange and blue labels (marked with a red arrow). At a high level, a line of code is a non-terminal AST node for JavaScript statements or declarations (e.g., EXP_STMT, VAR_DECL), that we represent with the predicate $isDeclOrStmt(n)$. Then, once we identify such an AST node n, we need to explore whether the node has two children c_1 and c_2 where one is of type `REQ` and the other is of type `WIN.LOC`. Following our notation for queries, we can write:

$$\begin{aligned} N_1 = \{ & n : isDeclOrStmt(n) \wedge \exists c_1, c_2,\ c_1 \neq c_2 \wedge \\ & hasChild(n,\ c_1) \wedge hasSemType(c_1,\ \text{“}REQ\text{”}), \wedge \\ & hasChild(n,\ c_2) \wedge hasSemType(c_2,\ \text{“}WIN.LOC\text{”})\} \end{aligned} \tag{3.1}$$

Query 3.1 is not a sufficient condition to determine the presence of a client-side CSRF vulnerability, as the returned nodes may correspond to lines of code not executed at page load. We refine it with additional checks for reachability. In general, starting from a node n such that $isDeclOrStmt(n)$, we could follow backward CFG edges (both ϵ, $true$, and $false$) to determine whether we reach the CFG entry node. Then, whenever we reach a function definition (e.g., F_DECL), we jump to all its call sites following the IPCG call edges. But this will not be sufficient because a function can be executed when a specific event is fired. Accordingly, we need to visit backward the ERDDG edges i.e., the dependency edge, followed by the registration and the dispatch edge. We handle separately special cases where events are fired by the browsers automatically during loading a page. We keep on following backward CFG, ERDDG, and IPCG edges until either we reach the CFG entry node or when there are no longer nodes matching any of the previous criteria. We say that a node n is reachable if the CFG entry node is in the query result set.

Analysis of Vulnerable Behaviors. The previous queries can identify the general vulnerable behavior of client-side CSRF, i.e., a program that submits a HTTP request using attacker-chosen data values. However, programs may implement a variety of checks on the inputs, which can eventually influence the exploitation landscape. In Figure 3.4, for example, Program 1 shows a vulnerable script whose domain validation of line 1 restrains the attacker from manipulating the entire request URL. Program 2, however, shows a case where the attacker can chose the complete URL string, including the path and query string. One aspect of client-side CSRF vulnerabilities that we intend to study is to identify the extent to which an attacker can manipulate the outgoing request. For instance, if `window.location` properties flow to a request parameter without any sanitization. Query 3.2 captures this aspect:

$$\begin{aligned} N_2 = \{ & n_1 : \forall n_1 \in N_1,\ \exists n_2,\ hasPDGPath(n_2,\ n_1) \wedge \\ & isAssignment(n_2) \wedge \exists c,\ hasChildOnRight(n_2,\ c) \wedge \\ & isMemberExp(c) \wedge hasValue(c, \text{“}window.location\text{”})\} \end{aligned} \tag{3.2}$$

Query 3.2 checks if the node n_1 returned by Query 3.1 is connected via PDG edges to an assignment statement whose right-hand side child is a property of the `window.location`. The predicate $hasPDGPath(n_2,\ n_1)$ specifies that there is a path from n_2 to n_1 following PDG edges, and $isAssignment(n_2)$ marks that n_2 is a VAR_DECL, or an ASSIGN_EXP node.

Another aspect to consider is the number of attacker-controllable items within a request. For example, Program 3 of Figure 3.4 shows a more complex example where the attacker can also control the content of the request body, increasing the flexibility to create an exploit for the vulnerable behavior. For this, a query can cluster vulnerable lines of code that belong to the same HTTP request, making use of the PDG dependencies among elements of the same request. Then, the query can count the number of attacker-controllable injection points (see, e.g., the two injection points in line 6 of Program 3 as well as the injection point in line 4).

3.3 Evaluation

The overarching goal of our evaluation is to study client-side CSRF vulnerabilities and to assess the efficacy and practicality of JAW. We run JAW on 4,836 web pages, crawled from 106 popular web applications, generating HPGs for 228,763,028 LoC. During this process, we discover 12,701 forgeable client-side requests split across 87 applications. We find that seven applications suffer from at least one zero-day client-side CSRF vulnerability that can be exploited to perform state-changing actions and violate the server's integrity.

Before presenting the evaluation results, we discuss the experimental setup (§3.3.1) and show properties of problem space and how JAW tackled them (§3.3.2). Then, we report the findings of our experiments (§3.3.3), and finally, conclude with the analysis of JAW's results (§3.3.4).

3.3.1 Experimental Setup and Methodology

Testbed. We select web applications from the Bitnami catalog [179] that offers ready-to-deploy containers of pre-configured web applications. We choose Bitnami applications due to their popularity (e.g., see [198]), diversity, and use by prior work (e.g., see [16]). At the time of the evaluation, Bitnami contains 211 containers. We discard 105 containers without web applications and duplicates, e.g., the same web application using different web servers. The remaining 106 web applications are: 23 content management system, 15 analytics, 11 customer relationship management, ten developer tools and bug tracking, eight e-commerce, eight forum and chat, five email and marketing automation, four e-learning, three media sharing, two project management, two accounting and poll management, and 15 other. The complete list of web applications is in Appendix A.1, among which we have WordPress, Drupal, GitLab, phpMyAdmin, and ownCloud.

Then, for each web application, we created one user account for each supported levels of privilege and a Selenium test case to perform the login. In total, we created 136 test scripts, ranging from one to five test cases per application.

JAW Inputs. The inputs of JAW are the seed URLs, the Selenium test cases, and a semantic type mapping. The seed URLs contain the URLs for the user login (total of 113 login URLs), whereas the test cases are the ones we prepared when configuring the testbed. Then, for all web applications, we used the semantic types listed in Table 3.1 in §3.2.2.

Methodology for Client-side CSRF Detection. We deployed the web applications under evaluation locally, and instantiated JAW against each of the targets. After the data collection and creation of the HPGs, we run a set of queries to identify attacker-controllable requests. We then use additional queries to identify the request fields under the control of the attacker and the type of control. We assess the accuracy of the query results via manual verification. For each forgeable request, we load the page in an instrumented browser and verify whether the manipulated inputs are observed in the client-side requests. For example, if the request uses data values of type `WIN.LOC`, we inject a token in the vulnerable page URL and search the token in the outgoing request. After confirming the forgeability of the request, we look for its use in an attack. First, we search for server-side endpoints performing security-relevant state-changing actions, such as modifying data on the server-side storage. Then, we construct a string that, when processed by the vulnerable page, it will result in a request towards the identified endpoint. Finally, we pack the string into a malicious URL and verify whether the attack works against a web application user with a valid session, who clicks on the URL.

Methodology for Impact of Dynamic Snapshotting. We performed additional experiments to assess the impact of our dynamic snapshotting approach in (i) vulnerability detection, and (ii) HPG construction. First, we prepared a variant of JAW, hereafter referred to as JAW-static, which follows a pure static approach for HPG construction and analysis (§3.1.1). Specifically, JAW-static does not consider the following dynamic information: fired events, handler registrations, HTTP messages, global object states, points-to analysis for DOM queries, dynamic insertion of script tags, and the DOM tree snapshot. We repeated our evaluation using JAW-static, and determined the lower bound of false negative and false positive vulnerabilities in JAW-static by comparing it to JAW's evaluation results. Also, we compare the differences in HPG nodes, edges and properties.

Then, we logged all the fired events that are not auto-triggered and that JAW failed to find their line of code for HPG construction. Such cases are an indication of false negative edges in HPGs generated by JAW. Accordingly, we manually review all cases to uncover the reasons for false negative edges. Finally, we conducted another experiment to assess the false positive and false negative edges as a result of using the DOM tree snapshots for points-to analysis of DOM queries. For all web pages, we instrumented the JavaScript code to log the actual element a DOM query is referring to, and compared it against the value that JAW resolved. JAW uses these resolutions to create ERDDG edges, opening the possibility for both false positive and false negative edges.

3.3.2 Analysis of Collected Data

Size of the Analysis. Starting from 113 seed URLs, JAW extracted 4,836 web pages, ranging from 1 to 456 web pages per web application, and about 46 web pages per application. The structural analysis of these URLs reveals that 865 of them have a hash fragment, an indication that these URLs carry state information for the client-side JavaScript program—a characteristic of single-page web applications. In total, 39 web applications use URLs with hash fragments. From the 4,836 pages, JAW extracted 228,763,028 LoC, which amounts to generating 4,836 HPGs by processing about 47,304 LoC per page. When looking at the origin of the code, we observed that the majority of it, i.e., 60.55%, is from shared libraries, e.g., jQuery (28,645 LoC per page and 138,525,092 LoC in total), whereas the remaining is application code in script tags (39.42% or 18,649 LoC per page, over 90,188,256 LoC in total) and a negligible amount is inline code (0.02% or 10 LoC per page, over 49,680 LoC in total).

Finally, at run-time, we observe that about 2.63% of the script tags are loaded dynamically (i.e., by inserting a script tag programmatically), over a total of 104,720 script tags. Also, JAW observed 51,974 events that are fired when loading the page (about 11 events per page) distributed across 46 event types, of which 38 are HTML5 types (e.g., animation and DOM mutation events) and 8 are custom. As we will show next, even if the number of run-time monitored events is negligible, their role in the analysis is fundamental.

Importance of Symbolic Modeling. The analysis of client-side programs requires to process 228,763,028 LoC of which 138,525,092 of them are for the libraries alone, about 60% of the total. Our analysis reveals that libraries are largely reused both across web applications and across pages. First, the 106 web applications in our testbed use in total 31 distinct libraries. Second, each page contains from zero to seven script libraries, with an average number of two libraries per page. Third, the total amount of code of the 31 libraries is 412,575 LoC, which is 335 times smaller than the total 138,525,092 LoC across all pages. Accordingly, pre-processing the library code to extract a symbolic model reduces by more than half (-60.37%) the effort required to generate HPGs, moving from 228,763,028 LoC to 90,650,511 (i.e., the sum of LoCs of the application, inline JavaScript, and the 31 libraries).

For each of the 31 libraries, JAW generates one HPG and extract a symbolic model. Table 3.2 shows an overview of the results of the symbolic modeling step. In total, JAW modeled 11,977 functions in around half an hour, half of which have the input-output relationship semantic types (i.e., 5,923 functions)—a relevant function behavior to correctly reconstruct the data flows of a program.

Role of ERDDG. In total, JAW generated 4,836 HPGs, one for each page, for a total of 508,810,412 nodes and 652,662,573 edges. Of these edges, the ones that are crucial to analyze

Table 3.2: Overview of symbolic modeling for shared JavaScript libraries.

Library	**Usage %**	**LoC**	**Funcs.**	*I/O*	**Time (s)**
JQuery	81.13%	10,872	428	238	57.54
Bootstrap	38.67%	2,377	55	55	41.07
JQuery UI	27.35%	18,706	320	320	82.33
ReactJS	9.43%	3,318	130	40	39.59
ReactDOM	9.43%	25,148	688	368	81.98
RequireJS	8.49%	1,232	50	50	35.72
AngularJS	5.66%	36,431	852	558	82.92
Analytics	5.66%	20,345	244	233	69.21
Backbone	5.66%	2,096	148	50	38.26
Modernizer	5.66%	834	292	21	34.50
Prototype	5.66%	7,764	266	243	54.10
YUI	4.71%	29,168	2,414	637	149.34
JIT	3.77%	17,163	430	255	69.11
ChartJS	2.83%	16,152	263	253	76,75
Dojo	2.83%	18,937	696	313	63.32
LeafletJS	2.83%	14,080	650	208	62.65
Scriptaculous	2.83%	3,588	97	84	46.11
HammerJS	1.88%	2,643	67	47	37.01
MomentJS	1.88%	4,602	138	138	45.44
ExtJS	1.88%	135,630	2,701	1,135	231.86
Vue	1.88%	11,965	638	288	62.77
YUI History	1.88%	789	20	10	18.41
Bootstrap Growl	0.94%	215	7	7	32.21
Bpmn-Modeler	0.94%	19,139	231	228	65.84
CookiesJS	0.94%	79	3	0	31.29
FlotChartsJS	0.94%	1,267	15	15	42.38
GWT WebStarterKit	0.94%	110	3	2	31.15
Gzip-JS	0.94%	280	4	4	31.87
Handlebars	0.94%	6,726	103	103	50.83
SpinJS	0.94%	190	4	4	31.99
SWFObject	0.94%	729	20	16	33.61
Total		412,575	11,977	5,923	1919.84

JavaScript programs are those modeling the transfer of control via event handlers. In total, JAW identified 64,854,097 event edges (i.e., registration, dependence and dispatch) of which 6,451,582 are dispatch edges, i.e., edges modeling the intention to execute the event handlers. For comparison, the number of call edges that also transfer the control to other sites of a program, are 7,179,021, meaning that the ERDDG representation enables the identification of +89.87% edges transferring the program control.

3.3.3 Prevalence of Forgeable Requests

The first step to detect client-side CSRF is the identification of lines of code that can generate attacker-controlled requests. For that, we prepared a set of queries as discussed in §3.2.3. Based on our threat model (§2.2.2), we considered different attacker-controlled inputs for JavaScript programs (see [32]) that can be forged by different attackers.

JAW identified 49,366 lines of code across 106 applications that can send an HTTP request, and marked 36,665 of them as unreachable during the page load or not using attacker-controlled inputs. The remaining 12,701 requests could be controlled by an attacker. We grouped these requests by the semantic types of the input source corresponding to different attackers (see §2.2.2), as shown in Table 3.3. We observe that the majority of applications, i.e., 87, sends at least one forgeable request at page load.

False Positives. Considering the high number of forgeable requests, we could not verify all of them via manual inspection. Instead, we first selected all requests across all groups, except for `DOM.READ`. Then, for `DOM.READ`, we focused on one request (randomly selected) for each web application, i.e., 83 requests. In total, we inspected 516 forgeable requests. For the inspection, we loaded the vulnerable page in an instrumented browser to inject manipulated strings and

Table 3.3: Number of forgeable requests and affected web applications.

Sources	**Forgeable**	**Apps**
DOM.COOKIES	67	5
DOM.READ	12,268	83
*-STORAGE	76	8
DOC.REFERRER	1	1
POST-MESSAGE	8	8
WIN.NAME	1	1
WIN.LOC	280	12
Total forgeable	12,701	87
Non-reachable code	36,665	101
Total	49,366	106

observe whether the outgoing requests include manipulated strings. We confirmed that all requests, except for one of the 83 DOM.READ requests include the manipulated content. After a careful investigation, we observed that the false positive occurs as a result of inaccurate pointer analysis of the context-sensitive `this` keyword, which has a run-time binding, and may be different for each invocation of a function depending on how the function is called, e.g., dynamically called functions, or different invocation parameters using a hierarchy of `call` and `apply` methods [196, 195] lead to different bindings of `this` keyword.

Exploitations. Next, we looked for practical exploitations for the 515 requests manually. In these experiments, we assumed a web attacker model for all input sources, except for cookies for which we assumed a network attacker model (see §2.2.2). We were able to generate a working exploit for 203 forgeable requests affecting seven web applications, all of them using data values of WIN.LOC, that can be forged by any web attacker. For the other groups of requests, we were not able to find an exploit. We point out that it is hard to achieve completeness when looking for exploitations manually as such a task requires extensive knowledge of web applications for identifying target URLs and the points where an attacker could inject malicious payloads. The fact that we could not find an exploit does not imply that an exploit does not exist. For these cases, we confirmed that the JavaScript code sends HTTP requests by processing data values taken from different data structures unconditionally. A highly motivated attacker could eventually find a way to inject malicious payloads in these data structures and exploit these forgeable requests.

3.3.4 Analysis of Forgeable Requests

In this section, we have a closer look at the degree of manipulation an attacker can have on the forgeable requests of Table 3.3. We extracted the stack trace for the lines of code that send forgeable requests and characterized the vulnerable behavior along three dimensions: forgeable request fields, type of manipulation, and the request template.

Forgeable Fields. First, the request field(s) that can be manipulated can determine the severity of the vulnerability. For example, if the attacker can change the domain name of a request, the client-side CSRF could be used to perform cross-origin attacks. We grouped web applications in four categories, based on the field being manipulated and found that in nine, 34, 41, and 41 web applications, an attacker can manipulate the URL domain, the URL path, the URL query string, and the body parameter, respectively. Also, we grouped applications by the number of fields that can be manipulated in a request. In total, 55, 34, and 12 applications allow modifying one, more than one, and all fields, respectively.

Operation to Forge a Field. Another factor that influences the severity is the operation that copies a manipulated value in one or more fields. We found that 28 applications allow an attacker to change the value of one or multiple fields. Also, 38 and 28 applications allow an attacker to add one or multiple fields by appending and prepending the attacker-controlled

Table 3.4: Taxonomy of client-side CSRF. Each template reflects the level of attacker's control on the outgoing HTTP request. * are the templates for which we found an exploit.

Outgoing HTTP Request						Total	
Dom.	Path	Query	Body	Part	Control	Reqs	Apps
		✓		One	`-, A, -`	16	11
			✓	One	`-, A, -`	5	5
			✓	One	`W, -, -`	(*)166	25
			✓	One	`-, -, P`	1	1
	✓			One	`W, -, -`	28	1
	✓			One	`-, A, -`	7	7
	✓			One	`-, -, P`	6	6
		✓		One	`-, -, P`	11	11
	✓		✓	Mult	`-, A, -`	4	1
	✓		✓	Mult	`W, -, -`	(*)20	1
	✓	✓		Mult	`W, A, P`	6	1
		✓	✓	Mult	`W, -, -`	2	1
		✓		Mult	`-, A, -`	7	7
			✓	Mult	`-, -, P`	2	2
	✓			Mult	`-, A, -`	3	3
		✓		Mult	`-, -, P`	1	1
			✓	Mult	`-, A, -`	5	5
	✓			Mult	`-, -, P`	6	6
			✓	Mult	`W, -, -`	28	8
	✓	✓		Any	`W, -, -`	1	1
✓	✓	✓		Any	`W, -, -`	(*)185	8
✓	✓	✓	✓	Any	`W, -, -`	1	1
			✓	Any	`W, -, -`	(*)1	1
			✓	Any	`W, A, -`	2	2
	✓	✓	✓	Any	`W, -, -`	1	1

Legend: A=Appending; P=Prepending; W=Writing.

string to the final string, respectively.

Forgeable Request Templates. We characterize HTTP requests via templates, where we encode the type and number of fields that can be manipulated as well as the type of operation. Table 3.4 lists all templates, and for each template, it shows the number of matching requests and web applications using them. In total, we identified 25 distinct templates. We observed that the majority of web applications use only one template (i.e., 68 applications) across all their web pages or two templates (i.e., 17 applications).

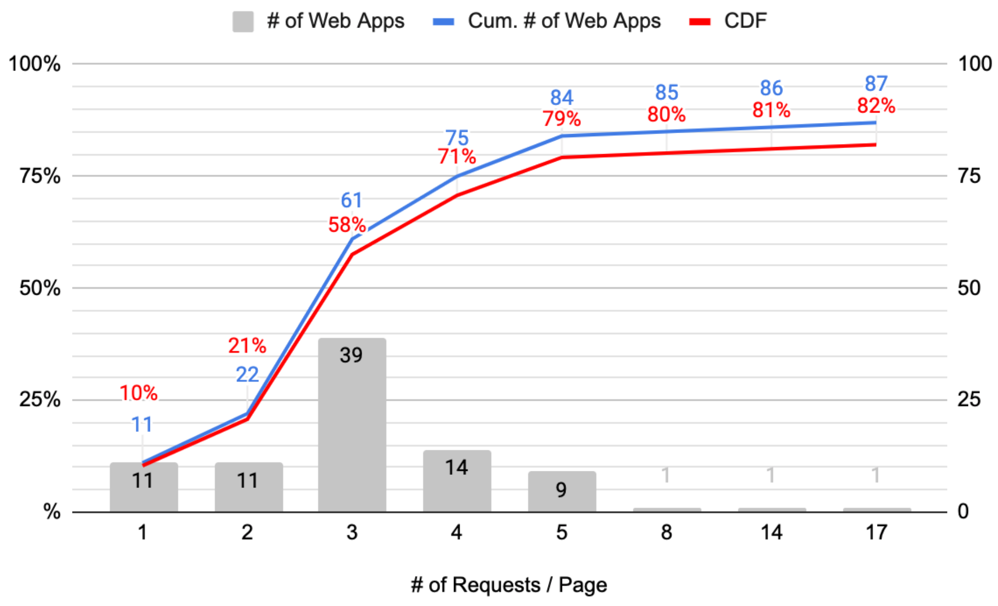
Distribution of Forgeable Requests. Figure 3.5 depicts the average number of forgeable HTTP requests per application web pages against the number of web applications, the cumulative number of web applications, and their cumulative distribution function (CDF). In this figure, the average number of forgeable requests per web page is rounded up for each application. The figure shows that JAW finds on average between one to five forgeable HTTP requests for the majority of vulnerable applications, i.e., 84, with only a few vulnerable applications, i.e., three, having more than five forgeable requests in their pages. Also, according to the figure, 82% of the applications in our testbed have at least one web page with a forgeable client-side HTTP request, an alarming signal that client-side CSRF vulnerabilities are very prevalent in the wild.

3.3.5 Exploitations and Attacks

The 203 exploitable client-side CSRF affect seven targets, as shown next. Our exploits attack web applications the same way classical CSRFs do, i.e., by performing security-relevant state-changing requests. In addition, we found exploitations of client-side CSRF that enable XSS and SQLi attacks, which cannot be exploited via the classical attack vector.

SuiteCRM and SugarCRM. In total, we found 115 and 38 forgeable requests in SuiteCRM and SugarCRM, which can be exploited to violate the server's integrity. In both applications, the JavaScript code reads a hash fragment parameter, e.g., `ajaxUILoc`, and uses it verbatim as the endpoint to which an asynchronous request is submitted. An attacker can forge any

Figure 3.5: Forgeable requests per application web page.



arbitrary request towards state-changing server-side endpoints to delete accounts, contacts, cases, or tasks–just to name only a few instances that we confirmed manually.

Neos. We found eight forgeable requests in Neos. In all of them, each parameter p of the HTTP request originates from the page's URL parameter `moduleArguments[@p]`. Among these, we have, for example, the action and controller parameters that are used by the backend server to route the request to internal modules. Such behavior allows an attacker to direct a request to any valid internal module, including those implementing state-changing operations. For example, we exploited this behavior to delete assets from the file system.

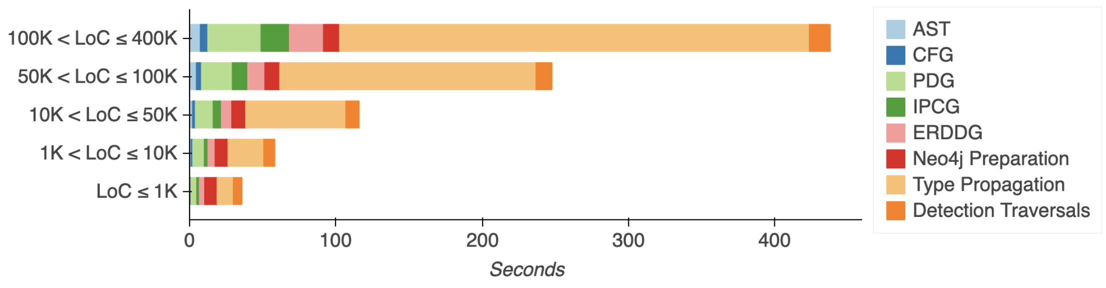
Kibana. We found one forgeable request, generated by Timelion, a Kibana's component that combines and visualizes independent data sources. Timelion allows running queries on data sources using a own query syntax. The JavaScript code can read queries from the page's URL fragment and pass them to the server side. As a result, an attacker can execute malicious queries without the victim's consent or awareness.

Modx. We discovered 20 forgeable requests in Modx that can be exploited in two distinct ways. First, Modx's JavaScript fetches a URL string from the query parameter of the page's URL, and uses it verbatim to submit an asynchronous request with a valid anti-CSRF token. Similarly to SuiteCRM and SugarCRM, an attacker can forge requests towards state-changing server-side endpoints. Also, in one forgeable request, Modx copies a page's URL parameter in a client-side request, which is reflected back in a response and inserted into the DOM tree, allowing an attacker to use client-side CSRF to mount client-side XSS. Based on our manual evaluation, the attacker can exploit the client-side XSS only via client-side CSRF.

Odoo. We found one forgeable request that uses an `id` parameter of the URL fragment to load a database entity. We discovered that the server uses this parameter in a `SQL` query which is not properly validated, resulting in an SQLi vulnerability. We note that, due to a anti-CSRF token, the exploitation of the SQLi vulnerability via direct requests is extremely hard without exploiting first the client-side CSRF vulnerability.

Shopware. We found 20 forgeable requests sent by Shopware on page load. The code maps the page's URL hash fragment to different parts of the outgoing request. First, the code uses

Figure 3.6: Average time required for JAW to construct and analyze a hybrid property graph categorized by lines of code (LoC).



the first fragment of the hash fragment as URL path of the outgoing request. Then, it uses the remaining fragments as parameters of the outgoing request body. This allows an attacker, for instance, to delete products of the shop's catalog.

3.3.6 Run-time Performance

We deployed the web applications under evaluation on a desktop computer (running MacOS Mojave 10.14.3 on an Intel Core i5 with 2.4 GHz, 16 GB RAM, and a SSD), and performed the data collection step (§3.2.1). We let JAW run for a maximum of 24 hours on each web application, although after a few hours the data collection module typically does not find any new URLs. Then, we imported the collected data on our own server (running Ubuntu 18.04 on an Intel(R) Xeon(R) CPU E5-2695 v4 with 2.10 GHz and 72 cores, 252 GB RAM), and instantiated JAW with the data to find client-side CSRF vulnerabilities. We log all processing times for throughput evaluation. Figure 3.6 depicts the average processing time for each tool component in order to construct and analyze a HPG. As shown in the figure, the processing time increases as the LoC grows. The least time consuming operations are AST and intra-procedural CFG construction. JAW also a incurs a preparation delay in order to import the constructed property graph into a Neo4j database which typically lasts around 8-11 seconds based on the LoC. The most time consuming operation is the semantic type propagation.

3.3.7 Impact of Dynamic Snapshotting

We designed and carried out experiments to show the impact of dynamic snapshotting in vulnerability detection and HPG construction following the methodology that we presented in §3.3.1). In this section, we present our findings.

3.3.7.1 Vulnerability Detection

We repeated our evaluation using JAW-static, and compared the results with JAW (§3.3.1). In total, JAW-static found 48,543 requests, out of which 11,878 reported to be forgeable. By comparing the difference, we observed that JAW-static has detected 840 less forgeable requests (i.e., a lower bound of +7.07% false negatives). Out of the 840 false negatives, 161 cases are vulnerabilities for which we found an exploit, i.e., JAW-static does not detect 79.3% of the *exploitable* client-side CSRF vulnerabilities that was detected by JAW. Additionally, JAW-static reported 17 more cases that were not vulnerable (i.e., a lower bound of +0.15% false positives). We manually examined *all* the false positive and false negative cases to discover the underlying reasons.

False Positives (FP). Out of 17 FPs, 12 were due to non-existing dynamically fetched code (i.e., by dynamic insertion of script tags) where the value of the tainted variable changed in the dynamic code. Such FPs are eliminated in JAW because it monitors the program execution leveraging the DOM tree and HTTP messages. Then, 3 out of the 17 cases were due to a subsequent removal of the event handlers using dynamic code evaluation constructs with dynamically generated strings. Finally, the last two FPs occurred due to the removal of elements from the DOM tree, and thus the implicit removal of their event handlers. Similarly, such FPs do not occur with JAW, as it monitors the fired events and their handlers at run-time.

False Negatives (FN). We observed that almost half of the FNs, i.e., 405, occurred because the vulnerability resided in dynamically loaded code. For 78 and 7 FNs, points-to analysis for DOM queries were not accurate as the state of the DOM tree and environment variables were necessary for such analysis, respectively. The remaining 350 FNs stemmed from the fact that the JavaScript program used `setTimeout` and `eval` for firing events by generating code at run-time.

3.3.7.2 HPG Construction

In total, JAW-static generated 498,054,077 nodes and 639,323,601 edges for the 4,836 HPGs, which is 10,756,335 nodes (-2.11%) and 13,338,972 edges (-2.04%) less than JAW (false negatives). Out of the total missing edges, 1,048,172 are ERDDG edges that are critical for modeling events, and the remaining 12,290,800 edges are the AST, CFG, PDG and IPCG edges. Furthermore, JAW-static misses 16,710 edge properties (set on ERDDG registration edges) that mark if an event handler has been triggered at run-time, and that has not been marked with static analysis.

Following additional experiments based on our methodology (§3.3.1), we logged the fired events that JAW cannot map to their line of code. In total, JAW observed 51,974 events at run-time across 4,836 HPGs, out of which 34,808 were already marked by static analysis and fired dynamically. The remaining 17,166 events trigger at run-time, while not captured by pure static analysis. Out of the 17,166 events, JAW fails to find the corresponding event handlers of 456 events in the code (0.88%), an indication of FN nodes and edges in the HPG. Manual analysis revealed that the reasons for the majority of cases (387 events) is the use of `eval` and `setTimeout` functions with dynamically constructed strings for firing events. The remaining 69 events are not mapped due to the dynamic loading of code and in ways that JAW does not monitor (e.g., loading code from inside iframes).

Finally, we assess the FP and FN edges introduced by the usage of the DOM tree snapshots when performing points-to analysis of DOM queries. In total, JAW encountered 241,428 DOM query selectors in 4,836 HPGs, out of which in 127 selectors (0.05%) JAW imprecisely resolved the DOM element the query is pointing to. To determine the ERDDG dispatch edges, JAW compares the pointers for a total of 87,340 pairs of DOM query selectors against each other (i.e., an event dispatched on one DOM query selector is linked to its event handler that uses another query selector for the event registration). Our evaluation suggests that JAW accurately decides to connect or not to connect a dispatch edge between the dispatch and registration sites in 87,212 cases (decision accuracy of 99.85%), with 56,923 true positives and 30,289 true negatives. In the remaining 128 cases, JAW's decision to create or not to create an edge is inaccurate, with 94 FN and 34 FP edges (decision inaccuracy of 0.15%). Interestingly, we observed that such FP and FN edges may occur for query selectors that are interpreted within 53.7 mili-seconds of page load (on average), and a maximum of 92.5 mili-seconds, which is up to ca. ten times lower than the average access time of all query selectors, i.e., 559.2 mili-seconds. In this experiment, we used run-time program instrumentation to obtain the ground truth for assessing JAW's accuracy in HPG construction. However, such techniques come with performance hits, and are poorly suitable for large HPGs (e.g., in model construction, and vulnerability detection). We believe the impact of JAW's FP and FN edges as a result of DOM snapshots is negligible.

3.4 Summary

In this chapter, we presented JAW, to the best of our knowledge the first framework for the detection and analysis of client-side CSRF vulnerabilities. At the core of JAW is the new concept of HPG, a canonical, static-dynamic model for client-side JavaScript programs. Our evaluation of JAW uncovered 12,701 forgeable client-side requests affecting 87 web applications. For 203 of them, we created a working exploit against seven applications that can be used to compromise the database integrity. We analyzed the forgeable requests and identified 25 different request templates. This work has successfully demonstrated the capabilities of our paradigm for detecting client-side CSRF. In the following chapters, we show how this approach can be useful to study other vulnerability classes.

4
Studying Request Hijacking Vulnerabilities in the Wild

In this chapter, we conduct the first systematic and comprehensive study of client-side request hijacking, covering new vulnerability variants, detection, prevalence, and impact, addressing RQ2 of §1.1. First, we review browser API capabilities and explore potential attacks when attackers manipulate one or more inputs of request-sending APIs (§4.1), identifying 10 different client-side request hijacking vulnerabilities, including seven new variants. Then, we present JAW-v2, an automated tool to detect and study client-side request hijacking vulnerabilities at scale (§4.2), which we use as our vehicle for a large-scale investigation into this issue in §4.3. In total, JAW-v2 uncovered 202K vulnerable data flows, affecting 9.6% of the top 10K sites, of which 49 that we manually confirmed to be exploitable, including Microsoft Azure, Indeed, Starz, and Reddit. Finally, in §4.4, we identify, review and assess the efficacy and adoption of existing countermeasures, showing that CSP and COOP cannot mitigate over 41% and 93% of request hijacking attacks. Also, we analyze coding mistakes of the 961 vulnerable websites and extract eight common insecure input validation patterns to avoid.

4.1 API Capabilities and Attack Systematization

In this section we address RQ2.1 of §1.1.2, where we systematically assess modern web browser APIs and their capabilities for sending various types of client-side requests (§4.1.1). Then, we examine each API call to evaluate the resulting vulnerabilities and attacks when an attacker controls one or more API inputs (§4.1.2). Finally, we assess the prevalence of request API usage on the Web platform (§4.1.3).

4.1.1 Browser API Capabilities

Client-side web applications have access to a wide range of browser functionality via JavaScript Web APIs. An important group of these APIs is responsible for creating and sending network requests, which malicious actors could exploit for request hijacking. To compile a comprehensive list of request-sending APIs susceptible to such abuse, we performed a systematic search of the Web request specifications [6, 23, 199, 5, 201, 200] from WHATWG [202] and W3C [203] repositories. Our search focused on identifying JavaScript Web APIs capable of creating network requests.

As a result, we identified a total of 10 request APIs across six broad request categories. Each API features different characteristics in terms of their supported capabilities, e.g., the network schemes and methods available for a given API, the configurable fields of the request (e.g., body and headers), and the constraints the APIs may be subject to by default, such as the Same-Origin Policy [160]. The resulting APIs are summarized in Table 4.1. By examining these request APIs, we uncover potential entry points for various forms of request hijacking, and their consequences.

4.1.2 Systematization of Request Hijacking Attacks

In this section we examine the security impact assuming the threat model presented in §2.2.3. That is, an attacker can control the URL (and if applicable, the body and header) of network requests for each of the APIs discovered in §4.1.1. First, we systematically surveyed the existing academic [17, P1, P4, 204, 8, 32, 211, 212, 16, 206, 213, 207, 109] and non-academic [210, 214, 208, 205, 209] literature, looking for known attacks leveraging these APIs. Then, we conducted an in-depth analysis of the threat landscape, where we examined the potential attacks resulting from an attacker's capability to manipulate different fields of each request-sending API.

As a result, we identify a total of 10 client-side request hijacking vulnerability variants, of which only three are previously known. Table 4.2 presents the list of request hijacking

Table 4.1: Overview of security-sensitive JavaScript APIs that inititate client-side requests, along with their supported capabilities, default constraints and usage in top 10K Tranco websites (Cf. §4.3). The table is ordered by the API usage in the wild.

			Capabilities								
	API	**Req. Type**	**Schemes**	**Methods**	**U**	**B**	**H**	**Specs**	**# Sites**	**# Pages**	**# Calls**
#1	Location Href	Top-Level Navigation	HTTP(S), JS	GET	●	○	○	[199] §7.2.4	8,044	214,554	1,096,306
#2	XMLHttpRequest	Async. Request	HTTP(S)	Any	●	●	●	[200] §3.5	7,522	407,819	2,884,556
#3	sendBeacon	Async. Request	HTTP(S)	POST	●	●	○	[6] §3.1	7,061	291,580	2,824,388
#4	Window Open	Window Navigation	HTTP(S)	GET	●	○	○	[199] §7.2.2.1	6,972	162,153	559,592
#5	Fetch	Async. Request	HTTP(S)	Any	●	●	●	[23] §5.4	5,215	105,463	403,701
#6	Push	Push Subscription	HTTP(S)	GET, POST	●	●	○	[5] §3.3	1,528	23,566	40,567
#7	WebSocket	Socket Connection	WS(S)	GET	●	●	○	[201] §3.1	1,280	33,724	145,713
#8	Location Assign	Top-Level Navigation	HTTP(S), JS	GET	●	○	○	[199] §7.2.4	987	10,092	22,309
#9	Location Replace	Top-Level Navigation	HTTP(S), JS	GET	●	○	○	[199] §7.2.4	731	6,421	14,309
#10	EventSource	Server-Sent Event	HTTP(S)	GET	●	○	○	[199] §9.2	453	1,690	5,503

Legend: SSC= SameSite Cookies; SOP= Same-Origin Policy; U= URL; B= Body; H= Header;
● = Supported Capability or Applicable Constraint; ○ = Otherwise.

Table 4.2: Overview of client-side request hijacking vulnerabilities and attacks. Rows marked with ▣ are new (i.e., client-side variants of) vulnerabilities and ⊞ represent vulnerabilities for which we consider a new API or exploitation. For new vulnerabilities, related references refer to their server-side vulnerability counterparts.

	Vulnerability	**Reqs.**	**CSRF**	**XSS**	**WS Hijack**	**SSE Hijack**	**Inf. Leak**	**Open Red.**	**DoS**	**Related Ref.**
⊞	Forge. Async Req. URL	#2, 3, 5	●	○	○	○	●	○	○	[P1, P4, 204]
⊞	Forge. Async Req. Body	#2, 3, 5	●	○	○	○	○	○	○	[17, P1, 204, 16]
▣	Forge. Async Req. Header	#2, 5	●	○	○	○	○	○	○	-
▣	Forge. Push Req. URL	#6	○	○	○	○	●	○	●	-
▣	Forge. Push Req. Body	#6	●	○	○	○	○	○	○	[205–207]
▣	Forge. EventSource URL	#10	○	○	○	●	●	○	○	[208]
▣	Forge. WebSocket URL	#7	○	○	●	○	●	○	○	-
▣	Forge. WebSocket Body	#7	●	○	●	○	○	○	○	[210, 204, 211, 212, 209]
	Forge. Location URL	#1, 8, 9	●	●	○	○	○	●	○	[32, 2, 213]
▣	Forge. Window Open URL	#4	●	●	○	○	○	●	○	-

Legend: Forge.= Forgeable; SSE= Server-Sent Event; WS= WebSocket;
#i= row i in Table 4.1; ● = Applicable Attack; ○ = Otherwise.

vulnerabilities, together with the responsible APIs and attacks which are made possible as a consequence of each vulnerability explained in more detail in the following.

4.1.2.1 Asynchronous Requests

Asynchronous requests such as `XMLHttpRequest` [200] or the low level `fetch` API [23] are typically used to communicate with web services such as REST APIs, without causing the top-level page to reload. Attacker manipulation of the URL, body, or header of asynchronous requests in client-side JavaScript programs can lead to the victim performing unwanted actions on behalf of the attacker, i.e., client-side CSRF [1, P1], similarly to their traditional counterpart [17, 21, 70, 16]

We are currently unaware of studies exploring the manipulation of `sendBeacon` API [6] for client-side CSRF attacks. Furthermore, attacker control of asynchronous request URLs can also lead to information leakage, which was not considered by prior works (e.g., [P1]). In this case the attacker manipulates the URL host to point to a malicious server, where they can access sensitive information stored in the request header or body, e.g., login credentials, personally identifiable information (PIIs), and CSRF tokens.

4.1.2.2 Push Requests

Push notifications [5] allow a web server to asynchronously send messages to a browser, even if the web application is not currently loaded. The Push API requires subscription to a push service via an HTTP POST request, and a browser can request new messages via HTTP GET requests. If Push subscriptions do not have anti-request forgery tokens, attackers can conduct classical CSRF attacks [205].

While not explored before, similar attacks are possible in the context of client-side programs. For example, when creating a Push subscription, the client sends information such as the subscription endpoint and public key in the body of an HTTP POST request. Attacker control of the request body would allow manipulation of these parameters and hence CSRF [205–207], e.g., by overwriting the subscription endpoint the application saves in the backend, the attacker can redirect Push messages to an arbitrary server. We note that in the case of Push requests, the URL must take a specific value (i.e. the endpoint listening for Push subscriptions), so URL manipulation will not usually lead to CSRF attacks. However, setting an invalid value enables attackers to cause a persistent client-side DoS, which can be mitigated when users reset their browser notification permissions. In addition, control of the Push URL could lead to information leakage if the request is redirected to an attacker-controlled server, e.g., the leaked Push endpoint and encryption key can be exploited to send malicious messages to the victim's browser.

4.1.2.3 Server-Sent Events

Server-sent events (SSEs) [208, 199] allow servers to push messages to a browser at any time, without waiting for a new request. SSEs are initiated via an HTTP GET request to a URL specified in the `EventSource` constructor. By manipulating the EventSource URL, attackers can redirect the request to a malicious server and achieve SSE hijacking, whereby malicious events can be sent to the victim's browser. Similarly to asynchronous and push requests, redirection of the EventSource URL can also lead to information leakage.

4.1.2.4 Web Sockets

WebSockets [215, 201] enable full-duplex, event-driven communications between browsers and servers, initiated via an HTTP GET request. An attacker controling the WebSocket connection can perform cross-site WebSocket hijacks (CSWSH) [204, 212, 209]. In this scenario, an attacker can embed a WebSocket to a target website on their own site. When a victim visits this malicious site, the victim's browser is tricked into performing authenticated actions on behalf of the attacker. In contrast to write-only CSRF attacks, CSWSH allows full read/write communication. In the context of client-side vulnerabilities, we show that similar attacks are possible if the attacker can control the URL used to perform the initial handshake, redirecting the request to a malicious server. As the attacker controls the WebSocket, this can also lead to information leakage from the victim's browser. Finally, controling the data used in WebSocket messages leads to message hijacking and potentially CSRF.

4.1.2.5 Top-Level Navigation Requests

Top-level navigation requests via the `location` API allow manipulation of the current browser URL, and trigger a new HTTP GET request when called [199]. Attacks leveraging this category of requests have been considered in the past. For example, manipulating the `location` URL can lead to client-side XSS by exploiting the `javascript` protocol if the entire URL is controlled by the attacker [32, 213]. Alternatively, replacing the full URL with a malicious URL will force an open redirect of the browser to a different site. Finally, even partial hijack of the URL (e.g., query parameters) can trigger the application to perform actions on behalf of the attacker leading to CSRF. This usually occurs if the application implements state-changing GET requests [21] or allows forging POST requests with GET where it incorrectly accepts and processes incoming requests regardless of their HTTP method, as shown in [P4].

4.1.2.6 Window Navigation Requests

The `window.open` API triggers a top level HTTP request in a specific browser context, such as a new window or the current one (i.e., redirection). Similarly to the `location` API, control of the `window.open` API could also lead to CSRF, XSS and open redirects. However, unlike `location`, we are not aware of previous work that has studied this vulnerability.

4.1.3 Request API Prevalence

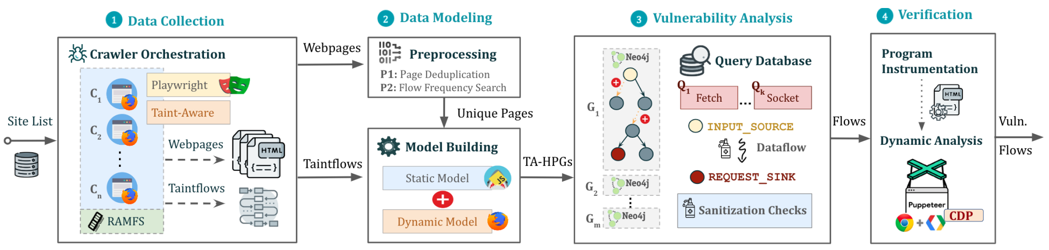
Having examined the web APIs which are susceptible to request hijacking, we also measured their prevalence in the wild. The last three columns of Table 4.1 list the number of sites, pages and calls of a particular API found in the top 10K Tranco websites. More details on the dataset and crawling strategy can be found in §4.3.

Overall, we find 9,901 domains which contain at least one API related to client-side requests, with a total of ~7.9M API calls across 1,032,795≈1M webpages. Top-level navigation requests via `location.href` are the most widespread, being present on more than 8K sites. Asynchronous requests via the `XMLHttpRequest` API are the most widely-used, with almost 3M calls across over 400K pages. We observed that request hijacking threats have not been considered for over 44.7% of API calls by prior work given the new vulnerability variants presented in Table 4.2. The widespread usage of request-related APIs in the wild, coupled with a wide variety of potential vulnerabilities, presents a tantalizing attack surface for hackers. The remainder of this paper is dedicated to techniques for the detection and evaluation of these vulnerabilities in the wild.

4.2 Vulnerability Detection

Starting from our systematization of vulnerabilities presented in §4.1, we now formulate our approach to detect and study request hijacking vulnerabilities, thereby addressing the first part of RQ2.2. Client-side request hijacking vulnerabilities arise due to the presence of insecure data

Figure 4.1: Architecture of JAW-v2.



flows from attacker-controlled inputs to request-sending instructions. In this paper, we design and implement an open-source, static-dynamic analysis tool, called JAW-v2, to detect such insecure data flows.

Figure 4.1 depicts the architecture of JAW-v2. Broadly, it comprises four main components: 1 a data collection module that gathers Web resources and dynamic taint flows from webpages, 2 a data modeling module that processes the collected data to identify and model unique webpages, creating a property graph for each one, 3 a vulnerability analysis module that traverses this graph following the propagation of unvalidated data flows from input sources to request-sending functions, and finally 4 a dynamic verification module that confirms the potential forgeability of requests. The rest of this section describes each component.

4.2.1 Data Collection

The first step of our analysis pipeline involves collecting client-side code and runtime values (e.g., DOM snapshots) of web applications for security testing. Starting from a list of sites under test like Tranco [216], JAW-v2 instantiates N crawling workers and continues orchestration until all input sites have been crawled. We developed a taint-aware crawler based on Playwright [217], an instrumented version of Firefox known as Foxhound (v98.0.1) [107, 106], and Firefox DevTools [218]. Since Foxhound does not provide instrumentation support for all request APIs listed in Table 4.1, we added further instrumentation to provide taint tracking support for these APIs (hereafter, Foxhound$^+$). Given a domain as input, the crawler visits webpages with a depth-first strategy, and stops when it does not find new URLs or visited a maximum of 200 URLs per site. During the visit, the crawler collects the following information: webpage resources (e.g., scripts), DOM snapshots, global objects' properties, event traces, network requests and responses, and finally dynamic taint flows from program inputs to security-sensitive instructions, such as request-sending functions.

4.2.2 Data Modeling

Preprocessing Given the webpages' data collected by the crawler, JAW-v2 performs data pre-processing for efficiency and scalability reasons. For example, JAW-v2 pre-processes the client-side code to filter out near-duplicate webpages [220, 219] by comparing SHA-256 script hashes, which allows it to focus on pages with distinct JavaScript code, reducing the overall effort for program analysis. Similarly, JAW-v2 can perform other types of data preprocessing, such as (custom) search-based filtering of data or code normalization, as discussed in §3.2.2.

Model Building After removing duplicate webpages, JAW-v2 instantiates a pool of workers to generate HPGs of the client-side programs under test, using an extended engine of JAW (Cf. §3.2). Then, these HPGs are enriched with taint flow information provided by Foxhound$^+$ to patch missing HPG edges due to static analysis shortcomings, and finally stored in a Neo4j [186] graph database which we can query for security testing.

Taintflow-Augmented (TA) HPGs. In this paper, we formulate the request hijacking vulnerability detection task over HPGs, where we intend to identify request-sending instructions that are triggered at page load, and that are susceptible to manipulation by attackers through program inputs. Unfortunately, conducting such inter-procedural reachability and data flow analysis tasks is non-trivial due to the dynamic nature of client-side JavaScript programs (e.g., [129, 11, 12, 14, P1, 50]). While HPG state values (i.e., environment properties and event traces [P1]) help to alleviate many of JavaScript static analysis shortcomings (e.g., imprecise control and data flow dependencies) by reasoning on concrete object snapshots (e.g., points-to analysis and triggered event handlers), they are not sufficient to identify many of the other missing call and data flow connections in the graph. Accordingly, in this paper, we use fully-fledged, in-browser dynamic taint tracking to further augment HPGs by adding supplementary edges and labels to nodes (e.g., to mark reachability, semantic types and runtime variable values), thus reconstructing missing connections that are reachable at page load, which are otherwise missed by static analysis.

TA-HPG Construction. We used Foxhound$^+$ to collect dynamic taint flows from input sources to all sink types, including those that are not request-sending instructions (Cf. Table A.1), so that we can complement as many potentially missing elements as possible in the HPG. Specifically, we first extract the dynamic call graph and data flow graph from Foxhound$^+$, and merge them with static call graph and PDG, respectively. To do the match between the dynamic and static graphs for merging, we first determine in which script file an instruction or node is located by comparing the script hash in the two models and, then, use the line of code to determine the top-level (i.e., CFG) node in the HPG for that instruction. When merging the dynamic data flow graph with PDG, we create data dependency edges if an edge is missing, with the labels being the variable name reported in the taint flow whose data is propagated. Conversely, if a PDG edge already exists between two nodes, we add a label to that edge marking the runtime value of the propagated variable.

Similarly, when merging the two call graphs, we create a new edge if it is missing and label it with the invoked function and parameter names as well as concrete parameter values of the function call. However, when the call edge exists in the HPG, we only enrich its information by adding the runtime values of the call site parameters. Finally, we added labels to all sources and sink nodes as semantic types, capturing the semantic of those instructions, e.g., the type `RD_DOC_URL` is set for instructions that read the value of `document.URI`, and then propagated to other HPG nodes following the calculation of the program. We exemplify our approach in more detail in §4.2.5. As we will show in §4.3, this configuration facilitates a comprehensive representation of program dynamics, enabling enhanced analysis capabilities for vulnerability discovery.

Static Analysis Engine Enhancements. To enhance JAW's HPG generation, we made several modifications addressing incomplete ES6 support for improved control transfer modeling and data flow analysis. For example, we added support for asynchronous operations using the `Promise` object and `setTimeout()` callbacks [221], improving the precision of call graph and PDG edges. Additionally, we applied multiple optimizations to improve scalability, such as handling inefficiencies in iteration constructs during the PDG construction and managing Neo4j graph databases in parallel by creating an orchestrator using ineo [222]. These modifications resulted in more precise analysis and improved scalability for constructing HPGs.

4.2.3 Vulnerability Analysis

After modeling the client-side code as a TA-HPG, we define the task of detecting request hijacking vulnerabilities as a graph traversal problem. Specifically, we intend to search for program instructions that send sensitive requests at page load, whose parameters originate from attacker-controlled program inputs. As the first step, we identify TA-HPG sources that

Listing 4.1: Example client-side request hijacking vulnerability derived from `bbc.com`, which is not captured by JAW's static analysis engine [P1].

```javascript
var c = {}, i = 0;
// handle incoming postMessages
window.addEventListener("message", h);
function h(e){
  if(e.origin.indexOf("bbc.com") > -1){
    i = i + 1;
    // [...]
    var d = JSON.stringify({
      "csrf_token": "xyz-token",
      "state": {...},
    });
    var u = e.data + '/userinfo';
    c["r" + i] = new
        Function("httpPostRequest("+ u + "," + d + ")");
  }
}
function httpPostRequest(url, body){
    // [...]
    navigator.sendBeacon(url, body)
}
// remember state upon closing the session
window.addEventListener("visibilitychange", saveState);
function saveState(e) {
  if (document.visibilityState === "hidden") {
    for(let j=1; j<= i; j++){
      c["r" + j]();
    }
}}
```

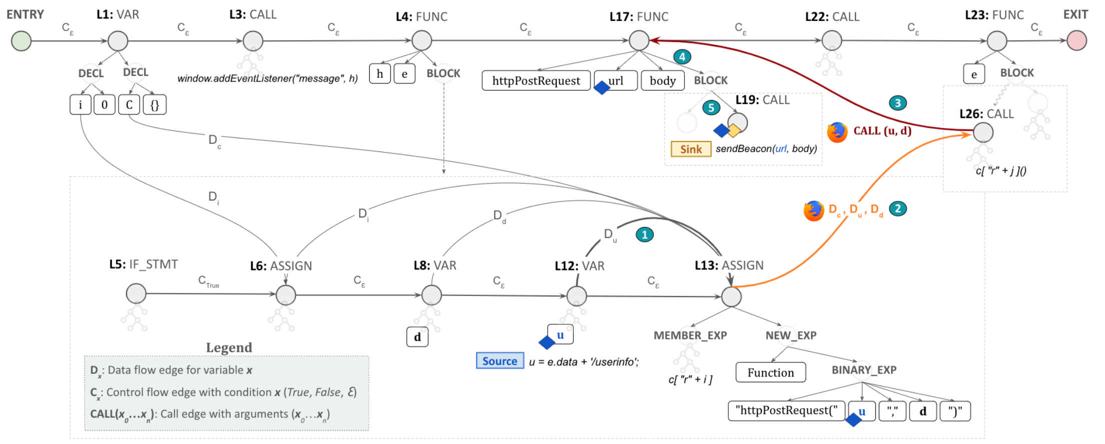
read attacker-controlled inputs (Cf. §2.2.3), and assign them a relevant semantic type similarly to JAW [P1], e.g., we set a label named `RD_WIN_LOC` for instructions that read the URL through `window.location` API. Then, given a list of browser APIs that are used for sending requests (Cf. Table 4.1), JAW-v2 searches the TA-HPG to identify nodes using these APIs, and marks them as a sink by assigning them a relevant semantic type, e.g., the label `WR_ASYNC_REQ_URL` is set for instructions that write the URL of an asynchronous request, such as `XMLHttpRequest.open()`.

Finally, to discover vulnerable paths, JAW-v2 performs data flow analysis by propagating semantic types from sources to sinks over PDG, CFG, CG, and ERDDG edges, and selects unvalidated paths where a node with a sink semantic type is tainted with a source type and picks up the attacker-controlled values. Then, JAW-v2 performs reachability analysis to check if the vulnerable path may correspond to lines of code executed at page load. To do that, it starts from both source and sink nodes and follows backward CFG, ERDDG, and CG edges until it reaches the CFG entry node or there are no longer edges matching the criteria to backtrack, and selects data flows where both the source and sink are reachable nodes. Ultimately, this component outputs a set of paths with potential data flows from a source to a request sink.

4.2.4 Vulnerability Verification

Given a set of candidate request hijacking data flows, the goal of this step is to confirm the feasibility of each flow dynamically and eliminate potential false positives. We relied on Playwright [217] and Chrome DevTools Protocol [108] to perform runtime monitoring, where we instrumented browser APIs responsible for sending requests (Cf. Table 4.1) and intercepted network messages that occur at page load. Specifically, for each request hijacking data flow, we input a token to the corresponding source, load the webpage in an instrumented browser controlled with Playwright and search for the token in the client-side request to check whether the manipulated inputs are observed. We test each candidate data flow both in the affected webpage and all its near-duplicate pages, which have the same set of scripts but potentially

Figure 4.2: Excerpt of the TA-HPG for the example in Listing 4.1. Connections highlighted in orange and red represent missing PDG and call graph edges that are reconstructed using dynamic taint flows of Foxhound^{+}, which are necessary for vulnerability discovery (steps 1-5). Blue and yellow diamonds attached to nodes represent source and sink semantic types propagated through the TA-HPG. For brevity, not all nodes and edges are shown.



different DOM environments (Cf. §4.2.2). By doing so, we switch DOM trees when testing the data flow within the affected JavaScript program, as the DOM environment can affect the execution of the program. To enable this approach, we need to provide the input differently depending on the type of the source, i.e., URL parameters, postMessages, document referrer and window name (Cf. §2.2.3). For example, in case of URL parameters, we can control them directly, and load the manipulated URL for testing. For other sources, we load a test webpage in the browser, which uses `window.open()` [154] to open the target webpage in a new window and set the window name through `window.name` API [155] or send postMessages to the opened window [3]. Alternatively, the test page can redirect to the target webpage and control the document referrer leveraging the URL of the test page. Finally, we perform manual analysis to validate the decision reported by JAW-v2 and examine the exploitability of the reported data flows.

4.2.5 Approach Exemplification

In this section, we exemplify the TA-HPG construction and analysis approach presented in Sections 4.2.2 and 4.2.3 through an example of a real vulnerability derived from `bbc.com`, highlighting the need for dynamic information provided by Foxhound^{+}.

Vulnerability. Listing 4.1 shows a real example of client-side request hijacking vulnerability derived from `bbc.com`, where the program uses attacker-controlled inputs to specify the endpoint to which an asynchronous HTTP POST request is sent to. In more detail, the code first listens for incoming postMessages (line 3), and then uses the message data to construct a URL (lines 4-12). Afterwards, it creates a closure function using the `new Function()` API [223] by generating a string of the target function call dynamically (lines 13-14). The string contains an invocation of the `httpPostRequest` function in line 17, with parameters being the constructed URL of line 12 (attacker-controlled), and sensitive data of line 8 (i.e., CSRF tokens). Subsequently, it stores the closure function as a property of the global object `c` (line 13). Finally, upon closing the session (line 22), the program uses dynamic property lookups to retreive the closure function stored in object `c` and invokes it (line 26), which in turn sends an HTTP POST request (line 19) to the attacker-controlled endpoint.

TA-HPG Construction and Traversals. The dynamic JavaScript language features used in

Table 4.3: Summary of the collected data and preprocessing steps.

Top 10K Sites	**Raw Data**	**Dedupl.**	**Top 50 Flows**
# Webpages	1,034,521	867,455	339,267
# Scripts	46.1 M	36.7 M	11.5 M
# LoC	129.8 B	104.1 B	32.4 B
# Taint Flows	43,143,773	35,209,216	21,673,167
# Req. Flows	8,024,030	7,205,914	3,318,747

Legend: Dedupl.= Page Deduplication.

Listing 4.1 present significant challenges for static analysis-based approaches like JAW [P1] to capture the aforementioned request hijacking vulnerability. For example, JAW cannot identify the invoked function and its corresponding arguments in line 26. This is due to dynamic property reads/writes on lines 13 and 26, as well as dynamic code generation using `new Function()` on line 14, which makes it difficult to create a comprehensive representation of the program.

Figure 4.2 presents the TA-HPG that JAW-v2 generates for the code in Listing 4.1, which alleviates the missing HPG edges due to the dynamic function calls. In particular, JAW-v2 uses dynamic taintflows provided by Foxhound^{+} to add (i) a call edge between the call expression node in line 26 and the function declaration node in line 17, and (ii) PDG data dependency edges from assignment expression in line 13 to call expression node in line 26 for call arguments `u` and `d`. Accordingly, a TA-HPG traversal can now start from the source node in L12, pass through L13, L26, and L17 nodes, and finally reach the sink instruction, who picks up the attacker-controlled values.

4.3 Empirical Evaluation

This section addresses the second part of RQ2.2 (Cf. §1.1.2), where we conduct the largest-to-date study to quantify the prevalence and impact of request hijacking vulnerabilities in the wild. To accomplish this, we utilized the Tranco site list downloaded on Sept. 29, 2022 (ID: N7QWW) [216], where we first selected the top 10K domains by excluding duplicate versions of websites (e.g., *google.com* and *google.co.uk*), and then instantiated JAW-v2 for each of them. We started our crawling infrastructure of §4.2.1 in Oct. 2022 by deploying 100 parallel browser instances. To ensure comprehensive coverage, we made up to three repeated attempts for each failed crawling website, followed by a detailed manual analysis. The entire data collection process spanned approximately six weeks. The rest of this section details our findings.

4.3.1 Data Collection and Processing

Table 4.3 summarizes the results of data collection and modeling steps. Starting with the 10K seed URLs, JAW-v2 obtained a grand total of 1,034,521≈1M webpages across all websites. The number of pages per site spanned from 1 to 200, averaging at 103 pages. These 1M pages contained around 46.1M scripts with over 129.8B LoC. Page de-duplication (Cf. §4.2.2) enabled us to focus on pages with unique sets of scripts and reduced the size of the dataset by about 17%, that is, out of the total 1M webpages, 867,455 pages were unique.

Considering the extensive size of the raw data and the need to analyze hundreds of thousands of webpages, we further reduced the size of our testbed by focusing our testing efforts on the top 50 pages of each site that exhibit the greatest frequency of dynamic taint flows (originating from input sources and reaching request-sending sinks), which is based on the higher probability of these pages containing vulnerabilities. In summary, the 867K unique pages contained ∼7.2M dynamic taint flows to request-sending sinks which we used for our page selection. Accordingly,

Table 4.4: Summary of client-side request hijacking vulnerabilities in top 10K sites. The table shows the total number of data flows from input sources (columns 3-6) to request sinks (Cf. Table A.1) as well as the affected webpages and sites. Rows marked with ➕ and ⊞ represent new vulnerabilitiy types proposed by our work and variants for which we also consider a new API, respectively. The table also shows the distribution of data flow paths in the TA-HPG (static, dynamic, or mixed) based on the type of edges involved in the flow, highlighting the contribution of dynamic information for vulnerability discovery.

		🔀 **Flows**						**••• Breakdown**			🌐	
	🕵 **Vulnerability**	**WURL**	**WN**	**DR**	**PM**	**Total**	**Verified**	**Dynamic**	**Mixed**	**Static**	**Pages**	**Sites**
⊞	Fg. Async Req. URL	106,218	105	1,232	46	107,601	91,688	47,631	8,037	36,020	12,908	616
⊞	Fg. Async Req. Body	76,517	428	5,209	6,564	88,718	78,240	63,308	7,442	7,490	9,510	819
➕	Fg. Window Open URL	20,574	21	76	3	20,674	16,566	49	652	15,865	8,846	365
	Fg. Location URL	4,533	300	108	8	4,949	4,079	1,157	131	2,791	2,610	324
➕	Fg. Async Req. Header	2,401	5	42	372	2,820	2,446	1,710	135	601	1,587	107
➕	Fg. WebSocket URL	5,322	32	320	807	6,482	5,520	2,865	543	2,113	1,096	56
➕	Fg. WebSocket Body	2,867	18	172	434	3,490	2,973	1,543	292	1,137	590	30
➕	Fg. Push Req. URL	592	93	0	0	685	539	0	530	9	497	25
➕	Fg. Push Req. Body	94	61	2	0	157	119	0	115	4	101	9
➕	Fg. EventSource URL	680	2	36	133	851	664	387	66	211	56	3
	Total	219,798	1,065	7,197	8,367	236,427	**202,834**	118,650	17,943	66,241	**17,805**	**961**

Legend: Fg.= Forgeable; WURL= Window URL; WN= Window Name; DR= Document Referrer; PM= postMessage.

the 867K webpages were filtered to 339,267 pages. Out of these, JAW-v2 extracted 11,544,754 scripts (32.4B LoC) and 21.6M dynamic taint flows, that we can use to enrich HPGs in order to remediate missing connections that are not discovered by static analysis. Out of these 21.6M taint flows, 3,318,747 flows contain request-sending sinks, which can indicate the presence of request hijacking vulnerabilities. In total, JAW-v2 processed an average of 34 scripts and 95K LoC per page, generating 339,267 TA-HPGs.

4.3.2 Prevalence in the Wild

After TA-HPG construction, JAW-v2 performed graph traversals for vulnerability discovery following §4.2.3. In summary, JAW-v2 identified an average of 23 request-sending sinks and 65 sources per webpage, totaling about 7.9M sinks and 22.3M sources. Among these, static analysis found a total of 236,427 potential data flows from sources to sinks, of which ~86% (i.e., 202,834) were verified following runtime experiments. These vulnerable data flows affected around 5.2% of the tested webpages (17,805 out of 339,267) and 9.6% of the sites (961 out of 10K), which is alarming. Table 4.4 presents a summary of the results.

Types of Hijacked Requests. Among the various types of requests that can be hijacked, asynchronous requests are the most widespread (85%), with over 172K instances across 905 sites. Interestingly, forged window loads are the second-most prevalent (8.2%), accounting for 16.5K flows in 365 sites. At the other extreme, hijacked push requests and EventSource occur the least often, each affecting only about 0.3% of the flows across 25 and three sites, respectively. Finally, hijacked web sockets and top-level requests demonstrated a moderate level of prevalence, corresponding to about 6% of the vulnerable data flows in total.

New Vulnerability Types. We observed that the new vulnerability types and variants listed in Table 4.2 constitute a significant fraction (i.e., 36.1%) of the request hijacks. First, the new vulnerability types account for over 14.2% (35,159) of the total 236K discovered cases. Among these, JAW-v2 verified 28,827 vulnerable data flows across 10,925 webpages and 439 sites, highlighting the widespread occurrence of the new vulnerabilities. Then, 21.9% of the request hijacks are new variants where we considered a new browser API.

Vulnerability Impact. We found that the 202K vulnerable data flows can have different security implications (Cf. Table 4.2), where each vulnerability could lead to multiple consequences through different exploitations, amplifying the potential risks. The most common consequence is client-side CSRF, for which 96% of the vulnerabilities (i.e. 196K) can be abused. However, 48.5%

of the hijackable requests can be abused for information leakage too, as the attacker can control the endpoint to which the request is sent to, and consequently steal the sensitive information contained in the request body, such as CSRF tokens, PII, push endpoint and encryption key, as we will show in §4.3.4. In comparison, the least common consequence is persistent DoS on push subscriptions that accounts for 0.2% of the total vulnerabilities. Other common consequences are client-side XSS and open redirections that affect 10.1% of the pages in total. Finally, 4.2% of the vulnerabilities could lead to cross-site connection hijack of WebSocket and EventSource.

Verification and False Positives. Given the extensive number of reported data flows by JAW-v2, we performed a semi-automatic verification as elaborated in §4.2.4.

In total, the dynamic verification module confirmed about 86% of the data flows (i.e., 202,834 out of 236,427) and eliminated a total of 33,593 FPs across 1,954 webpages and 28 sites. Notably, for the majority of the confirmed flows (i.e., 81%), the verification module successfully validated the vulnerable flow by loading the affected webpage and executing it via Playwright. However, in the remaining 19% of cases (i.e., 38,522 flows), the verifier required executing between one to 41 near-duplicate pages before confirming it. We note that the verifier tests the presence of the data flow also in near-duplicate pages in order to switch the DOM tree with one of the duplicated pages. This is aimed to determine if the same data flow could be observed across various DOM environments during page load, capturing different executions of the program (Cf. §4.2.4).

We manually confirmed and investigated the reason for false positives by focusing on a random subset. Specifically, we sampled 10 pages per each of the 28 affected sites, which included 5,032 flows in total, and manually inspected them. We observed that a large number of FPs (i.e., 3,951 or 78.5%) occur during the data flow analysis from sources to sinks, and the rest (i.e., 1,081 FPs) occur when checking if a request is triggered at page load or not (i.e., reachability analysis). The former cases happened due to presence of dynamic code evaluation constructs like `eval()` that changed the values of tainted variables, usage of prototype chain with late static binding, generator functions, and inaccurate pointer analysis for property lookups. The latter cases happened due to dynamically called functions, inaccurate pointer analysis, usage of reflection, and dynamic removal of event handlers. Accordingly, verification was critical to eliminate FPs.

Contribution of Dynamic Analysis. We observed that dynamic information plays a crucial role in identifying 67.3% of the discovered request hijacking data flows, as shown in Table 4.4. First, dynamic taint analysis detected 118.6K vulnerable data flows that were not found by the static analyzer, i.e., data flow paths containing only dynamic edges. Second, it aided static analysis in identifying 17.9K additional data flows by patching missing HPG edges necessary for vulnerability detection (i.e., mixed data flow paths). However, Table 4.3 highlights a key challenge of pure dynamic analysis: the large size of reported taint flows, the majority of which were not under attacker control (Cf. Table 4.9). Conversely, static analysis was able to detect 66.2K data flows. Therefore, a combination of dynamic and static analysis can be advantageous. Dynamic analysis enhances static analysis by supplementing HPG edges (e.g., call graph), while static analysis helps eliminate spurious taint flows that are not controllable by attackers, e.g., due to input validation.

4.3.3 Anatomy of Hijacked Requests

In this section, we examine the extent of manipulation an attacker can exert on the hijacked requests of Table 4.4, as the specific forgeable field(s) and the degree of control an attacker possesses over them may affect the potential risk and severity of vulnerabilities. We used JAW-v2 to extract the vulnerable lines of code, examined the code stack trace and semantics, and characterized the request anatomy as a binary pattern, encoding information about the type and number of request fields that could be manipulated, as well as the type of control in each

Table 4.5: Anatomy of client-side forgeable requests. The table shows 29 distinct request patterns ordered by the degree of control (descending).

⚙ **Request Fields**							**!**	🌐 **Prevalence**		
D	**P**	**B**	**Q**	**F**	**H**	**S**	**Total**	**Flows**	**Pages**	**Sites**
●	●	○	●	●	○	●	5	2,897	1,103	101
●	●	○	●	○	●	●	5	1,235	235	26
●	●	○	●	○	○	○	3	110	34	11
●	●	○	○	●	○	●	4	88	52	13
●	●	○	○	●	○	○	3	1	1	1
●	●	○	○	○	○	●	3	1,456	391	52
●	●	○	○	○	○	○	2	65	39	5
●	○	●	○	●	○	●	4	1	1	1
●	○	●	○	○	○	○	2	973	159	12
●	○	○	●	○	○	○	2	18	10	2
●	○	○	○	◐	○	●	3	8	6	1
●	○	○	○	○	○	●	2	672	118	10
◐	◐	○	●	○	●	●	5	2	1	1
◐	◐	○	○	○	○	●	3	5	4	1
◐	○	◐	○	○	○	●	3	10	2	2
◐	○	○	○	○	○	○	1	564	95	9
○	●	●	●	○	○	○	3	8	6	1
○	●	○	●	○	○	●	3	1	1	1
○	●	○	○	○	○	●	2	3	3	1
○	●	○	○	○	○	○	1	342	124	13
○	◐	○	●	○	●	○	3	3	1	1
○	◐	○	○	●	○	●	3	1	1	1
○	◐	○	○	○	○	●	2	15	1	1
○	◐	○	○	○	○	○	1	9,640	2,024	219
○	○	●	●	○	○	○	2	92	47	6
○	○	◐	○	○	○	○	1	95,601	12,187	981
○	○	○	◐	●	○	○	2	215	74	5
○	○	○	◐	○	○	○	1	88,009	9,356	747
○	○	○	○	○	●	○	1	799	400	36

Legend: D= Domain; P= Path; B= Body; Q= Query; F= Fragment; H=Headers; S=Scheme;
○ = Not Controllable (00); ◐ = Partial Control (01); ● = Full Control (10);

Table 4.6: Summary of exploitations for client-side request hijacking vulnerabilities. Rows marked with ✚ and ⊞ represent new vulnerabilitiy types and variants with a new API or exploitation, respectively.

	Vulnerability	CSRF	XSS	WS Hijack	SSE Hijack	Inf. Leak	Open Red.	DoS	Total
⊞	Forge. Aysnc Req. URL	7/6	-	-	-	12/7	-	-	19/13
⊞	Forge. Aysnc Req. Body	4/4	-	-	-	-	-	-	4/4
✚	Forge. Aysnc Req. Header	1/1	-	-	-	-	-	-	1/1
✚	Forge. Push Req. URL	-	-	-	-	2/2	-	2/2	4/4
✚	Forge. Push Req. Body	1/1	-	-	-	-	-	-	1/1
✚	Forge. EventSource URL	-	-	-	1/1	1/1	-	-	2/2
✚	Forge. WebSocket URL	-	-	2/2	-	4/2	-	-	6/4
✚	Forge. WebSocket Body	2/1	-	2/2	-	-	-	-	4/3
	Forge. Location URL	1/1	3/3	-	-	-	7/6	-	11/7
✚	Forge. Window Open URL	1/1	6/6	-	-	-	8/8	-	15/10
	Total	17/15	9/9	4/4	1/1	19/12	15/14	2/2	**67/49**

Legend: $M/N = M$ exploits across N sites.

field. As a result, we identified 29 distinct forgeable request patterns. Table 4.5 summarizes our findings.

Type of Control. Our analysis revealed that 80% of the forgeable request fields are fully controllable, allowing the attacker to overwrite their values entirely. In the remaining cases, the attacker has partial control over specific parts of the field, such as one or more parts of query parameters, hash fragment, or body, but not complete control.

Forgeable Request Field. The severity of the vulnerability can be influenced by the type of manipulable field. For example, we found that in 8,105 forgeable requests of 161 sites, the attacker can manipulate the domain, and request hijacking could be used to perform cross-origin attacks (e.g., leakage of CSRF tokens). We grouped requests in seven categories based on the specific field(s) being manipulated, where each request may fall into multiple groups. Our analysis uncovered that the most frequent types of manipulable fields are request body and query parameters, accounting for over 47% and 45% of the forgeable requests respectively. Additionally, the forgeability of domain and path fields in ∼11.8% of the requests is concerning. Finally, we observed that other request fields like headers, hash fragment and scheme are forgeable in about 5.7% of the cases.

Degree of Manipulation. We found that the number of concurrently manipulable request fields varies from one to five out of a total of seven forgeable fields. For example, for 2,897 forgeable requests from 101 sites, the attacker has full control over all URL fields but lacks control over request headers and body. In contrast, in 95K requests on 981 sites and 88K requests on 747 sites, the attacker can manipulate only the request body and query parameters, respectively. We observed that in the majority of the hijacked requests (i.e., 97%), only one or two fields can be manipulated. However, such ad-hoc manipulation capability can still lead to severe consequences (Cf. §4.3.4).

4.3.4 Exploitations

We manually examined the exploitability of the identified vulnerabilities by a web attacker. To ensure comprehensiveness, we aimed to maximize the coverage of our testing across various sites. We randomly selected two vulnerable pages from each of the 961 affected sites. Given the high number of vulnerable data flows within webpages, we used our analysis of §4.3.3 to prioritize testing efforts by focusing first on requests with a higher degree of manipulation across various types of client-side requests. Then, we confirm the forgeability of requests and look for their use in attacks that we presented in §4.1. For each attack scenario, we conducted specific checks. For example, we looked for server-side endpoints that could lead to security-sensitive state changes (e.g., modifying user settings) for client-side CSRF. For information leakage, we examined the request body for the presence of sensitive data like PII, authorization keys, and CSRF tokens. Furthermore, For WebSocket and EventSource, we check whether we can establish arbitrary connections to attacker-controlled endpoints. Finally, for open redirect and client-side XSS attacks, we assessed the susceptibility of top-level requests to arbitrary redirections and improper validation of `javascript` URIs, respectively. In doing so, we limited our tests exclusively to user accounts that we created on those sites, and excluded testing requests and functionalities where we could not control the impact (e.g., publicly accessible content).

Table 4.6 summarizes the attacks we uncovered during our investigation. In total, we created 67 proof-of-concept exploits across 49 websites, with far-reaching consequences like CSRF, client-side XSS, open redirections and leakage of sensitive information across various popular platforms and functionalities. Notably, we discovered an account takeover exploit in the Starz movie streaming service, user VM deletion in Microsoft Azure, arbitrary redirection in Google DoubleClick and VK, manipulation of account settings in DW and BBC, tampering of job applications in Indeed, data exfiltration through WebSocket and EventSource hijacks in JustWatch and Forbes, CSRF on `PushManager` subscriptions in Reddit, persistent client-side DoS on push notifications in Yoox shopping website, and finally client-side XSS in TP-Link, to name only a few examples. Among these, a total of 33 exploits across 24 sites belong to new vulnerability types presented in our work.

4.4 Defenses

This section addresses RQ3.3 of §1.1.2, where we review and assess the adoption and efficacy of existing countermeasures against client-side request hijacking vulnerabilities. We systematically surveyed academic literature (i.e., [17, 21, 138, 227, 66, 38, P1, P4, 204, 107, 32, 224, 70, 16, 72, 225, 226, 228, 71, 19]), W3C specifications [229], and OWASP CheatSheet Series [69], looking for classical anti-CSRF countermeasures and those defenses that can mitigate client-side request hijacking. Table 4.7 summarizes our findings. In total, we identified 10 distinct request forgery defenses, that we grouped into two broad categories based on the party that enforces them (i.e., web application or the browser). For each defense, the table represents whether the defense is effective against client-side request hijacks, there is built-in browser support to enforce it, it is enabled-by-default, whether it requires correct configuration (when offered built-in by the browser), and finally whether it requires custom implementation by web application developers. The rest of this section discusses adoption and efficacy of each defense.

Traditional Mechanisms. CSRF attacks can be mitigated by employing various countermeasures, such as anti-forgery tokens [17, 70, 69, 16, 71, 19], CORS preflight requests [70], `Origin/Referer` [17, 231, 69, 20] header checks, and SameSite cookies [P4]. Our measurement in Table 4.7 shows that these countermeasures are well adopted. For example, we found that 130,359 of the 202K forgeable requests (Cf. Table 4.4) include a token in the request body or header. Among these, 116,002 cases featured a token name containing 'csrf' or 'xsrf', indicating it was an anti-forgery token. Then, when looking at JavaScript code, we observed that developers explicitly included `Origin/Referer` headers in 42,310 same-site requests. Finally, we observed that 3,751 vulnerable pages (out of 17.8K) use SameSite cookies with `Lax` or `Strict` policies.

While these defenses are necessary to prevent classical request forgery attacks (assuming correct implemention), they are not sufficient to prevent client-side hijack of requests, because JavaScript programs and web browsers include these tokens, headers, and cookies in same-site requests.

Table 4.7: Summary of existing defenses and their protective coverage against client-side hijacks. The table shows the adoption rate of the various defense mechanisms in the wild. For rows marked with *, the adoption rate only reflects the explicit inclusion of headers/tokens in the client-side code.

Category	Defense	🛡	◎	✓	⚙	</>	References	# Pages	# Sites
Custom (Application)	Input Validation	●	◐	○	○	●	[230, P1, 107, 32, 3, 56]	125,738	7,021
	CSRF Tokens*	○	○	○	○	●	[17, 21, 70, 69, 20, 71, 19]	32,925	7,692
	Fetch MetaData*	◐	●	○	●	●	[228]	13,873	910
	Origin/Ref. Headers*	○	●	○	○	●	[17, P4, 70, 69]	9,922	1,745
Policy-based (Browser)	Cross-Origin Resource Sharing	○	●	●	●	○	[162]	284,984	8,741
	SameSite Cookies	○	●	●	●	○	[28, 138, P4, 72]	69,865	5,621
	Content Security Policy	◐	●	○	●	○	[204, 8, 109, 27]	25,799	4,616
	Cross-Origin Opener Policy	◐	●	○	●	○	[73, 228]	6,581	231
	Cross-Origin Embedder Policy	◐	●	○	●	○	[74]	3,314	96

Legend: 🛡= Effective; ◎= Built-in Browser Support; ✓= Enabled-by-Default; ⚙= Require Configuration; </>= Require Implementation; ○ = Not Applicable; ◐ = Partially Applicable; ● = Fully Applicable; *= Server-side enforced.

Table 4.8: Types of input validation checks in vulnerable data flows.

Check	Instances	Flows	Pages	Sites
No Check	S	95,321	8,876	709
Substring Search	S.indexOf(‘benign.com’) > 0	62,495	3,950	285
	S.startswith(‘benign.com’)	11,448	821	83
	S.includes(‘benign.com’)	2,024	145	32
Not Null	typeof S !== ‘undefined’ && S!== null	32,002	2,616	194
Length	S && S.length > 0	13,995	1,023	83
Empty String	S !== ‘’	6,179	638	156
Comparison of Forgeable Params	QUERY(Q, S)=== window.name	4,776	445	65
	QUERY(Q, S)=== loc.hash.substr(i, j)	556	39	3
	postMessage(S) === loc.hash.substr(i, j)	102	10	1
URL Fields Check	PATH(S) == ‘index.php’	1,199	92	10
	QUERY(Q, S) === ‘benign’	402	33	5
Faulty Conditionals (Always True)	if (S === ‘b1.com’ || ‘b2.com’)	130	11	3
	!! ‘benign.com’ == !! S	629	40	3
	S !== undefined + (S === ‘benign.com’)	14	6	1
	intersection([‘b1.com’, ‘b2.com’], [S]) !== []	21	5	2
	S.length $\leq$ Math.min() + CONST	5	2	1

Legend: S= Source; QUERY(Q, URL)= query parameter Q in URL PATH(URL) = URL path.

Input Validation. Robust input validation can ensure data integrity and reliability by requiring untrusted inputs to conform to specific, expected formats [230], preventing malicious inputs reaching request-sending instructions. Accordingly, we identified and analyzed secure and insecure input validation patterns and practices that is employed by websites in the wild against request hijacking attacks as described below.

First, to identify insecure input validation code patterns, we analyzed vulnerable data flows discovered in §4.3.2, and extracted the underlying reason why the flow was marked as vulnerable, focusing on the presence of insufficient, missing or logically flawed input validation checks. Table 4.8 summarizes our findings, where we grouped the checks into eight different categories. Our analysis uncovers that ∼47% of vulnerable data flows do not have any input validation checks, suggesting that developers are largely unaware of risks associated with controlling client-side requests. Furthermore, over 13.8% of the cases rely solely on a variety of trivial checks, such as length and type checks, and 24.9% use string operations to search for existence of trusted domains in URLs or check different URL fields, which is insufficient, e.g., the check for presence of `benign.com` can be trivially bypassed if the attacker uses the payload `benign.com.evil.com`. Similarly, checks that only test partial URL fields, such as

Table 4.9: Summary of program behaviours that can eliminate unique client-side request hijacking vulnerabilities.

Property	**Instances**	**Flows**	**Pages**	**Sites**
Infeasible Source Manipu.	SP is URL domain	1,152,266	116,441	3,249
	SP is URL path	911,897	95,269	2,657
Reassignment to Source	SP = constant	627,460	68,251	3,358
	SP: fragment string replace	21,709	3,067	1,502
	SP: fragment object assign	4,092	666	434
Whitelist / Equality Check	SP === constant	367,024	49,089	2,294
	SP: postMessage origin check	40,185	7,634	965
	SP includes a set of constants	15,032	2,541	367
	arrayConstants.includes(SP)	10,022	899	610
Duplicate Function Calls	SP flow executed $\geq$1 times	8,743	1,228	202
Length Check	Length(SP) === 1	3,560	1,155	501
Type Check	typeof SP === "number"	12,121	2,001	1,328
	typeof SP === "boolean"	1,667	1,001	542
	SP instanceof Date	789	300	91
	SP is JSON && valid(SP)	443	235	55
Benign Control / Taint	SP taints request fragment only	97,528	17,802	1,029
	SP taints request scheme only	44,209	10,512	836

Legend: SP= Source Parameter.

path or query parameters are insufficient, because attackers may be able to forge the request domains, or overwrite query parameters with parameter pollution [232]. We observed that a different group of data flows (11.1%) apply a combination of these checks simultaneously.

Then, about 2.7% of the data flows contain validation checks that compare two different attacker-controlled values with one another, e.g., a query parameter value, used to generate an asynchronous request, is compared with `window.name`, suggesting that developers treat `window` properties as trusted values that can be safely used in sensitive operations. Finally, less than 0.4% of the input validation checks exhibit logical flaws, where the tested condition always evaluates to true, which could indicate a potential gap between the semantics of the JavaScript language and the developers' comprehension. Also, we examined the input validation implemented on various data flows within the same webpage and across different pages of a site, and we observed at least two distinct types of input validation in 699 pages and 412 sites, respectively, which may suggest the presence of multiple developers' implementations and differences in their approaches to input validation.

Then, we also examine secure patterns that can hinder request hijacking, both intentionally and unintentionally. Specifically, out of the ~3.3M taint flows that Foxhound$^+$ discovered (Cf. Table 4.3), JAW-v2 marks only ~118K of them as vulnerable, and discards the rest (~3.2M) due to a variety of (preventive) program behaviours, e.g., validity checks, duplicate executions of the same data flow, and re-assignment of constant values to sources like URL fragment. We used static analysis to examine more closely the reasons why these 3.2M requests were not vulnerable, and we grouped them into seven categories (Cf. Table 4.9).

In total, JAW-v2 identified 1,104,104 data flows across 125,738 webpages and 7,021 sites that implement robust input validation against request hijacking. We found that overwriting attacker-controlled sources with variable assignments and strict equality / whitelist comparisons are the most common type of input validation, which prevents request hijacking, with a total of 653K and 432K instances across 3,935 and 2,824 sites, respectively. Then, contrary to these intentional checks, we found that other program behaviours may prevent request hijacking too. For example, the most frequent reason for the non-vulnerability of a taint flow was its sole reliance on the domain or path of the webpage to generate outgoing requests, because modifying the domain or path of the top-level URL by the attacker would result in the victim accessing a different webpage altogether.

Content Security Policy (CSP). CSP [109, 27] can limit the impact of request hijacking when attackers can forge the URL of requests. In these cases, CSP `connect-src` directive [163] can be used to constrain endpoints for asynchronous requests, EventSource and WebSockets to trusted domains, preventing sensitive data exfiltration to other domains. We found that a correct configuration of CSP could mitigate information leakage and XSS exploitations in 58.7% of the request hijacking data flows. However, we observed that only 7.6% of the webpages in our dataset deploy a CSP using this directive, including 1,265 pages with vulnerable data flows. While CSP can mitigate information leakage, it does not prevent hijacking requests for CSRF attacks (i.e., same-site request endpoints, or forging request body).

Cross-Origin Opener Policy (COOP). When attackers use `window.open()` to open vulnerable target pages in a new window, COOP [73] can be used to isolate the browsing context to same-origin documents. For example, if an honest, cross-origin page with COOP is opened in a new window, the malicious opening page will not have a reference to it, preventing attackers to set the window name, or send postMessages to the new window, which in turn prevents the forgery of requests generated by these inputs. We found that about 7% of the request hijacking data flows could be mitigated by COOP, as they rely on window name, document referrer and postMessages to provide program inputs. However, we observed that only ∼1.9% of webpages in the wild implement COOP, and, strikingly, none of the webpages exhibiting request hijacking data flows had adopted this policy, calling for increased awareness about COOP.

Cross-Origin Embedder Policy (COEP). COEP [74] controls embedded cross-origin resources in a webpage. Developers can use the COEP `require-corp` policy to restrict fetched resources to either the same origin or a set of explicitly marked cross-origin resources. As such, COEP can constrain the `fetch()` API to trusted domains, mitigating the impact of 5.3% of the total request hijacks. We observed that only ∼1% of the webpages in our dataset use the `require-corp` policy, including 141 pages with vulnerable data flows across 32 sites.

Fetch MetaData. These are a series of HTTP request headers [233, 228] that send additional provenance meta data about the request, such as the context it originated from. Websites can use this information to implement policies that block potentially malicious requests. While Fetch MetaData headers are automatically included in client-side requests by JavaScript programs, they can still restrict exploitations. For example, websites can use the `Sec-Fetch-Mode` header with the `navigate` option to restrict top-level requests exclusively for page navigation, and block request hijacking attacks that trigger state changes [P4]. We observed that these headers are present in 67,221 requests across ∼9% of websites including 85 vulnerable sites.

4.5 Summary

Request forgery attacks are among the oldest threats to Web applications, traditionally caused by server-side confused deputy vulnerabilities. However, recent advancements in client-side technologies have introduced more subtle variants of request forgery, where attackers no longer rely on confused deputy flaws but instead exploit input validation issues in client-side programs to hijack outgoing requests. Unfortunately, we still lack a comprehensive assessment of these new client-side variants, including their prevalence, impact, and countermeasures, and in this thesis, we embark on one of the initial evaluations of the state of client-side request hijacking on the Web platform.

Starting with a comprehensive review of browser API capabilities and Web specifications, we systematize request hijacking vulnerabilities and the resulting attacks, identifying 10 distinct vulnerability variants, including seven new ones. Then, we use our systematization to design and implement JAW-v2, a static-dynamic tool that detects data flows from attacker-controllable inputs to request-sending instructions. We instantiate JAW-v2 on the top of the Tranco top 10K sites, performing, to our knowledge, the first investigation into the prevalence of request hijacking flaws in the wild.

Our study uncovers that request hijacking vulnerabilities are ubiquitous, affecting 9.6% of the top 10K sites. We demonstrate the impact of these vulnerabilities by constructing 67 proof-of-concept exploits across 49 sites, making it possible to mount arbitrary code execution, information leakage, open redirections and CSRF also against popular websites like Microsoft Azure, Starz, Reddit, and Indeed. Finally, we review and evaluate the adoption and efficacy of existing countermeasures against client-side request hijacking attacks, including browser-based solutions like CSP, COOP and COEP, and input validation. Our systematic analysis shows that because request hijacking can be exploited in a variety of ways, existing countermeasures can typically only mitigate a fraction of the attacks. We present eight insecure input validation patterns which developers should avoid, and seven behaviours which provide successful mitigation.

5
Understanding DOM Clobbering Attacks and Defenses

In this chapter, we extend our security testing framework, JAW, to study the threat posed by DOM Clobbering attacks, addressing RQ3 of §1.1. As presented in §2.2.4, DOM Clobbering is a type of HTML-only injection attack, where attackers confuse a web application by injecting HTML elements whose `id` or `name` attribute matches the name of security-sensitive variables or built-in browser APIs, such as variables used for fetching remote content (e.g., script src), and overshadow their value.

DOM Clobbering vulnerabilities have been known for over a decade, with the first instance identified in 2010 [83] where an `iframe` named `self` allowed attackers to overwrite the top window location of webpages containing framebusting code, i.e., assignments such as `top.location = self.location`. Since then, security researchers have identified new, more subtle attack variants, combining pairs of HTML tags (e.g., [10, 26]) or browser-specific markups and attributes (e.g., [84, 87, 85, 86]), and clobbering not only variables, but also deep object properties (e.g., [88, 234, 9]), nested window proxies (e.g., [89, 9]) and loops (e.g., [9]). When looking at the possible combinations of tags, attributes, code features, and runtime behaviors, prior works have merely scratched the attack surface, and, to date, we still miss a systematic and comprehensive exploration of this threat.

Recently, DOM Clobbering vulnerabilities in Gmail [88] and Google Analytics [235, 91] revamped new discussions about defenses, such as proposing to switch off named property accesses for DOM elements at the browser level (see, e.g., [90, 91, 236]), which has been dismissed since, according to Google Chrome telemetry data, about 10.5% of the pages in 2021 use named property accesses to implement functionalities that could otherwise break [237]. To date, the burden of protecting from DOM Clobbering attacks is solely on developers' hands, who can use existing countermeasures such as HTML sanitizers tailored to protect against DOM Clobbering, e.g., DOMPurify [78], or mitigate the risk of code execution via Content Security Policy (CSP) [238, 239, 92]. Unfortunately, DOMPurify protects only from specific DOM Clobbering cases, whereas CSP cannot prevent the execution of already-present code that reacts to markup injections, suggesting that existing countermeasures may be incomplete or even insufficient. As a last resort, developers can develop their own defenses, requiring a deep understanding of the main threat and its variants, which, unfortunately, may not be the case. For example, as witnessed by recent DOM Clobbering vulnerabilities discovered in HTML sanitizers, e.g., DOMPurify [85] and HTML Janitor [234], developers may still be largely unaware of the risk posed by DOM Clobbering vulnerabilities.

In this chapter, we conduct the first comprehensive and systematic study of DOM Clobbering, covering vulnerability, attack techniques, detection, prevalence, impact, and defenses. First, we propose a systematic technique to identify DOM Clobbering markups and test browsers automatically, identifying 148 previously-unknown ones, 30,803 new variants, and 114 new browser APIs that can be clobbered in at least one browser (§5.1). Then, we present JAW-v3, an automated detection tool for DOM Clobbering that uncovered 9,467 DOM Clobbering vulnerabilities, affecting 9.8% of the Tranco top 5K sites, of which 44 that we manually confirmed to be exploitable, including popular sites like GitHub, Fandom, Vimeo, Trello, TripAdvisor, and AliExpress (§5.2). Finally, we evaluate the robustness of 29 client-side and server-side HTML sanitizers and CSP, showing that 55% of sanitizers are vulnerable and 85% of the DOM Clobbering vulnerabilities cannot be mitigated by CSP. Also, we review existing countermeasures, analyze common mistakes of the 491 vulnerable sites, and distill a list of recommendations and secure coding patterns (§5.3).

5.1 Attack Techniques

The first part of this paper addresses RQ3.1 of §1.1.3, investigating the different ways DOM Clobbering markups can manipulate JavaScript variables, object properties, and native APIs.

Before presenting our findings (§5.1.2), we describe the methodology we followed to answer this RQ (§5.1.1).

5.1.1 Methodology

Our methodology comprises two main steps. First, we review existing works on DOM Clobbering attacks, looking for the various techniques to generate markups and at the browser specifications causing the overrides. Then, we apply the information gathered to generate markups exhaustively and thoroughly test browsers.

5.1.1.1 Systematization of Known Instances

As the first step, we systematically reviewed the existing literature on DOM Clobbering attack markups, i.e., the academic literature [78, 236, 8, 83], HackerOne vulnerability reports [240], the CVE database [241], Bugzilla bug reports [84], and non-academic resources (see, i.e., [88, 242, 89, 25, 10, 9, 26, 243, 85]). Then, for each discovered DOM Clobbering instance, we extracted the HTML tags, attributes, the clobbered target (e.g., variable or `window`/`document` property), the object type of the clobbered target (e.g., `HTMLElement` or `WindowProxy`), and tags relation (i.e., child, `srcdoc`, or sibling). Then, we looked for the corresponding browser specification rules that explain the reason why the clobbering instance works. When the rule defines other variants of the clobbering instance, we add them to the list of the instances. Accordingly, we reviewed the HTML and DOM specifications [244, 116], and GitHub issues in the specifications' repositories, i.e., W3C permissions policy [90], WICG document policy [91, 87], and WHATWG HTML and DOM standard repositories [86, 245]. Finally, we group instances together based on their similarity, i.e., tags, attributes, target, and the type of the value it refers to. Table 5.1 shows the result of our systematization.

5.1.1.2 Markup Generation and Browser Testing

Starting from our systematization, we derived a list of rules for generating DOM Clobbering markups, covering all HTML tags, attributes, tags' relations, and attack targets (i.e., a variable, an object property, or a native browser API). First, we generated candidate HTML markups for a target 'x' using all the 142 valid HTML tags, including a custom tag (e.g., `mytag`), and all the 244 valid HTML attributes, including a custom attribute. For each tag, we set the value of each attribute to 'x' and add the JavaScript code that checks whether the markup clobbers the target 'x'. Then, we generated markups for object properties '$x.y$' and '$x.x$' combining all pairs of the 142 HTML tags considering three relations: sibling tags, parent-child tags, and the `srcdoc` attribute value. The experiments with a single tag showed that only `name` and `id` attributes create named properties. Accordingly, to reduce the number of test cases to a testable size, the generation of markups for object properties did not consider combinations of all HTML attributes, but only those of the `name` and `id`, e.g., `id=x`, or `id=x, name=y`.

After generating all markups, we put each of them in a test webpage, along with a JavaScript code that verifies if the target is clobbered. Then, we instantiate each browser and visit the test pages automatically. For web browsers, we used BrowserStack [246] to programmatically control browser versions, names, and their execution life-cycle in a fully automatic fashion. We evaluated (the latest versions of) all mobile and desktop browsers available in BrowserStack (i.e., 16 browsers), and additionally tested the Tor Browser for the sake of completeness. Finally, for Safari, we considered three different versions that correspond to the three recent macOS operating systems as Safari cannot be upgraded standalone [247]. In total, we evaluated 19 browsers.

Overall, our generation algorithm produced 3,906,136 candidate test markups, of which 34,648 are for targets 'x', i.e., variables or native APIs, and the rest are for object properties '$x.y$'

Table 5.1: Overview of known DOM Clobbering markups grouped by their corresponding rules in the HTML [116] and DOM [244] specifications.

Rule(s)	Target	Ref. Type	Tag 1	Tag 2	Attribute 1	Attribute 2	Relation	Total	References
Named Access Window									
R1	win.x, x	WindowProxy	iframe	-	n=x	-	-	1	[90, 167, 243]
	win.x, x	HTMLElement	TS1, TS2	-	n=x	-	-	5	[90, 167, 26, 85]
	win.x, x	HTMLElement	any	-	id=x	-	-	141	[88, 90, 10, 9, 167]
DOM Tree Accessors									
R2	doc.x	WindowProxy	iframe	-	n=x	-	-	1	[242, 9, 168]
	doc.x	HTMLElement	TS1, TS2	-	n=x	-	-	5	[168, 26]
	doc.x	HTMLElement	object	-	id=x	-	-	1	[168]
	doc.x	HTMLElement	img, image	-	id=x, n=any	-	-	2	[168, 26, 85]
Form Parent-Child									
R3, R1, R2	win.x.y, doc.x.y	HTMLElement	form	TS2, TS3	id=x ‖ n=x	id=y ‖ n=y	child	36	[234, 9, 26, 85, 86]
Nested Window Proxy									
R4, R1, R2	win.x.y, doc.x.y	WindowProxy	iframe	iframe	n=x	n=y	`srcdoc` attr.	1	[242, 89, 9]
HTMLCollection									
R5, R1, R2	win.x.x	HTMLCollection	any	any	id=x	id=x	child, sibling	141	[88, 244, 10]
	doc.x.x	HTMLCollection	TS2	TS2	id=x	id=x	child, sibling	3	[244, 10, 168]
	win.x.y	HTMLCollection	any	any	id=x, n=y	id=x	child, sibling	141	[88, 242, 25, 244, 10, 9]
	doc.x.y	HTMLCollection	TS2	TS2	id=x, n=y	id=x	child, sibling	3	[244, 10, 168]

Legend: R1= Named Access on Window Rule ([116] §7.3.3); R2= DOM Tree Accessors Rule ([116] §3.1.5); R3= Form Element Rule ([116] §4.10.3); R4= Iframe `srcdoc` Rule ([116] §4.8.5); R5= HTMLCollection Rule ([244] §4.2.10.2); win=window; doc=document; n=name; TS1=form, embed; TS2= object, img; image; TS3=button, fieldset, input, output, select, textarea.

and '$x.x$'. When testing variables, we replace the target 'x' with the variable name generating in total 34,648 test cases for variables. When testing native DOM APIs, we replace the target 'x' with the API function or property name (e.g., the `cookie` property of `document`), obtaining 34,648 test cases per API function. As of October 2021, the total number of DOM API objects is 581 [132], of which 347 are `window` APIs (i.e., 291 properties and 56 methods) [248], and 234 APIs are for the `document` object (i.e., 178 properties and 56 methods) [131]. In total, we generated 20,130,488 test cases for native APIs.

5.1.2 Results

This section presents the results of our literature review and browser testing.

5.1.2.1 Systematization of Known Instances

Table 5.1 summarizes the DOM Clobbering markups[1]. Our review identified 481 DOM Clobbering instances that we grouped into 13 classes based on their structural similarity. Each instance shows how a specific HTML markup (e.g., `<a id=x>`) can clobber a specific target, i.e., variable (e.g., x) or object property (e.g., window.x), and replaced it with a JavaScript object (e.g., x is shadowed by an `HTMLAnchorElement`). For each class, the table shows the clobbered target, the HTML code that can overwrite it, and the object type stored in the target. Also, the review of the HTML and DOM specifications resulted in the identification of five rules that instruct the browser to store the reference type in the target, which is mapped to each known DOM Clobbering instance. The rules are Named Access on Window ([116] §7.3.3), DOM Tree Accessors ([116] §3.1.5), Form Element ([116] §4.10.3), Iframe `srcdoc` attribute ([116] §4.8.5), and HTMLCollection ([244] §4.2.10.2), which we labeled as R1 to R5, respectively. The rest of this section details each group of clobbering markups and the rules abused by them.

Named Access Window. These group of markups leverage a single HTML element whose `id` or `name` is set to a target variable 'x', clobbering window.x due to browsers' compliance

[1] An interactive version of the markups available on `https://domclob.xyz/domc_markups/list`

with the Named Access on the Window Object rule (R1) [167]. We reviewed this rule in §2.2.4. Note that we use window.x and 'x' interchangeably because all global variables belong to the global `window` object by default.

DOM Tree Accessors. The markups of this group can shadow `document` properties because browsers comply with the DOM Tree Accessors rule (R2) [168], which instructs browsers how to retrieve properties of the `document` object (e.g., DOM elements). Similarly to the previous group, these markups use a single named HTML element (e.g., `object`, or `embed`) to clobber a property 'x' of the `document`.

Form Parent-Child Relationship. These markups clobber properties '$X.y$' where 'X' can be any of 'x', window.x, and document.x. First, they exploit either the rules R1 or R2 to clobber the base object 'X'. Then, they use the Form Element rule (R3) to clobber property 'y' of object 'X', i.e., the form elements' parent-child relationships where the browser creates a property of the second element for the first element's accessor variable [9]. DOM Clobbering code that rely on this technique comprise a `form` tag and a child (e.g., an `input`) named 'x' and 'y', respectively.

Nested Window Proxies. These markups use the Iframe `srcdoc` rule (R4) to create nested window proxies that are named with 'x' and 'y', respectively. Similarly to the previous group of markups, it uses the rule R1 or R2 to clobber the base object. Then, the stacked iframes enable attackers to exploit frame navigation features to clobber object properties like '$x.y$' [89, 9].

HTMLCollection. The last gour groups of markups rely on a different rule known as HTMLCollection (R5). Specifically, when two or more elements have the same `id` in the DOM tree, browsers create an array-like object called `HTMLCollection` [10, 249], which contains all elements with the same id. Elements inside HTMLCollections can be accessed by (i) their index in the collection and (ii) their `id` and `name`, enabling attackers to abuse R5 to clobber arrays [9] and loop elements (e.g., 'x' and '$x[i]$') as well as object properties like '$x.x$' and '$x.y$' [88]. Similarly to the previous techniques, rules R1-2 can be combined with R5 to clobber nested object properties like window.$x.y$.

5.1.2.2 Clobbering Variables and Object Properties

Our browser testing experiments uncovered 31,432 distinct DOM Clobbering markups that work in at least one browser, as summarized in §5.1.2.1[2], from which 145 clobber a variable 'x', and the remaining 31,287 clobber '$x.y$' and '$x.x$'.

Post-processing of Results. As the manual review of 31K individual instances is infeasible, we group instances by similar features. We start with preliminary groups based on the set of browsers they work in and the target they clobber. Then, we look at the structural features, i.e., tag1, tag2, attribute1, attribute2, and relationship, and we merge two groups when all the structural features but one are the same. Accordingly, we reduced the 31K instances to 74 classes, as shown in §5.1.2.1, and map each class to our systematization of known instances. In summary, out of the 74 classes, 10 classes rely on the Window Named Access, four classes on DOM Accessors, 13 classes on the Parent-Child Relationship, four classes on Nested Window Proxies, and finally 43 classes leverage HTMLCollections.

Findings. By comparing the 74 DOM Clobbering classes in §5.1.2.1 with the 13 previously identified classes in Table 5.1, we discovered that the 31,432 DOM Clobbering markups include 148 new instances, 481 previously known ones, and 30,803 variants of the known ones, which rely on one of the five DOM Clobbering techniques of §5.1.2.1.

The variants derive from markups that are already known for DOM Clobbering according to Table 5.1, but now have one or more additional attributes, or are permuted *in part* with a different HTML tag. For example, HTMLCollections clobbering `window` properties may be

[2]Interactive version: `https://soheilkhodayari.github.io/DOMClobbering/domc_markups/list`

Table 5.2: Overview of DOM Clobbering markups. Rows marked with ⊕ are classes that contain new DOM Clobbering instances. For all rows, clobbering window.x also implies clobbering the variable x. Browsers with similar behaviours are grouped with the same color. The table highlights a total of 10 distinct groups of browser behaviours with respect to DOM Clobbering. **Legend:** w=window; d=document; Rel.= Relationship; T_i= Tag set in Table A.3 of Appendix A.2; n= name; ch= child; sib= sibling; (&p)= optional property p; − = minus operator; TB = Tor Browser; SI= Samsung Internet; UC= UC Browser; ● = clobbered; ○ = clobbering fails.

								Chrome			Firefox			Opera			Edge			Safari				TB	SI	UC
	HTML Markup							95.0.4638	96.0	92.0.4515	94.1.2	95.0	39.0	65.2.3381	82.0.4227	3.2.3	95.0.1020	96.0.1054	95.0.1020	15.1	14.1	13.1	14.7.1	11.0.1	15.0.6	13.3.8
Clobbered	Tag 1	Tag 2	Attribute 1	Attribute 2	Rel.	Total	New	🤖	🖥	🍎	🤖	🖥	🍎	🤖	🖥	🍎	🤖	🖥	🍎	🖥	🖥	🖥	🍎	🖥	🤖	🤖
Named Access Window																										
win.x	T2	-	id=x	-	-	106		●	●	○	●	●	○	●	●	○	●	●	○	○	○	○	○	●	●	●
⊕ win.x	customtag,iframe,T5	-	id=x	-	-	8	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
win.x	T6,bdi,bdo,big	-	id=x	-	-	6		●	●	○	●	●	○	●	●	○	●	●	○	○	○	●	○	●	●	●
win.x	T4,embed,form	-	n=x	-	-	5		●	●	○	●	●	○	●	●	○	●	●	○	○	○	○	○	●	●	●
win.x	video,wbr,xmp	-	id=x	-	-	3		●	●	○	●	●	○	●	●	○	●	●	○	●	○	○	○	●	●	●
win.x	aside,audio,b	-	id=x	-	-	3		●	●	○	●	●	○	●	●	○	●	●	○	○	●	●	○	●	●	●
⊕ win.x	applet	-	n=x	-	-	1	1	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
win.x	iframe	-	n=x	-	-	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
win.x	base	-	id=x	-	-	1		●	●	○	●	●	●	●	●	○	●	●	○	○	●	●	○	●	●	●
win.x	article	-	id=x	-	-	1		●	●	○	●	●	●	●	●	○	●	●	○	○	●	●	●	●	●	●
DOM Tree Accessors																										
doc.x	T4,embed,form	-	n=x	-	-	5		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕ doc.x	applet	-	id=x ‖ n=x		-	2	2	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕ doc.x	iframe	-	id=x ‖ n=x	-	-	2	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
doc.x	object	-	id=x	-	-	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Form Parent-child																										
win.x.y	form	T3,T4 − fieldset	id=x ‖ n=x (& id=y)	id=y ‖ n=y	ch	64		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
doc.x.y	form	T3,T4	n=x (& id=y)	id=y (& n=x ‖ n=y)	ch	36		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
win.x.y	form	T3,T4	id=x (& n=y)	id=x & n=y	ch	18		●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	●	●	●
win.x.y	form	T3,T4,embed	n=x	id=y & n=x	ch	10		●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	●	●	●
doc.x.y	form	T3,T4,embed, form	n=x	id=y & n=x	ch	10		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
win.x.x	form	T3,T4	id=x	id=x & n=y	ch	9		○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
win.x.y	form	button	id=x ‖ n=x (& id=y)	id=y ‖ n=y	ch	8		●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○	●	●	●
win.x.x	form	T3	n=x	id=y & n=x	ch	6		●	●	●	●	●	○	●	●	○	●	●	●	○	●	○	○	●	●	●
doc.x.x	form	T3,T4	n=x	id=y & n=x	ch	6		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕ doc/win.x.x	form	T4,embed	n=x	id=y & n=x	ch	4	1	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
⊕ doc.x.y	form	iframe	n=x	id=y & n=x	ch	1	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕ doc/win.x.y	form	T4,embed	id=y & n=x	id=y & n=x	ch	4	1	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
⊕ win.x.y	form	applet	n=x	id=y & n=x	ch	1	1	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
Nested Window Proxy																										
doc.x.x	iframe	iframe	n=x	id=y & n=x	srcdoc	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕ doc.x.y	iframe	iframe	n=x	id=y ‖ n=y	srcdoc	2	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
win.x.x	iframe	iframe	n=x	id=y & n=x	srcdoc	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕ win.x.y	iframe	iframe	n=x	id=y ‖ n=y	srcdoc	2	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
HTMLCollection																										
win.x.y	T1,svg,customtag	T1,plaintext	id=x	id=x & n=y	sib	787		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
win.x.y	T1,customtag − T7,iframe	T1,plaintext	id=x	id=x & n=y	ch	774		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
win.x.y	abbr,dl,dt	T13	id=x	id=x & n=y	ch, sib	274		●	●	●	○	○	●	●	●	○	●	●	●	●	●	○	●	○	●	●
win.x.y	abbr,dl,image,img	T8,T12,T20	id=x	id=x & n=y	ch, sib	392		●	●	●	○	○	●	●	●	○	●	●	●	●	●	●	●	○	●	●
win.x.y	T18	T13,T14	id=x	id=x & n=y	ch, sib	7,480		●	●	○	○	○	●	●	●	●	●	●	●	●	●	○	●	○	●	●
win.x.y	address,dir,dt	T15	id=x	id=x & n=y	ch, sib	338		●	●	●	○	○	●	●	●	●	●	●	●	●	●	○	●	○	●	●
doc.x.y	applet	T4,applet	id=x	id=x & n=y	ch	4		○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕ doc.x.y	T4,applet,embed form,iframe	T4,applet,embed form,iframe	n=x	id=y & n=x &	sib	13	13	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕ doc.x.y	applet,embed form,image,img	T4,applet,embed T4,applet,embed	n=x	id=y & n=x	ch	11	11	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
doc.x.y	applet,object	T4,applet	id=x	id=x & n=y	sib	5		○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
win.x.y	dir,div,dt,element	T16	id=x	id=x & n=y	ch, sib	252		●	●	●	○	○	●	●	●	●	●	●	●	○	●	○	●	○	●	●
win.x.y	div	T17	id=x	id=x & n=y	ch, sib	66		●	●	●	○	○	●	●	●	●	●	●	●	○	●	●	●	○	●	●
win.x.y	div,dl	T12	id=x	id=x & n=y	ch, sib	186		●	●	●	○	○	○	●	●	●	●	●	●	●	●	●	●	○	●	●
win.x.y	element,em,embed,fieldset	T1,plaintext-iframe	id=x	id=x & n=y	ch, sib	876		●	●	●	○	○	●	●	●	●	●	●	●	○	●	○	○	○	●	●
⊕ win.x.y	embed	T4,embed,form	n=x	id=y & n=x	ch, sib	10	10	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○	○	●	●	●
⊕ doc.x.y	T4,embed,form,iframe	T4,embed,form,iframe	n=x	id=y & n=x	sib	11	11	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕ doc.x.y	T4,embed,form	T4,embed,form	n=x	id=y & n=x	sib	25	25	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕ doc.x.y	embed,image,img	iframe	n=x	id=y & n=x	ch	3	3	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕ doc.x.y	embed,image,img	T3,T4,embed,form	n=x	id=y & n=x	ch	15	15	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
win.x.y	T9,iframe	T1,plaintext-iframe	id=x	id=x & n=y	sib	1,436		●	●	●	○	○	●	●	●	●	●	●	●	○	●	●	○	○	●	●
win.x.y	T9	T1,plaintext-iframe	id=x	id=x & n=y	ch	1,301		●	●	●	○	○	●	●	●	●	●	●	●	○	●	●	○	○	●	●
⊕ win.x.y	form,image	T4,embed,form	n=x	id=y & n=x	sib	7	7	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	●	●	●
⊕ win.x.y	T4,form	applet	n=x	id=y & n=x	sib	4	4	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕ win.x.y	image	embed,form	n=x	id=y & n=x	ch	2	2	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	●	●	●
⊕ win.x.y	image,img	T4,embed,form	n=x	id=y & n=x	ch, sib	16	16	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●
⊕ win.x.y	T4	applet	n=x	id=y & n=x	ch	3	3	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
win.x.y	ins	content,data	id=x	id=x & n=y	ch, sib	4		●	●	●	○	○	●	●	●	●	●	●	●	●	○	●	●	○	●	●
win.x.y	T7, T8	T1,plaintext − iframe	id=x	id=x & n=y	sib	8,848		●	●	●	○	○	●	●	●	●	●	●	○	●	○	●	●	○	●	●
win.x.y	T8	T1,T11,plaintext − iframe	id=x	id=x & n=y	ch	7,526		●	●	●	○	○	●	●	●	●	●	●	○	●	○	●	●	○	●	●
⊕ doc/win.x.x	object	T4,embed,form	n=x	id=y & n=x	ch	5	5	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
doc.x.y	object	T4	id=x	id=x & n=y	sib	3		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕ doc.x.y	object	form,image,img	n=x	id=y & n=x	ch	3	3	●	●	○	●	●	○	●	●	○	●	●	○	○	○	○	○	●	●	●
⊕ doc.x.y	object	iframe	n=x	id=y & n=x	ch	1	1	●	●	○	○	○	○	●	●	○	●	●	○	○	○	○	○	○	●	●
doc.x.y	object	image,img	id=x	id=x & n=y	ch	2		●	●	○	○	○	○	●	●	○	●	●	○	○	○	○	○	○	●	●
⊕ doc.x.y	object	embed,object	n=x	id=y & n=x	ch	2	2	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
⊕ win.x.y	object	T4,embed,form	n=x	id=y & n=x	ch, sib	1	1	●	●	●	●	●	●	●	●	●	●	●	○	●	○	●	●	●	●	●
⊕ doc/win.x.y	object	T4,embed,form	id=y & n=x	id=y & n=x	ch	5	5	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
win.x.y	svg	iframe	id=x & n=y	id=x & n=y	sib	1		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
win.x.y	svg	T1,plaintext	id=x & n=y	id=x & n=y	sib	125		●	●	○	○	○	○	●	●	●	●	●	○	●	○	○	○	○	●	●
win.x.y	svg,table	T1,plaintext − T19	id=x & n=y	id=x & n=y	ch	157		●	●	○	○	○	○	●	●	●	●	●	○	●	○	○	○	○	●	●
win.x.x	table	iframe	id=x	id=x & n=y	ch	1		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
win.x.x	table	T1,plaintext,svg − T10	id=x	id=x & n=y	ch	119	0	●	●	●	○	○	○	●	●	○	●	●	○	○	●	○	○	○	●	●
win.x.y	table	iframe	id=x & n=y	id=x & n=y	ch	1		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
Total						31,432	148	59	59	46	35	35	46	59	59	44	59	59	43	38	45	52	37	35	59	59

formed not only for two similar HTML tags as in Table 5.1 (e.g., two `a` tags with `id=x`), but also for certain combinations of dissimilar tags (e.g., `svg` and `a`), which accounts for a large number of the clobbering instances. Other variants are cases where additional `id` and `name` attributes are added to the existing clobbering markups. For example, when looking at `form` elements and their childern in Table 5.1, we observe that each tag of the markup has only one `id` or `name`. However, as demonstrated by the results in §5.1.2.1, these attributes may exist simultaneously on HTML tags and with similar or dissimilar values, resulting in additional clobbering variants.

In comparison, the new clobbering instances rely on new (pairs of) HTML tags and attributes that were previously not known to be applicable for DOM Clobbering. We observed that 28 out of the 74 identified classes contain at least one new instance, with a total of 148 new instances. From these, 22 classes contain only new instances (i.e., 142 instances). In the remaining of this section, we briefly describe the new instances within each DOM Clobbering technique.

Named Access Window and DOM Tree Accessors. We discovered that any custom HTML tag (e.g., `customtag`) can be used to clobber a target variable x and window.x in all web browsers. Also, `iframe` tags with `id=x` can clobber document.x and named `applet` elements can clobber both window.x and document.x. In total, we found five new instances across four out of the 14 classes that rely on the Window Named Access and DOM Accessors techniques.

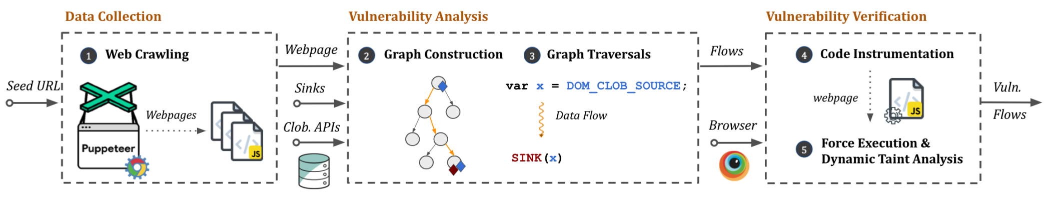
Form Parent-Child. We discovered that browsers like Firefox and Safari create accessor properties on JavaScript objects due to element's ancestral relationship in the DOM tree for previously unknown pairs of tags and attributes, such as a parent `form` tag with a `embed`, `iframe`, or `applet` child with both a `name` and `id` attribute. Overall, among the 13 classes that rely on elements' parent-child relationships, we found four new markups in four different classes.

Nested Window Proxy. We identified two new clobbering markups in two out of the four classes which use the Nested Window Proxies technique. In particular, we discovered that using the `id` attribute in the nested frames creates a named property on the base frame, referring to a `WindowProxy`, whereas `id` on the base frame does not create a `WindowProxy` accessible through the global `window` or `document`.

HTMLCollection. We found 137 new clobbering instances (across 18 classes) that lead to the construction of HTMLCollections in a different way. Specifically, we discovered that some browsers (e.g., Chrome and Firefox) create an HTMLCollection not only when two elements share the same `id`, but also when they have the same `name` value. However, we observed that this happens only for certain (combinations of) HTML tags, e.g., two `object` tags and two `form` tags with the same name can form an HTMLCollection, but not two `div` tags.

Analysis of Browsers' Behaviours. Our experiments revealed that browsers exhibit divergent behaviours when linking named HTML elements to JavaScript variables (§5.1.2.1). For example, we observed that for a significant fraction of the clobbering markups (i.e., 31,243 out of 31,432), there is at least one browser that disagrees with others, rendering the task of defending against DOM Clobbering increasingly more challenging. In summary, we identified 10 distinct groups of browser behaviours with respect to different DOM Clobbering markups, which are highlighted in §5.1.2.1 in colors, showing that while most of the attacks are shared across browsers, many others only work with specific browsers. The table shows that all Safari and iOS-based browsers have their own distinct behaviours, whereas browsers like Chrome, Opera, and Edge on Desktop and Android exhibit the same behaviour. Note that, in general, similarities in behaviours are expected because some browsers rely on the same underlying engine. For example, Chrome, Edge and Opera on Desktop are all Blink-based browsers [250], whereas iOS browsers are required to use the WebKit engine of Apple [251]. Finally, we observed that the least and highest amount of DOM Clobbering risk is associated with using browsers like Firefox Desktop/Android and

Figure 5.1: Architecture of JAW-v3.



Chromium-based browsers on Desktop/Android in which 35 and 59 classes of DOM Clobbering markups work, respectively.

5.1.2.3 Clobbering Native APIs

Overall, we identified a total of 347 DOM APIs that can be clobbered in at least one browser using one of the markups of §5.1.2.2, including 233 `document` and 114 `window` APIs. We observed that all `document` methods and properties except the `location` property (i.e., 233 APIs) can be clobbered in all browsers unanimously, as expected by the named property visibility algorithm [170] of the specification [87, 116]. However, this experiment resulted in a new finding that for a total of 114/347 `window` APIs (i.e., 91 properties and 23 methods), named properties can shadow native properties that would otherwise appear on the object in at least one browser, resulting in DOM Clobbering. This includes security-sensitive APIs such as the cache storage [111], notification API [252], trusted types [112], and web storage [253]–to name only a few instances. We observed that for 57/114 clobbered APIs, there is at least one browser that disagrees with others.

5.2 Detection and Prevalence

The second part of this chapter intends to evaluate the impact, prevalence and variety of DOM Clobbering vulnerabilities in real-world web applications (RQ3.2 of §1.1.3). In §5.2.1, we first present JAW-v3, an automated DOM Clobbering detection tool. Then, in §5.2.2, we present our experiment results.

5.2.1 Detection

We formulate the problem of detecting DOM Clobbering vulnerabilities into a series of data flow analysis tasks where we identify *clobberable* JavaScript variables, object properties, and native APIs whose value ultimately reach security-sensitive instructions, such as script `src` and `eval`. Identifying such data flows via pure static analysis is not an easy task given the dynamic nature of client-side JavaScript programs [11, 12, P1] and the scale of the analysis as studying DOM Clobbering vulnerabilities requires the collection and analysis of hundreds of webpages of real web applications. Accordingly, we use and extend state-of-the-art property graphs for JavaScript and graph traversals [P1] to identify potentially-vulnerable data flows and then use forced execution to confirm the presence of the vulnerability.

Figure 5.1 shows the architecture of JAW-v3. At a high level, it has three main components: (i) a web crawler to collect webpages' data and the JavaScript code, (ii) a vulnerability analysis component that uses property graphs and traversals for identifying potential DOM Clobbering sources and capturing data flows to security-sensitive sinks, and finally (iii) a vulnerability verification component that dynamically confirms the candidate data flows by instrumenting the code and forcefully executing it in a browser to check if the flow can occur at runtime. The rest of this section details each component.

5.2.1.1 Data Collection

To collect the client-side code of web applications, we developed a JavaScript-enabled crawler leveraging Puppeteer [254] and Chrome DevTools Protocol (CDP) [108]. Starting from a seed URL of the website under test, it visits the webpages following a depth-first strategy, and stops when it doesn't find new URLs, or the maximum of 100 URLs is reached. During the visit, it collects the page resources (e.g., scripts) and runtime state values (i.e., fired events and DOM objects' properties) using the CDP and Puppeteer.

5.2.1.2 Vulnerability Analysis

Given the webpages' data collected by the crawler, JAW-v3 creates a property graph of the client-side JavaScript program leveraging a modified engine of JAW [P1]. Then, we formulate the problem of finding potential DOM Clobbering data flows into a series of graph traversal queries.

Hybrid Property Graphs. As presented in §3.1, HPGs are graph-based representations of client-side JavaScript programs that unify multiple static code representations and runtime state values. State values are event traces and environment properties, e.g., the values of cookies and web storage. The static code representation comprises several graphs, e.g., Abstract Syntax Tree (AST), Control Flow Graph (CFG) and Program Dependence Graph (PDG) that model the nesting of the syntactical constructs of a program, the order and conditions for the execution of program instructions, and the data flow and control dependencies within the statements of a program, respectively. HPGs also model the event-driven transfer of control within JavaScript programs via the Event Registration, Dispatch and Dependency Graph [P1]. Finally, they include Semantic Types, which are labels initially assigned to source and sink nodes to capture the semantic of those instructions and then propagated through the graph following the program calculation. These representations are encoded in a directed graph in which nodes and edges can have labels and key-value properties, known as a labeled property graph [P1, 15].

Model Construction. After collecting the webpages' scripts and state values, JAW-v3 instantiates an HPG, and imports it into a Neo4j docker instance [186], allowing the graph to be traversed declaratively using the Cypher query language [197]. Unfortunately, we could not use JAW as-is and modified it to address several of its shortcomings. First, when building a graph, JAW normalizes the webpage code by combining code inside script tags into a single script. However, identifying DOM Clobbering sources may require to distinguish the code across two different scripts due to JavaScript variable hoisting [166] and double-clobbering [85]. For example, a runtime error in one script causes the browser to stop parsing that script, and continue with parsing of the rest of the scripts. Hence, variables initialized in the first script are treated as undefined and can be a candidate DOM Clobbering source. Such runtime errors can be caused intentionally by attackers by a preliminary clobbering, e.g., clobbering a native DOM function that is invoked in a script shadows its value to an HTML element, which is not callable, leading to a runtime error (Cf. Table 5.3). Accordingly, we changed the normalization procedure to keep track of the script of origin for each AST node.

Second, the semantic types of JAW are tailored for client-side CSRF vulnerabilities and are not sufficient to model DOM clobbering. Accordingly, we added a new set of generic semantic types for DOM Clobbering sources (Cf. Tables 5.3 and 5.4) and security-sensitive JavaScript sinks (Cf. Table A.2).

Third, JAW does not fully support ES6, resulting in imprecise control and data flow models. Accordingly, we applied several enhancements. For example, we added support to bind the function call arguments to their definition parameters when the code uses the ES6 Rest parameters [255] and the Spread operator [256] which improves the precision of the call graph and PDG edges. Also, we created bindings for the `this` object depending on the calling

Table 5.3: Description of properties of DOM Clobbering sources.

Object	**DOM Clobbering Source When?**
v	S1: $v \in$ NP, CLOB(v) S2: $v \notin$ NP, v and window.v are not assigned before, v is not declared with `var`, `let` and `const` before
window.v	S3: $v \in$ NP, CLOB(v) S4: $v \notin$ NP, v and window.v are not assigned before, v is not declared with `var` afterwards within the same script, or anywhere before S5: $v \notin$ NP, v or window.v is assigned or declared with any of the `var`, `let` and `const` keywords within any previous script that contains an invocation of function f such that $f \in$ NP, CLOB(f)
document.v	S6: $v \notin$ NP S7: $v \in$ NP, CLOB(v)

Legend: NP= native property; CLOB(v)= v is a clobberable NP based on §5.1.2.3.

context [257], and the binding for the `arguments` object for non-arrow functions [258] to improve pointer analysis tasks.

Analysis Traversals. After construction of an HPG, we traverse it to identify DOM Clobbering source nodes in the graph. Table 5.3 presents the various types of DOM Clobbering sources and their properties. The table shows that clobberable native DOM APIs discovered in §5.1.2.3 can act as a DOM Clobbering source. Identifying these objects in the program is a matter of searching for a pre-defined syntactic structure, which is similar to other taint-style vulnerabilities like client-side XSS. However, contrary to the traditional taint analysis, not all DOM Clobbering sources are pre-defined syntactic objects. Instead, they can be a specific property of a program, identifying which requires tracking the propagation of data flows within the program itself. This is because any used variable that is undefined within its execution context (i.e., previously not declared and assigned) can act as a DOM Clobbering source. To identify such sources, we use PDG data dependency edges, which specify that a variable defined at a source node is subsequently used at the destination node. Specifically, we query the graph for Identifier nodes containing a variable v with no incoming PDG edge from any AssignmentExpression or VariableDeclaration nodes that assign to or declare the variable v. If there is such PDG edge, we further check whether the declaration/ assignment statement can hinder the clobberability of v based on the criteria in Table 5.3, which can depend on the declaration scope (i.e., same script or not), declaration position (i.e., before or after), and the declaration keyword (e.g., `var` vs `let`) of that statement.

After identifying the source nodes, we associate to each of them a label that captures the semantic type of the source, e.g., a clobberable native property or custom variable (Cf. Table 5.4). Then, given a list of JavaScript sinks, we identify each of them in the graph and assign each a relevant semantic type. Semantic types assigned to sink instructions are propagated to other functions that encapsulate the same semantic, e.g., the type `WIN_LOC_WRITE` is set for instructions that set the value of `window.location`, such as `window.location.replace()`, and is then propagated to all other developer-defined functions that can set its value through one of their parameters. JAW-v3 considers different sink types to enable us to capture the potential consequences of DOM Clobbering. The complete list of sinks is in Table A.2, which is derived by surveying and aggregating the JavaScript sinks considered in prior academic and non-academic resources (see, i.e., [114, 263, 265, 261, 266, 113, 259, 32, 262, 260, 53, 160, 264, 54, 3]). Finally, we conduct forward data flow analysis by propagating semantic types from sources to sinks, and select those flows where a node with a sink semantic type is tainted with a source type (i.e., pick up the attacker-controlled values). This component outputs a set of paths with potential data flows from a DOM Clobbering source to a sink.

Table 5.4: Summary of DOM Clobbering sources and their semantic types based on the seven cases of Table 5.3.

Source	**Semantic Type**
S1: variable v	CLOB_CUSTOM_VAR
S4, S5: window.v	CLOB_WIN_CUSTOM_VAR
S6: document.v	CLOB_DOC_CUSTOM_VAR
S2: property p	CLOB_NATIVE_PROP
S3: window.p	CLOB_WIN_NATIVE_PROP
S7: document.p	CLOB_DOC_NATIVE_PROP

Legend: S_i= case S_i in Table 5.3;

Table 5.5: Prevalence and impact of DOM Clobbering in Tranco top 5K sites. The table shows the number of clobberable data flows to security sensitive sinks of Table A.2, the number of affected webpages, and websites.

Threat	**# Sinks**	**# Flows**	**# Conf.**	**# Pages**	**# Sites**
Client-side XSS	37,941,540	3,688	3,677	1,572	474
Request Forgery	2,555,147	1,406	1,403	541	398
Storage Manipulation	1,047,512	1,369	1,365	418	382
Open Redirect	1,306,603	1,228	1,227	391	385
JSON Injection	9,610,162	793	793	345	343
Cookie Manipulation	1,702,340	266	266	204	195
Websocket Hijacking	21,252	367	367	183	147
RegEx Injection	13,325,791	284	284	98	98
Doc. Domain Manip.	55,266	85	85	69	69
postMessage Manip.	119,971	0	0	0	0
File Read Path Manip.	57,789	0	0	0	0
Total	67,743,373	9,486	9,467	3,821	491

Legend: Conf.= Dynamically Confirmed

5.2.1.3 Vulnerability Verification

Given a set of potential DOM Clobbering data flows, the goal of this step is to verify each flow and eliminate potential false positives. To accomplish this goal, JAW-v3 features a light-weight, in-browser dynamic taint analysis engine leveraging *Iroh.js* [110]. After instrumenting the code with Iroh for dynamic analysis, we first check whether the source variable of the data flow is clobberable by creating a suitable HTML clobbering payload for that variable using the DOM Clobbering classes of §5.1. We inject the payload to the DOM tree and subsequently verify the clobberability of the source variable by dynamically logging its value at the source location.

As the next step, we confirm the existence of the data flow to the sink instructions. To do that, we first taint each clobberable source, execute the program by loading it via Puppeteer, and check if we can observe the data flow reported by the static analyzer. If that is not the case, we forcefully execute the path toward sinks to check if there is an execution of the program in which the data flow to the target sink occurs. We use forced execution to find candidate pages among those where Puppeteer could not connect sources with sinks, and later validate the presence of the vulnerability manually. Specifically, for each branch in the path control flow, we forcefully execute the program once for the true and once for the false branch, until we hit a execution path with the target data flow, or we exhaustively checked possible execution paths. We observed that the number of branches between DOM Clobbering sources and sinks is in practice small (i.e., less than 10), as we will show in §5.2.2. Finally, as forced execution may also lead to spurious execution paths, we manually validate the decision reported by JAW-v3 and examine the exploitability.

5.2.2 Prevalence in the Wild

We quantified the prevalence and impact of DOM Clobbering on the top 5K websites using the Tranco list [216] of Nov 1st, 2021 (ID: Y3JG), where we first selected the top 5K domains by excluding the duplicates like local versions of websites (e.g., *google.com* vs *google.de*), and then instantiated JAW-v3 for each of the them.

Data Collection Statistics. Starting from the 5K seed URLs, JAW-v3 collected 205,696 webpages, ranging between 1 to 91 pages per site (41 pages on average). Out of the 205,696 webpages, 187,280 are unique pages based on their set of scripts. From the 187K pages, JAW-v3 extracted 18,351,815 scripts with a total of 24,664,686,928 LoC. Accordingly, JAW-v3 generated 187,280 HPGs by processing an average of 98 scripts and 131,700 LoC per page.

Vulnerability Prevalence. The analysis of 187,280 HPGs resulted in the identification of 20,580,350 DOM Clobbering sources and 67,743,373 sinks, which amounts to an average of 110 sources and 362 sinks per webpage. Out of these, static analysis revealed a total of 9,486 potential data flows from the sources to the sinks, from which the majority (i.e., 9,467) were confirmed dynamically. We observed that these vulnerable data flows affect around 2% of the webpages (i.e., 3,821 out of 187,280) and 9.8% of the tested websites (i.e., 491 out of 5K) in total. Table 5.5 summarizes our findings.

Vulnerability Impact. We observed that the 9,467 vulnerabilities can have different security implications, as shown in Table 5.5. The most common consequence is XSS that accounts for around 38.8% of the vulnerabilities, whereas the least common consequence is document domain manipulation [261, 160] that corresponds to less than 1% of the total vulnerabilities. Other common consequences were client-side state manipulation (17.2%), client-side request forgery (14.8%) and DOM-based open redirection (12.9%). Finally, the remaining 15.3% of vulnerabilities had other repercussions like JSON injection and Websocket connection hijack. We provide more information on each of these threats in Table A.2.

Verification and False Positives. Considering the high number of reported data flows by the static analyzer (Cf. §5.2.1.2), it was infeasible to verify all of them manually. Instead, we followed a semi-automatic approach leveraging a combination of dynamic analysis, forceful execution and manual analysis, as detailed in §5.2.1.3.

We observed that in a large number of cases (46.1%, i.e., 4,373 flows), the dynamic verification component can successfully confirm the existence of the vulnerability by loading the page and executing it via Puppetter, whereas in the remaining cases (i.e., 5,113 flows), it needs to force execute between one to ten conditional branches (four on average) before it can confirm or reject the data flow and terminate. As a result of this process, the verifier eliminated a total of 19 FPs across 11 of the 491 vulnerable sites, and confirmed the rest (i.e., 5,094 flows within 2,643 webpages of 491 sites). We manually verified and investigated the reason for each FP, and discovered that eight FPs occur during the data flow analysis for identification of DOM Clobbering sources, and 11 during the data flow analysis from sources to sinks. The former cases happened because a variable was declared or assigned using a dynamic code generation construct for which the statement nodes and PDG edges were missing in the HPG, and the latter cases occurred due to dynamically fetched code where the value of the tainted variables changed, inaccurate pointer analysis for dynamic property lookups, and removal of event handlers that changed the tainted variables.

Finally, we manually validated the feasibility of the forcefully executed data flows by randomly selecting two pages per site, from the 2,643 pages of the 491 websites whose data flows were confirmed by forced execution. Our random sampling included 491 sites, 982 pages and 2076 data flows, out of which we could not determine a realistic execution path for at least 42 data flows in 42 sites, leaving us with 2,034 vulnerable data flows of 491 websites.

5.2.3 Confirming Exploitability of Vulnerabilities

We manually examined whether the identified vulnerabilities can be effectively exploited by an attacker. Given the high number of affected webpages, we randomly selected two vulnerable pages per each of the 491 affected sites, and subsequently checked whether we can insert a DOM Clobbering markup in the page by leveraging the functionalities offered by the application, or through URL parameters, which could allow us to overwrite the clobberable variable identified by JAW-v3. To be able to use protected functionalities offered by the websites (e.g., creating posts, adding comments, etc) and also prevent any side effects for other users, we created our own test accounts for 358 sites that supported this feature without monetary costs, and for the rest, we limited our tests to the public functionalities (e.g., search) without persisting any data. As a result, we created a proof-of-concept exploit for 44 websites in total, affecting popular sites and functionalities like Trello boards, Wiki pages in WikiBooks and WikiDot, comments in Vimeo and VK, reviews in TripAdvisor and OpenTable, posts in Fandom and JustPaste, surveys in SuveryMonkey, poster designs in PosterMyWall, and finally item searches in GitHub Shop, AliExpress, AliBaba and Telam News–to name only a few examples. The exploits enable an attacker to achieve XSS, open redirect, and client-side request forgery in 35, five, and four sites, respectively.

5.2.3.1 Case Studies

This section reports on a few manually vetted case studies of the confirmed attacks. We note that the affected parties have been promptly informed of the vulnerability, and have already patched them (see §8.1).

GitHub. This vulnerability affects the GitHub Shop and originated when loading the Boomerag JavaScript library [267]. In more details, the code followed the vulnerable pattern G of Table 5.7, where a variable called `BOOMR` was defined in an inital script that contained a clobberable, invoked native method, and a second script that used the object property `window.BOOMR.url` as the `src` of a dynamically added script. Attackers can escalate this vulnerable pattern to client-side XSS via double clobbering. First, they clobber the invoked native method, causing a runtime error when the browser parses the first script. Therefore, the browser stops parsing the rest of the script and `BOOMR` becomes undefined. Then, attackers can clobber `window.BOOMR.url` and consequently control the script `src` by injecting a DOM Clobbering markup, e.g., `<a id=BOOMR><a id=BOOMR name=url href=malicious.js>`. We discovered that it is possible to inject such non-script markup to the client-side page leveraging the search functionality and the URL query parameters, which were reflected back to the page.

Trello. We discovered that Trello uses a global object property called `window.ClickTaleScriptSource` to programmatically load a script named `wrScript`. However, this property was clobberable as `ClickTaleScriptSource` was an undefined variable following the vulnerable pattern A of Table 5.7. Finally, we found that it is possible to insert a persistent, non-script markup to overwrite this object property by editing a comment for a card in Trello boards, which resulted in arbitrary client-side code execution.

Fandom. We discovered a DOM Clobbering vulnerability in Fandom affecting the users' message wall that resulted in open redirection. Specifically, the JavaScript program contained an assignment to the `location.href` property of the top-level window, whose value was tainted with a clobberable object property, i.e., `form.elements.targetUsername.value`. Attackers can manipulate the value of this property by, e.g., two nested `iframe` tags that are named `form` and `elements`, and an additional `input` element in the nested frame. The input is named `targetUsername`, and has a `value` containing a malicious URL, which will be set as the window URL. We found that it is possible to inject non-script markup in the page in two distinct ways: (i) attackers can insert persistent payloads using the post functionality in the profile message wall, and (ii) a URL parameter in the path was reflected back to the page without extra validation, enabling transient insertion of clobbering payloads in the page.

5.3 Defenses

This section addresses RQ3.3 of §1.1.3. First, in §5.3.1, we have a critical look at the existing countermeasures and evaluate their robustness and cost-benefit tradeoff leveraging what we learned from Sections 5.1 and 5.2. Then, in §5.3.2, we analyze the common mistakes of the 491 vulnerable sites (see §5.2), and distill a list of recommendations and secure coding patterns that can resolve those issues.

5.3.1 Evaluation of Existing Countermeasures

Disabling DOM Clobbering Features. DOM Clobbering can be solved by disabling named properties [90, 91, 87]. According to Chrome telemetry [237], disabling named properties for clobbered variable accesses could break ∼10.5% of the *webpages*. Our results of §5.2.2 are in line with these numbers, and we observed that 13.3% of the webpages use at least an instance of clobbered variable accesses.

As webpages tend to reuse code via shared scripts, a patch in a script may fix multiple websites. Accordingly, using the number of webpages may not accurately quantify the cost of fixing breakage. As an alternative, we can measure the number of affected websites, and our results show that the affected pages do not concentrate on a small number of sites, but they scatter over 51.2% of the top 5K sites.

While breakage adequately measures the cost of this solution, it may not be a good indicator for the actual benefits, i.e., fixed websites. Our results show that 118 websites of 2,561 potentially broken sites will be fixed, which is about 4.61% of the broken websites (and 2.4% of the total). However, our results also show that a large fraction of vulnerable websites are not considered by breakage. In particular, we found 373 websites (76% of the vulnerable ones and 7.5% of the total) that will benefit from such a solution. Overall, when comparing the cost and benefits, the ratio of vulnerable over potentially-broken websites is about 1:5.2 (i.e., 491 vulnerable and 2,561 potentially-broken sites).

HTML Sanitization. HTML sanitizers can sanitize the input markups before adding them to the DOM tree, e.g., by removing the *id* and *name* attributes from certain (combinations of) HTML tags (Cf. §5.1). To assess the robustness of the popular HTML sanitizers against DOM Clobbering, we dynamically tested them against all of the DOM Clobbering instances we identified in §5.1. First, we selected the top five *web* programming languages based on the GitHub 2021 Octoverse report [296], i.e., JavaScript, Python, Java, C# and PHP. We considered both client-side and server-side JavaScript (i.e., node.js). Then, we searched for sanitizers of each language and selected the top five based on their GitHub stars, forks and UsedBy, and the number of downloads in their respective package managers (e.g., npm for node.js, packagist for PHP, etc). This process led to the identification of 29 HTML sanitizer libraries, as for Java, we identified only four sanitizers.

After identifying the popular sanitizers, we input the 31.4K DOM Clobbering markups identified in §5.1 to each of them, and for each input vetted whether the sanitizer removes or changes the named properties in the output markup. For each sanitizer, we tested both the default and most strict configuration that it offers. We marked a sanitizer as vulnerable if there is at least one clobbering markup that bypasses the sanitizer without being altered. Finally, we marked sanitizers as *partially* vulnerable when they encode the $<$ and $>$ symbols of HTML tags but do not remove or change the DOM Clobbering named properties because encoding these symbols would not help when applications expect inputs in an HTML format.

Table 5.6 summarizes our findings. In total, we observed that 16 and 13 out of 29 sanitizers are vulnerable to at least one DOM Clobbering markup in their default and most strict sanitization configuration, respectively. In both of the configurations, four sanitizers are only partially vulnerable, as they escape the markup rather than cleansing the named properties. Finally,

Table 5.6: Robustness of top five HTML sanitizers of web programming languages against the 31.4K DOM Clobbering instances of §5.1.2. The table shows the results for both the default and the most strict sanitizer configurations.

HTML Sanitizer	**Version**	★	⑂	👥	⬇	**Default**	**Strict**	**Bypassed**	**Pct.**	**Ref.**
Client-side JS										
1. DOMPurify	2.3.4	8.7K	534	49.7K	7.9M	●	●	29,995	95.4%	[78]
2. Google Closure Lib.	20211201.0.0	4.3K	1K	-	117K	○	○	-	-	[268]
3. JS-XSS	1.0.10	4.4K	584	136K	8.7M	◐	◐	25,592	81.4%	[269]
4. Sanitize-HTML	2.6.1	2.8K	316	102K	4.7M	●	○	79	0.25%	[270]
5. Google Caja	6015	1.1K	123	-	-	●	●	27,951	88.9%	[271]
Node.js										
1. Insane	2.6.2	394	21	-	55.3K	●	○	5	0.02%	[272]
2. Bleach	0.3.0	117	19	-	1.6K	●	●	2,288	7.2%	[273]
3. Angular-sanitize	1.8.2	100	237	49.1K	936K	○	○	-	-	[274]
4. Yahoo html-purify	1.1.0	40	6	-	708	●	●	28,807	91.6%	[275]
5. Arcgis	2.9.0	11	2	-	32.6K	○	○	-	-	[276]
Python										
1. Mozilla Bleach	4.1.0	2.3K	230	155K	17.5M	◐	◐	31,132	99.05%	[277]
2. LXML	4.7.1	2K	481	216K	29.9M	●	●	28,211	89.7%	[278]
3. HTML Sanitizer	1.9.3	61	19	-	17.9K	●	●	332	1.06%	[279]
4. Htmllaundry	2.2	27	4	-	1.1K	●	●	1,460	4.6%	[280]
5. Django-html-sanitizer	0.1.5	20	62	-	2.8K	○	○	-	-	[281]
PHP										
1. Htmlpurifier	4.14.0	2.4K	284	82.7K	2.5M	○	○	-	-	[282]
2. Html-sanitizer	1.5.0	333	36	-	30.8K	○	○	-	-	[283]
3. Symfony Sanitizer	1.0.0	104	1	-	7	○	○	-	-	[284]
4. HTMLawed	1.2	30	14	-	390K	●	●	21,211	67.4%	[285]
5. Typo3 Sanitizer	2.0.13	13	10	-	88.9K	◐	◐	23,942	76.1%	[286]
C#										
1. AntiXssEncoder	4.3.0	2.6K	1K	-	6.4K	◐	◐	31,390	99.8%	[287]
2. HtmlSanitizer	7.0.473	1.1K	162	1.8K	108K	●	○	654	2.08%	[288]
3. AJAX Toolkit	20.1.0	275	133	4.2K	264	○	○	-	-	[289]
4. NSoup	0.8.0	147	46	-	72	○	○	-	-	[290]
5. HtmlRuleSanitizer	1.6.0.1	50	16	30	308	○	○	-	-	[291]
Java										
1. Jsoup	1.14.3	9.2K	2K	98.4K	-	○	○	-	-	[292]
2. OWASP HTML Sanitizer	20211018.2	647	171	-	-	○	○	-	-	[293]
3. Antisamy	1.6.4	105	72	-	-	○	○	-	-	[294]
4. HtmlCleaner	2.25	-	-	-	824	●	●	28,951	92.1%	[295]
Total Vuln. (● + ◐)						16	13			

Legend: ★= GitHub Stars; ⑂= GitHub Forks; 👥= GitHub UsedBy; ⬇= Monthly Downloads;
● = Vulnerable; ◐ = Partially Vulnerable; ○ = Not Vulnerable

when looking at the remaining 13 sanitizers, we observe that they implement a robust, enabled-by-default defense. However, in all cases, they remove named properties unconditionally, i.e., for *all* input markups including those combinations that do not lead to DOM Clobbering, e.g., an anchor tag with `name=x` does not clobber the variable x. While such a strict approach is effective, it may hinder the usability of these libraries in cases where developers need to use `id` and `name` attributes for legitimate functionalities.

Content-Security Policy (CSP). When attackers can clobber the `src` attribute of dynamically created scripts, they can load and execute arbitrary JavaScript code. In these cases, the CSP `script-src` directive [92] can be used to constrain the value of script sources to a set of trusted domains, preventing attacker-loaded code to be executed [88, 8, 238]. However, unlike malicious JavaScript injected by the attacker, injected HTML code is not blocked by CSP. Accordingly, CSP does not mitigate other variants of DOM Clobbering that do not require script `src` manipulation, e.g., clobbering the parameters of dynamic code evaluation constructs like `new Function()` can lead to CSP-bypassable XSS. Our evaluation in §5.2.2 shows that 37.7% of the DOM Clobbering vulnerabilities that lead to XSS (i.e., 1,385 out of 3,677), which accounts for 14.7% of the total vulnerabilities can be mitigated by CSP, whereas the remaining ones cannot.

Freezing Object Properties. Another way to mitigate DOM Clobbering is to freeze DOM objects [93], e.g., via `Object.freeze()` method [297], which prevents the object to be overwritten by named DOM elements. While effective, determining all objects and object properties that need to be frozen is a non-trivial, error-prone task for web developers. Also, sealed objects cannot be changed anymore, hindering the dynamic composition of webpages. Finally, native properties cannot be frozen, rendering this approach ineffective when the DOM Clobbering source is a clobberable native property, which accounts for ∼21.5% of vulnerabilities (i.e., 2,037 out of 9,467) in §5.2.2.

5.3.2 Secure Code Patterns

Our evaluation of existing DOM Clobbering countermeasures in §5.3.1 revealed that they are not sufficient for complete protection in a large number of cases. In this section, we have a closer look at the variety of DOM Clobbering vulnerabilities in real web applications (§5.2.2), identifying vulnerable behaviours and the common types of coding mistakes. Then, we use these vulnerable behaviours to distill a list of recommendations and defensive coding patterns that developers could apply to prevent DOM Clobbering. To achieve this objective, we extracted the vulnerable lines of code and characterized them based on their high-level syntax and semantics, identifying eight distinct vulnerable code patterns in the wild.

Table 5.7 summarizes our findings. We observe that the most common mistakes are patterns A and E, in which the developer references an undefined variable through the `window` object, and then use the result in a sensitive instruction, whereas the least common, but also more complex mistakes are patterns F, G and H where the vulnerability originates due to the position of the instructions that span across two different script tags. Other common mistakes are patterns B and C, where developers treat custom and native `document` and `window` properties as trusted values that can be safely used in sensitive operations. The rest of this section presents secure coding patterns that can prevent DOM Clobbering.

Explicit Variable Declarations. As shown in Table 5.7, a key element enabling DOM Clobbering is use of the `||` operator to rely on specific defaults when the primary, intended variable or property is undefined. As an alternative solution, developers can initialize those variables with the default value when they are undefined using `var` declarations, which prevents named properties to overshadow them according to the named property visibility algorithm [170]. This solution could patch the patterns A, D, E, F, and H. When the value needs to be used in

Table 5.7: Overview of DOM Clobbering code patterns in the wild. Different background colors represent code in two different `script` tags.

#	Code Pattern	Description	# Flows	# Pages	# Sites

A	`var VAR2 = window.VAR1 || CONST;`
	`SINK(VAR2);`
		`VAR1` is not declared or assigned yet, thus `window.VAR1` is clobberable.	3,134	1,214	143

B	`var VAR2 = [WinDoc.]BA || CONST;`
	`SINK(VAR2);`
		`BA` is a clobberable built-in API (§5.1.2.3), thus `BA`, `window.BA` and `document.BA` are clobberable.	2,037	832	99

C	`[document.VAR1 = CONST];`
	`SINK(document.VAR1 || CONST);`
		Assignment to `document` properties is always shadowed by DOM Clobbering (§5.1.2.3).	1,896	655	81

D	`let VAR1 = VAR2 = CONST;`
	`SINK(window.VAR1 || CONST);`
		`VAR1` is declared with `let` that does not create property on window, thus `window.VAR1` is clobberable.	367	153	18

E	`SINK(window.VAR1 || CONST);`
	`VAR1 = CONST;`
		`VAR1` is initialized without `var` in the same script and after the sink, but this does not result in *hoisting*.	1,635	792	116

F	`SINK(window.VAR1 || CONST);`
	`var VAR1 = CONST;`
		`VAR1` is initialized with `var`, but in a different script and after the sink statement.	121	50	12

G	`BA() // clobberable built-in API`
	`[window.]VAR1 = CONST;`
	`SINK(window.VAR1)`
		`VAR1` is initialized in a script where a built-in method can be clobbered and cause an error in parsing that script, hence `window.VAR1` can be clobbered in a subsequent script (double clobbering).	53	36	7

H	`SINK(window.VAR1 || CONST);`
	`[window.]VAR1 = CONST;`
		`VAR1` is initialized in a different script as a property of the window or without any modifiers after the sink statement, thus `window.VAR1` is clobberable.	224	89	15

Legend: BA= Built-in API; WinDoc = Window or Document Object; [*code*]= Alternative *code* statement; Red= Clobberable; Yellow = script 1; Orange = script 2.

multiple scripts, as in patterns F and H, the declaration should be in the same (or a previous) script, but not in subsequent ones.

Strict Type Checking. Another common mistake enabling DOM Clobbering is treating DOM properties, like `document` and `window` properties as safe, trusted values (e.g., patterns B, C, and G). Instead, developers should extend the trust boundary to these properties, verifying their type before using them in security-sensitive instructions, e.g., using the `instanceof` [298] and `typeof` [299] operators.

Do Not Use Document for Global Variables. Properties of `document` can always be overwritten by DOM Clobbering, even immediately after they are assigned a value, as in pattern C. Accordingly, developers should refrain from using `document` as a mean to store and retrieve global values. Instead, they can declare variables with `const` or `var` in the global context, or use the `globalThis` object [300].

Namespace Isolation. While robust sanitizers in §5.3.1 remove named properties, an alternative solution is to separate the namespace of variables defined by JavaScript code and named properties in user-generated markups. For example, we observed that the markdown to HTML converter of applications like GitHub and BitBucket prefixes `id` and `name` attribute values of user-generated markup with a specific string. Motivated by this solution, one can monitor runtime changes in the DOM tree via the `MutationObserver` API [301], and prefix named properties of *all* dynamically inserted markups before adding them to the tree, which patches all patterns in Table 5.7.

5.4 Summary

DOM Clobbering is a type of code-less injection attack where attackers insert a piece of non-script, seemingly benign HTML markup into a webpage and transform it to executable code by exploiting the unforeseen interactions between JavaScript code and the runtime environment. Surprisingly, the attack techniques, browser behaviors, vulnerable code patterns, and defense mechanisms associated with DOM Clobbering have not been thoroughly investigated until now. This thesis addressed this gap, providing an in-depth exploration of the intricacies surrounding DOM Clobbering, from its underlying techniques to its prevalence and potential defenses.

As the first part of this chapter, we conducted an extensive exploration of the existing literature regarding DOM Clobbering techniques and performed dynamic analysis of 19 different web browsers. The outcome is the introduction of the first taxonomy of DOM Clobbering, revealing a staggering 31.4K distinct markups employing five distinct techniques to manipulate JavaScript variables. Then, we presented JAW-v3, the first DOM Clobbering detection tool, and instantiated it on the top of the Tranco top 5K sites, showing that DOM Clobbering vulnerabilities are prevalent, with a total of 9,467 vulnerable data flows across 491 affected sites, making it possible to mount arbitrary code execution, open redirections, or client-side request forgery attacks also against popular websites such as Fandom, Trello, Vimeo, TripAdvisor, WikiBooks and GitHub, that were not exploitable through the traditional attack vectors.

Finally, we assess the robustness of the existing countermeasures, such as HTML sanitizers and Content Security Policy, against DOM Clobbering, revealing shortcomings and proposing recommendations and secure coding patterns for developers. Our study marks a significant contribution to understanding and addressing DOM Clobbering in the Web increasingly dynamic landscape.

6 Studying the Effectiveness of SameSite Policies

In this chapter, we study the adequacy and effectiveness of same-site policies against cross-site attacks (XS attacks), such as cross-site information leakage (XS-Leaks) [137, 102, S2, 146] or cross-site request forgery (CSRF) [21, P1, 16, 20], addressing RQ4 of §1.1. Limiting the scope of cookies to first-party context is a long known countermeasure [66] to protect web applications from XS attacks, by stripping authentication cookies from cross-site requests the user nor the web application intended to initiate. Existing solutions require installing additional components such as HTTP proxies [66] or browser extensions [72], limiting their impact considerably. However, only very recently, Google revamped the idea of same-site policies for cookies by proposing and implementing in Chrome a new cookie attribute [174], the `SameSite` attribute. The `SameSite` attribute introduces three pre-defined same-site policies (None, Lax, and Strict)—one of which is the new default policy—each defining a set of cross-site requests contexts where the browser will not include cookies. By switching to a same-site policy by default, the hope is that XS attacks become old news [302, 303, 306, 305, S2, 304, 30].

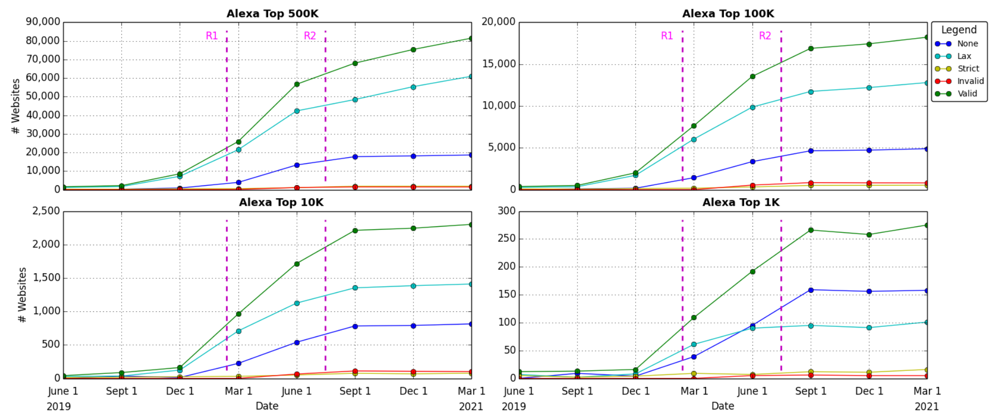
The radical change introduced by the SameSite attribute is that browsers no longer include cookies in all cross-site requests by default. As such a change can disrupt existing websites and to help developers transition to the new policy, Google rolled out `SameSite`'s features, spreading them over a period of four years, starting from April 2016, where it introduced the support for explicitly-defined SameSite policies, till July 2020 with the enforcement of the new default policy. As the new policies will play a major role to the security of the web platform, in this chapter, we take a closer look at the status of `SameSite` attribute before and after the enforcement of the new default SameSite policy. We conduct, to the best of our knowledge, the first evaluation of the SameSite cookie policy, systematically covering the trend of its usage (§6.1), the impact of its new default (§6.2), and the threats against its effectiveness (§6.3). We collect and examine the security risks as a result of the way developers are adapting to the new changes, and systematize the threats that can undermine the Lax protection, with the overarching purpose of studying the adequacy of the new default SameSite cookie policy.

Adoption and Breakage. First, in §6.1, we start with a longitudinal analysis of the SameSite cookie usage from June 2019 to March 2021 on the top 500K sites, and show that, even with a four-year rollout plan, only 18.94% of sites adopted one of the three policies, with a steep increase of +203.54% and +18.95% at the two dates of the new policy enforcement, respectively. Interestingly, 3.7% of the sites disable the SameSite protection using the None policy, with a significant increase towards more popular websites, i.e., 18% for the top 1K sites. Then, as the new Lax-by-default policy can break functionalities, in §6.2, we provide one of the first systematic analyses to identify and quantify functionality breakage in the wild due to the Lax policy. Our results show that, after the rollout of the new policy, about 19% of functionalities implemented via cross-site requests no longer work, most of which (77.5%) are for online advertisement.

Threats to Effectiveness. In §6.3, we take a look at the adequacy of the Lax policy to protect existing websites. In particular, we explore the tension between the protected contexts and the contexts that are used by existing websites to implement security-sensitive operations. In our evaluation, we first review academic and non-academic literature, and identify ten distinct threats, including three new threats inspired by prior work [307, 308]. Then, we assess their prevalence and practicality in the wild, showing a rather concerning scenario.

For example, we showed that 10.3% of state-changing requests of the top 1K sites (i.e., 721 out of 6,951) are still implemented via GET requests, which are not protected by the Lax policy, and in 2.6% of them, we successfully verified that CSRF attacks are possible (including popular sites like Mailchimp and Pixiv). Then, we discovered 1,302 distinct information leakage vulnerabilities (XS-Leaks) that leak the user's login status or identity leveraging window properties and postMessage, via requests that are not protected by the Lax policy. These XS-Leak attacks affect 40 websites of Alexa top 500, including popular ones such as Tumblr, Twitch, and SoundCloud. When looking at the requests that are protected by the policy, we identified known and new vulnerabilities in web applications hampering the SameSite cookies effectiveness.

Figure 6.1: SameSite Cookies usage (June 2019 - March 2021). The vertical lines `R1` and `R2` mark the two rollouts of the new default policy by Chrome.



For example, we discovered that 1.5% of the sensitive state-changing POST requests can be exploited for CSRF attacks by switching the method to GET. Among the vulnerable sites we found IMDB, PayPal, and Meetup. Also, we discovered a wide-spread behavior of Single Sign-On identity providers that can be used to abuse the exceptional SameSite policy used by Chrome (i.e., Lax+POST) to refresh session cookies and perform XS attacks within two minutes of the cookie refresh. The affected IdPs, among which we have Google, Facebook and Linkedin, are used by 49% of the top 10K Alexa sites. Finally, we observed three different incorrect and inconsistent use of the `SameSite` attribute, that can be exploited by attackers in XS attacks, i.e., different policies between mobile and desktop, cookies with different policies across web pages, and duplicated cookies with different `SameSite` attributes.

Browsers and Web Frameworks. Finally, in §6.4, we conduct a comprehensive analysis of 14 popular web browsers (both mobile and desktop) with regards to the SameSite cookies, and observed that none of them fully complies with the RFC 6265bis specification [28], exposing a total of seven divergent behaviours when enforcing the SameSite cookie policy. Even if browsers offer a Lax-by-default SameSite cookie protection, we show that 24% of the top five web frameworks of top five programming languages can downgrade that protection by default when the developer sets a cookie via one of the frameworks' offered APIs (e.g., in Django [309] or Pyramid [310]).

6.1 SameSite Cookie Usage

The first part of this chapter addresses RQ4.1 of §1.1.4, where we intend to measure how website developers adapted to the new cookie policies. In this section, we first review the methodology that we used to perform our measurements. Then, we present our findings.

Methodology. The methodology of this section consists in grabbing the cookie response headers of the Alexa top 500K sites (fetched in June 2020), and then extracting and counting the number of unique cookie attributes. Instead of conducting live measurements, we used the websites' copies stored by the Internet Archive, allowing us to retrieve past data enabling a longitudinal analysis of the usage of the SameSite. In total, we submitted queries to the Internet Archive for the response headers of the 500K domains archived from June 2019 to March 2021 with a three-months time interval. Fetching headers for 500K took in average six days. We divided the data collection in two periods: September 2020 and March 2021. At the end of each period, we performed a live measurement of the 500K domains to estimate the accuracy of

Internet Archive's data. In both measurements, live measurements resulted in at most +0.5% more successful responses.

Trend Analysis. Figure 6.1 shows the usage of the different SameSite policies from June 2019 to March 2021. As of March 2021, 80.7% of the sites rely on the default cookie policy (i.e., 347,251) whereas 18.94% of them adopted one of the three valid policy. When looking at the trend, the rollout dates R1 and R2 seem to have played a relevant role with a steep increase of SameSite attribute usage, especially for the None policy within the top 1K sites. A small, yet non-negligible fraction of sites set an invalid policy (1,430 sites, i.e., 0.33%). An invalid policy is a string that does not match any of the three known policies, such as `SameSite=1`, which are most likely developers' mistakes. Invalid policies should be treated as the None policy by web browsers according to RFC 6265bis [28]. We refer interested readers to Table A.5 that shows the top ten popular invalid policies. Finally, for at most 69,823 sites, a request to Internet Archive failed due to timeout.

None policy. In total, 18,640 sites use the None policy (i.e., 3.7%). However, we observe that the fraction of sites using None policy increases with the sites' popularity, from about one out of 10 among the top 10K sites (8.1%) to one out of five sites within the top 1K sites (i.e., 18%), making the None policy the most used policy among the top 1K sites.

Stricter policies. Then, 62,856 sites make it stricter with either the Strict policy or by explicitly setting the Lax policy. However, we note that the Strict policy is seldomly used (1,854 sites) when compared to the explicitly-set Lax policy, which covers almost all cases (61,002 sites, i.e., 97%).

6.2 Functionality Breakage

As we have shown in the previous section, 80.7% of the sites did not set the SameSite attribute, thus may rely on the new default policy which could break cross-site functionalities. In this section, we first identify the functionalities implemented via cross-site requests and then we provide a first measurement of the requests and websites that are affected. Before showing our results, we present the methodology of our analysis.

Methodology. Websites can use cross-site requests to implement various functionalities, e.g., advertising, social media buttons, etc. Identifying the functionality starting from a cross-site request is not trivial, and we are not aware of an automated technique able to do that. Accordingly, we design this section's experiments considering a human in the loop.

As we may need to evaluate requests manually, we limit our analysis to Alexa top 500 sites. We exclude duplicate sites (e.g., *google.com* and *google.co.uk*), sites that are not available in English (i.e., language barrier), and sites that do not offer free account creation and user login, resulting in 211 websites. Then, we register a user account for each site and crawl all sites using a browser enforcing the pre-SameSite default policy, searching for cross-site requests with cookies. We developed a JavaScript-enabled web crawler leveraging Puppeteer [254] and Chrome DevTools Protocol (CDP) [108]. Our crawler uses Chrome 83.0.4103.61, which does not enforce the new default Lax policy. Instead, it warns via the CDP Audits [311] when a cookie is attached to the request without specifying the `SameSite`, thus potentially breaking after the enforcement of the new default Lax policy. The seed URLs are the login pages, and after performing the user login with the manually-created credentials, the crawler follows a breadth-first visiting strategy to collect new URLs. The crawler stops when one of the two criteria is met first: it doesn't find new URLs, or the maximum of 200 URLs is reached. In total, our crawler collected 22,992 cross-site HTTP requests without a `SameSite`, which were initiated from 9,073 unique URLs.

To determine each request's high-level purpose, we first label our requests using the Web-Shrinker API [312]—a URL categorization service based on the industry-standard IAB taxonomy.

Table 6.1: Overview of the affected functionalities.

		After R2	
Functionality	**# Requests**	**# Broken**	**# Patched**
Advertising / User Tracking	374	93	281
Single-Sign On	81	1	80
Social Media Like / Share	76	11	65
Live Chat Frames	62	8	54
PDF Embed APIs	13	4	9
(Re-) CAPTCHA	12	2	10
Content Servers / CDNs	9	0	9
Survery/Rating Services	6	1	5
Total	633	120	513

Legend: R2= The Second Lax-by-Default Policy Rollout.

The WebShrinker API is limited in regard to the short-lived, continuously changing advertisement domains. Accordingly, we also match the request URLs against the EasyList [313], Host BlackList [314], and Host BlockList [315]—three popular blocklists specialized for advertisement domains. Finally, we derive a more fine-grained list of functionalities by manually inspecting the requests. We pick 633 random requests out of the 22,992, three from each of the 211 websites, and manually identify the exact functionality implemented by the cross-site request. Then, we forcefully remove the cookies to observe whether the functionality is broken. Broken requests are marked as *affected*.

To precisely evaluate functionality breakage after the new policy's enforcement, we execute our experiments before and after the second rollout R2. In June 2020, before R2, we identified and collected cross-site requests as presented in this section. Then, in February 2021, after R2, we revisited the affected requests to confirm whether the new policy has indeed broken the functionality.

Categorization. The mapping between affected cross-site requests and IAB categories is in Table A.4. In total, our crawler identified 22,992 cross-site HTTP requests without a `SameSite`, for a total of 9,073 unique URLs. The mapping identified 16 high-level categories of websites, providing 32 different types of functionalities. The vast majority of affected requests are for those sites offering technology and computing functionalities (e.g., file sharing, or live chat) or business services (e.g., advertising, marketing, or analytics), accounting for over 43.6% and 27.6% of the requests, respectively. For 303 requests of 17 websites, WebShrinker only matches the *uncategorized* category (see Table A.4), thus we used EasyList, Host BlackList, and Host BlockList, and observed that all the 303 requests are for undesired content, e.g., ads.

While the IAB categorization provides insights about the type of the service provided by third-party sites, it does not help to identify the exact functionality implemented with cross-site requests. The manual investigation of 633 requests identified eight functionalities, as shown in Table 6.1. Advertising and user tracking is the first type of functionality implemented via cross-site requests, covering 59% of the requests. Then, 12.8% of the requests are for Single Sign-On services, 12% for social media buttons, and 9% for embedded live chat services (e.g., *livechat.com*). The remaining groups are embedded PDF readers (e.g., Adobe Audience Manager), CAPTCHAs, CDNs (e.g., Gigya), and survey/rating services (e.g., *surveymonkey.com*).

Breakage. For each of the 633 requests, we manually confirm whether the new policy for cookies breaks the implemented functionality. We conduct these experiments in February 2021. First, we visit the affected site using a Chrome version enforcing the new Lax-by-default policy. Then we check whether the developers of the affected websites adopted one of the three SameSite policies to avoid service discontinuity. About 81% of the affected cross-site requests are correctly patched, and we do not observe breakage of functionality. However, we observe that the functionalities implemented by 19% of the affected requests are broken. Out of these, 77.5% of the requests are directed to advertisement networks, 9.2% to social media platforms,

Table 6.2: Overview of threats to SameSite cookies, grouped by those not covered by Lax (top part) and those covered by Lax (bottom part). Threats marked with * are new, yet inspired by prior work.

		Attack			Evaluation			
Category	**Threat**	**COSI**	**CSRF**	**References**	**Testbed**	**% Vuln.**	**# UAV**	**# Apps**
Not Protected by Lax	Replaying State-changing GET	○	●	[105, 328, 331]	Top 1K	2.6% G-SCRs	7	4
	Window Properties Leak	●	○	[34, S2, 334]	Top 500	18.48%	1021	39
	postMessage Leak	●	○	[342, 341, S2]	Top 500	1.9%	11	4
	Pervasive Monitoring	●	○	[175, 343]	Top 500K	0.4%	2,080	2,080
Protected by Lax	Forging State-changing POST	○	●	[104, 330, 329]	Top 1K	1.5% P-SCRs	7	6
	SSC SSO Redirects Bypass	●	●	[103, 316, 339]	Top 10K	49.3%	6	4,935
	SSC Intra-Page Inconsistency*	●	●	[345, 344]	Top 500	1.4%	3	3
	SSC Inter-Page Inconsistency*	●	●	[307]	Top 500	3.3%	11	7
	SSC User-Agent Inconsistency*	●	●	[307, 308, 319]	Top 500K	1.8%	9,215	9,215
	Client-side CSRF vulnerability	○	●	[1, P1]	-	-	-	-

Legend: ● = threat applicable; ○ = threat not applicable; SSC= SameSite Cookie; UAV= Unique Attack Vectors; G/P-SCR= GET/POST-based State-Changing Request.

and 13.3% for the remaining functionalities.

6.3 New Default Policy Adequacy and Threats

This section evaluates the adequacy of the new default policy in protecting websites in the wild and an extensive analysis of threats that can hamper the effectiveness of the new same-site policies.

Methodology. The first step of our analysis is identifying a list of threats, distinguishing them in threats leveraging cross-site requests that are not covered by the Lax policy and threats covered by the Lax policy. We systematically review academic literature (i.e., [21, 307, 318, 316, 317, P1, 308, S2, 319]), the Stack Exchange [320] and the Dev [321] security communities, and 21 non-academic resources (i.e., [103, 323, 325, 326, 104, 336, 322, 338, 330, 324, 105, 329, 328, 335, 331, 333, 327, 339, 332, 337, 334]). We searched for non-academic resources via Google search (up to page eight of the results). We used the search term "SameSite cookie" in combination with "bypass", "attack", and "vulnerability", and ignored irrelevant or redundant entries. We consider in scope those threats that can be exploited by a web attacker, and those that are relevant for SameSite cookies according to the identified resources (see, e.g., [17, 103, 104, 340, P1, S2]). Our review identified seven known threats. We also defined three new threats that are inspired by prior work. We present the threats in §6.3.1. The second step of our analysis is determining the severity of the threats by looking at their prevalence in the wild. As each threat requires an ad-hoc testing procedure, we describe our tests in §6.3.2.

6.3.1 Threats

We now present ten threats, of which four are against server-side end-points that are not protected by the Lax policy, and six due to vulnerabilities introduced by developers that can hamper the effectiveness of the SameSite attribute. Table 6.2 shows an overview of the threats.

6.3.1.1 Threats not Covered by the Lax Policy

Replaying State-changing GET Requests. The new default policy does not prevent the inclusion of cookies in top-level navigation requests. If web applications use GET requests for security-sensitive state-changing operations, attackers can forge *authenticated*, cross-origin HTTP requests on behalf of victims, e.g., leveraging the `window.open()` JavaScript API (see Table 2.1).

Window Properties Leak. Attackers can issue a top-level navigation request via, for example, `w=window.open()`, and read the number of frames in a target webpage using the `length` property of the opened JavaScript window objects (note that `w.frames.length` is the leaking channel). For example, if the victim is logged in, the attacker will count x frames, and zero otherwise. By comparing the frame count, attackers can leak the user state. For example, using this side-channel, it was to possible leak sensitive information about a user and their friends on Facebook [34], or to determine if a user owns a specific profile in Linkedin (i.e., user deanonymization) [S2]. Top-level navigation requests are not covered by the Lax policy, meaning that top-level requests will include cookies, leaving the leaking channel observable by the attacker. On the contrary, if developers set the SameSite policy to strict, this XS-Leak is mitigated.

postMessage Leak. Attackers can issue top-level navigation requests using `window.open()`, listen to broadcasted postMessages from the opened web page, and leak the user state by comparing the set of observed messages [S2]. For the postMessage leak, the attack pattern is similar to the previous threat. The only difference is that the observable leaking channel is no longer the number of frames but the attacker is listening for broadcasted postMessages. For instance, if the victim is logged in, a postMessage m is observed, and no messages otherwise. Also, in this case, should the cookie not be included in top-level navigation requests, the attacker would not be able to observe differences across user states (e.g., logged in vs logged out) [S2].

Pervasive Monitoring. Third-party cookies are widely used to track users online, and they often contain sensitive data. If websites do not set the `Secure` attribute for these cookies, a viable threat is pervasive monitoring at network level.

Assume a website $W1$ that set a privacy-sensitive cookie with `SameSite=None` and another website $W2$ that performs cross-site requests to $W1$. Because of the policy set by $W1$, browsers will include cookies in all requests from $W2$ to $W1$. This is the typical setting of third-party cookies widely used for tracking users. Pervasive (network) monitoring is a threat to these scenarios because if cookies are not securely transported (i.e., over TLS), they can reveal sensitive information about user identity. For this reason, browsers like Chrome and Opera reject cookies that do not set the `Secure` flag together with `SameSite=None` policy [340]. However, other browsers such as Firefox and Safari do not reject these cookies (see Table 6.9), exposing users of these websites to pervasive monitoring attacks.

Cookie-less Request Authentication. While cookies are one of the most prevalent forms of request authentication, they are not the only one (see, e.g., [346]). SameSite cookies can protect those class of request forgery attacks that perform ambient HTTP request authentication with cookies. Accordingly, other forms of request authentication, such as HTTP authentication, client certificate authentication [347], or network-based authentication are not protected by SameSite cookies.

6.3.1.2 Threats Bypassing Protection of Lax or Strict Policy

Forging State-changing POST Requests. One of the fundamental limitations imposed with the new default Lax policy is that an attack page cannot submit cross-site POST requests to a third-party context with the victim cookies attached. However, some applications are vulnerable in the sense that a state-changing POST request can be replayed and forged with a GET request interchangeably. In other words, the vulnerable application still processes the incoming request regardless of the HTTP verb used to submit the request. In this setting, the new default SameSite policy can be bypassed, e.g., by replaying the request using a top-level navigation GET request.

Single Sign-On HTTP Redirects Bypass. The Lax+POST exceptional policy (see §2.3) provides a time window of two minutes where Lax protection is not enforced, which is counted starting from the time of setting of a cookie. A possible attack consists of installing new cookies using cross-site requests and using the two-minute window to exploit XS vulnerabilities. Fresh cookies could be installed, for example, by abusing Single Sign-On Identity Providers (IdPs) that allow for user auto re-login via HTTP GET requests and without requiring user interaction (e.g., CAPTCHAs) [103]. The attack against a target site is the following. First, the attacker convinces a user to visit an attack page. Via the `window.open()` API, the page asks the IdP to re-login the user at the target site. As a result of the SSO login, the target site establishes a new authenticated session with the user's browser. Since the cookie is not older than two minutes, Lax protection of the target site is not enforced, enabling the attacker to mount XS attacks.

Listing 6.1: A vulnerable example of a duplicate cookie setting.

```
// for incompatible clients
Set-cookie: 3pc-legacy=value;
// for newer clients
Set-cookie: 3pc=value; SameSite=Strict;
```

SameSite Cookie Intra-page Inconsistency (new). When developing web applications, providing support for older web browsers that are incompatible with the SameSite cookie policy is challenging. In such incompatible clients, a cookie marked with a `SameSite` attribute or an unsupported SameSite policy may be rejected and not set, thus breaking the application functionality. As a workaround, developers may set redundant cookies, both with and without the `SameSite` attribute [345, 344], or with different `SameSite` policies. However, this can introduce vulnerabilities if not properly applied. For example, Listing 6.1 shows a vulnerable cookie setting that can be exploited to mount a CSRF attack. In this example, the application sets two duplicate cookies, namely `3pc` and `3pc-legacy`, with Strict and no `SameSite` policy, respectively, and resorts to the `3pc-legacy` if the `3pc` cookie is not included in the request. For a victim vising a CSRF attack page using a modern client, the `3pc` cookie is not attached to the cross-origin CSRF request, but the `3pc-legacy` cookie is still automatically attached to the request, both when assuming a client enforcing a default None or default Lax policy (i.e., using top-level navigation requests), enabling CSRF on server-side.

SameSite Cookie Inter-Page Inconsistency (new). This vulnerability occurs when a web application sets two different SameSite cookie policies for the same cookie with the same `Path` attribute across two different web pages. For example, if an application sets `3pc=value; SameSite=Strict; Path=/` when visiting URL$_1$ and `3pc=value; SameSite=None; Path=/` when visiting URL$_2$, then the Strict policy for this cookie can be bypassed. In this example, the bypass happens by issuing a top-level navigation request to URL$_2$, which overwrites the cookie with the `SameSite=None` attribute, relaxing the SameSite policy.

SameSite Cookie User-Agent Inconsistency (new). This vulnerability arises when an application set inconsistent cookie policies when using two different user agents. For example, a web application may enforce the Strict or Lax policy for a sensitive cookie when the user is using a desktop browser, but enforce the None policy if the user uses a mobile browser (or vice versa). One reason for such inconsistency is that the mobile and the desktop version are two different applications exposing themselves to the public based on the request user-agent. In such circumstances, CSRF and COSI attacks are possible provided that the victim uses a user agent with the less stricter SameSite policy to visit the target website.

Client-side CSRF. When an attacker-controllable input of client-side JavaScript program is used to generate same-site requests, a web application is exposed to client-side CSRF attacks. As these requests are same-site, the browser will attach cookies, even when using the Strict policy. In this paper, we do not examine the prevalence of this threat as it has been extensively studied in a recent work (see, i.e., [P1]).

6.3.2 Threats Prevalence in the Wild

Starting from the threats of §6.3.1, we now quantify their impact and prevalence in the wild.

Summary of Findings. Our evaluation shows that the coverage of the new default SameSite cookie policy (i.e., Lax) is not sufficient to protect web applications from a noticeable set of CSRF and COSI attacks. More specifically, as we show later, the first category of attacks presented in this section leverage cross-site request contexts that are not covered by the Lax policy. Then, the second category of attacks demonstrate the prevalence of cases where the protection of Lax can be circumvented. Table 6.2 summarizes our findings.

Lax Adequacy. In our evaluation, we identified both CSRF and COSI attacks against popular websites. First, GET requests (not covered by the Lax policy) are still prevalently used for state-changing operations—accounting for over 10.3% of the total state-changing requests in top 1K Alexa websites, 2.6% of which can be leveraged for mounting CSRF attacks by replaying the request (e.g., in Mailchimp or Pixiv). Second, we discovered 1,032 distinct information leakage (i.e., COSI) vulnerabilities affecting 40 websites of Alexa top 500, including Tumblr, Twitch and SoundCloud, that leak the user's login status or identity leveraging window properties and postMessage side-channels. We detail these threats in Sections 6.3.2.1 to 6.3.2.3.

Bypassing Lax. We identified a wide range of attacks that hamper the effectivenss of the Lax policy. For example, we discovered that 1.5% of the sensitive state-changing POST requests can be exploited for a CSRF attack by using the GET HTTP method instead. We found instances of these CSRF attacks in popular websites, e.g., IMDB, PayPal, or Meetup. Also, we discovered six unprotected SSO IdPs, including Google, Facebook and Linkedin, that enable trivial bypass of the Lax policy on over 49% of the top 10K Alexa sites, leveraging the exceptional Lax+POST policy. When looking at SameSite cookie policy inconsistencies, our evaluation of popular Alexa top 500 websites revealed seven vulnerable sites with inter-page policy inconsistencies, including Vimeo, AliExpress and Office365, as well as three vulnerable sites with intra-page policy inconsistencies, i.e., GitHub, CNN, and Yahoo. Finally, we discovered 9,951 vulnerable websites with inconsistent SameSite cookie policies for different user-agents, by systematically testing half a million Alexa sites, which can be used to bypass the constraints of the new SameSite cookie setting. We detail these threats in Sections 6.3.2.4 to 6.3.2.8.

6.3.2.1 Replaying State-changing GET Requests

In this section, we show that state-changing requests that use the GET method and that are *not* protected by the new default SameSite cookie policy are prevalent, and we demonstrate real-world CSRF exploitations in popular websites leveraging such requests.

Quantification of Request Types. Our methodology to quantify the prevalence of state-changing GET requests are as follows. We use the web crawler of JAW [P1] to crawl Alexa top 1K websites. The crawler stores a JavaScript-enabled, DOM snapshot of the web page after ten seconds. To identify state-changing requests at large-scale, we create a script that finds all HTML forms in the DOM snapshots with an anti-forgery token—an indication that the HTTP request would change the server-side state when the form is submitted. Then, the script filters all forms based on their HTTP method, quantifying their prevalence. We note that, even if these requests are seemingly protected by a CSRF token, the implementation of the defense may have been wrong (e.g., faulty token verification when overriding the HTTP request method [348, 349])—a mistake that may have acted as a contributing factor for the introduction of the new SameSite cookie setting. Finally, we compare our findings with prior work, i.e., the data of Mitch [21].

Table 6.3 summarizes the results. In total, the crawler finds 42,571 URLs for 922 websites. In these pages, our script identifies a total of 6,951 state-changing requests. Out of this number, the majority, i.e., 6,230 are POST-based requests. Still, a noticeable fraction of all identified

Table 6.3: State-chaning GET and POST requests in Alexa top 1K websites.

Method	POST	GET	Total
# Reqs	6,230 (89.6%)	721 (10.3%)	6,951
# URLs	1,870	251	2,121 / 42,571
# Apps	602 (65.2%)	88 (9.5%)	690 / 922

Table 6.4: Summary of CSRF vulnerabilities discovered for a set of randomly selected requests of Alexa top 1K websites.

Rank	Website	Replay Req.	Forge Method	Total Req.
58	imdb.com	0	2	2
81	fandom.com	0	2	2
102	paypal.com	0	1	1
289	ilovepdf.com	0	1	1
300	investing.com	2	0	2
427	meetup.com	0	2	2
524	mailchimp.com	1	1	2
586	brilio.net	3	0	3
627	pixiv.net	1	0	1
Total Vuln.	9 / 690	7 / 264	9 / 602	16 / 866

state-changing requests are based on the GET HTTP method, i.e., over 10.3%, which as we will show, is in line with prior research [21]. Specifically, we use the dataset of Mitch [350] to confirm our findings. The dataset contains a total of 58,828 HTTP requests for 60 popular Alexa websites. Out of this number, we observe that 938 requests contain an anti-forgery token, an indication that the request is state-changing. From 939 requests, 121 use the GET HTTP method, i.e., 12.8%, which is statistically close to our finding of 10.3%. Therefore, GET requests are still used in practice for state changes, despite the fact that they are not protected with the default SameSite cookie policy.

Exploitations. We manually explored the collected data to detect concrete GET-based CSRF exploitations. Given the scale of our data, we randomly selected three GET requests from each web application for which we detected a GET request, i.e., 88 applications (see Table 6.3), resulting in a total of 264 requests. Then, for each selected request, we checked if the CSRF token verification is performed correctly by replaying it. Table 6.4 presents our findings. In total, we discovered that seven out of the 264 GET requests (i.e., 2.6%) are forgeable due to faulty CSRF token verification, affecting four websites, i.e., Mailchimp, Brillo, Investing, or Pixiv. We created a working proof-of-concept exploit for each vulnerable web application. The exploits allow an attacker to delete user sketches in Pixiv, delete articles, videos and pictures (i.e., user-generated content) in Brillo, create or remove user portfolios in Investing, and finally change user settings' defaults (e.g., notifications) in Mailchimp.

6.3.2.2 Window Properties and postMessage Leaks

We investigate the prevalence of window properties and postMessage XS-Leaks using the data of our crawl of §6.2 on the Alexa top 500 websites, i.e., 9,073 URLs of 211 websites. We automatically explore the presence of login detection and user deanonymization attack vectors leveraging dynamic analysis. Specifically, we create a script that, for each URL in our dataset, loads a candidate test web page inside the browser in two different user states, i.e., logged and not logged for login detection and logged as two different users for deanonymization. For window properties leak, the test web page opens the URL in a new window leveraging the `window.open()` API, and reads the `length` property of the opened window, repeating this process for both user states under test. Similarly, the same process is performed for postMessage leaks, but instead of reading the number of frames in the opened window, the test web page listens for broadcasted postMessages using the `window.addEventListener` API. Finally,

Table 6.5: Summary of Window Properties (WP) and postMessage (PM) information leakage vulnerabilities discovered in Alexa top 500 websites.

XS-Leak		**Login Det.**	**Deanonym.**	**Total**
PM	# Apps	4	1	4
	# URLs	9	2	11
WP	# Apps	39	8	39
	# URLs	986	35	1,021
Total	# Apps	40	8	40
	# URLs	995	37	1,032

Table 6.6: Summary of the discovered security risks due to the missing `Secure` flag in `SameSite=None` cookies.

	# Vuln. Alexa Websites			
Crawl	**Top 1K**	**Top 10K**	**Top 100K**	**Top 500K**
June 2019	0	0	3	12
Sept. 2019	9	34	70	113
Dec. 2019	1	4	35	135
March 2020	6	29	227	644
June 2020	12	74	556	1,918
Sept. 2020	12	82	609	2,076
Dec. 2021	12	71	597	1,987
March 2021	12	82	634	2,148

the script compares the values collected at the two different states, and outputs the set of state-dependent, leaky URLs.

Table 6.5 summarizes the results of our experiment. In total, we discovered 1,302 vulnerable URLs, belonging to a total of 40 distinct websites. Out of 1,302, 37 URLs can be exploited for deanonimyzing the user's identity in eight websites, i.e., Tumblr, Twitch, AliExpress, Blogger, Office365, Tokopedia, Ebay, and SoundCloud, and the rest (i.e., 995) can be trivially exploited for mounting login detection attacks in all 40 vulnerable websites, including privacy-sensitive sites, such as PornHub. Note that being logged in implies having an account, which may be problematic for privacy-sensitive websites. Overall, we observe that 18.4% and 1.9% of the tested web applications are vulnerable to the window properties and postMessage side-channels, respectively.

6.3.2.3 Pervasive Monitoring

We reuse the data we collected from Internet Archive between June 2019 to March 2021 (§6.1) to identify cookies marked with `SameSite=None` that miss the `Secure` flag. In total, we detected 2,148 websites who are at risk of compromising user's privacy, 12 and 82 of which belong to the top 1K and 10K websites, respectively. Table 6.6 summarizes the results of our analysis. We observe that there is an increasing trend on the instances of this security risk in the wild.

6.3.2.4 Forging State-changing POST Requests

We reuse the data we collected in §6.3.2.1 to assess the prevalence of forgeable state-changing POST requests. We manually explored the collected data to detect concrete instances of forgeable POST requests, where attackers can bypass the Lax protection by changing the HTTP method. As shown in Table 6.3, in total, we identified 6,230 state-changing POST requests in 602 web applications of Alexa top 1K websites. Given the scale of the data, we randomly selected one state-changing POST request per web application, resulting in 602 requests. For each selected request, we checked the susceptibility to a CSRF attack by replaying the request using a different HTTP method, i.e., GET. In addition to the HTTP method change, we encode the key-value pairs in the POST request body, if any, in the form of GET request query parameters. Table 6.4

summarizes our findings. In total, we discovered that nine out of the 602 requests (i.e., 1.5%) are forgeable, affecting six popular websites (e.g., PayPal, IMDB, or Meetup). We created a proof-of-concept exploit for each of the six vulnerable web applications. The exploits allow an attacker to add or remove movies from a user watchlist in IMDB, change user settings (e.g., name, gender, or profile title) in Fandom, modify user invoices and extend the user session in PayPal, editing a user's signature in iLovePDF, and finally creating or removing notification alerts in Meetup.

6.3.2.5 Single Sign-On HTTP Redirects

To identify SSO IdPs that enable bypass of the Lax policy, we create a web application that integrates SSO using 13 different popular IdPs, i.e., Google, Facebook, Amazon, Apple, Microsoft, Linkedin, Github, Twitter, VK, Mail.ru, Twitch, Instagram, and Yahoo. To derive the list of popular IdPs, we manually review Alexa top 500 sites, and list the IdPs they use for SSO. We investigate if each IdP can be leveraged to bypass Lax by checking if it offers a cross-origin GET-based auto re-login feature that does not require any user interaction (e.g., CAPTCHA). For each affected IdP, we find websites from Alexa top 500 that integrate a SSO feature via that IdP, and verify if the attack still works in the real-world setting.

To quantify the impact of affected IdPs on websites, We built a JavaScript-enabled, Chrome-based web crawler on the top of XDriver [318], and used it to detect the IdPs each website is using for the SSO. The detection of the IdPs is similar to that of [318], and is based on a set of fine-grained static probes and regular expressions that we design for each IdP. We use our crawler to detect the IdPs in Alexa top 10K websites, examining tens of thousands of web pages.

Results. In total, we found six SSO IdPs that enable trivial bypass of the new default policy, i.e., Google, Facebook, Microsoft, Linkedin, VK and GitHub. These IdPs are integrated in 4,935 websites, accounting for more than 49% of the top 10K Alexa sites. To identify the affected websites, our crawler examined a total of 208,464 web pages of 9,485 sites in a period of around two weeks, designating 6,638 login pages, out of which 5,180 are login pages with an SSO. From these pages, 4,935 are login pages with at least one of the six affected SSO. For 515 sites, our crawler failed because either the website was unresponsive or XDriver failed when looking for DOM elements. Table A.6 in Appendix A.2 summarizes our findings.

False Positives. To evaluate the potential false positives (FPs) of our automated SSO detection mechanism, we randomly selected 500 websites, and manually verified the detected IdPs. This resulted in a total of seven FP IdP instances for four websites. In all cases, the underlying reason for the FP was that the heuristic used to match the existence of the IdP was present in the website for non-SSO usecases, e.g., in websites containing tutorials, or documentation about an SSO. Accordingly, our crawler exhibits an estimated false positive rate of $7/(13\times500)$ IdP instances, or 4/500 websites (0.8%).

6.3.2.6 SameSite Cookie Inter-Page Inconsistency

We investigate inter-page policy inconsistencies using the data of our crawl of §6.2 on the top 500 Alexa websites. We create a script that compiles a list of cookies set on each website together with the URL of web pages on which the cookie was set. Then, the script looks for redundant cookie entries across web pages, and for each matching case, it checks if the value of the `SameSite` attribute is consistent in all cases, and otherwise, it reports the inconsistency. Finally, for each reported case by the automated script, we manually confirm the inconsistency on the live instance of the application.

In total, out of the 211 websites of the top 500 Alexa, this process led to the detection of seven vulnerable sites having a total of 11 cookies with policy inconsistencies. This includes,

Table 6.7: Summary of inter-page SameSite cookie inconsistencies in Alexa top 500 websites.

Rank	Website	Policy Downgrade	# Vuln. Cookies
38	aliexpress.com	Lax to None	2
148	kompas.com	Lax to None	3
151	office365.com	Lax to None	1
170	canva.com	Strict to None	1
176	vimeo.com	Lax to None	1
191	abs-cbn.com	Lax to None	2
199	aliyun.com	Strict to None	1
Total	7		11

among others, popular websites such as AliExpress, Vimeo, and Office365. In all cases, an attacker can downgrade the Lax or Strict policy to None. Table 6.7 summarizes the results.

6.3.2.7 SameSite Cookie Intra-Page Inconsistency

We explore the presence of duplicate cookies having inconsistent SameSite cookie policies with a semi-automated approach, leveraging the data of our crawl of §6.2 on the Alexa top 500 websites. Specifically, we create a script that compares all the cookies set in a web page with each other. The script checks if it can find a pair of cookies that have the same value. If a match is found, it checks if their specified `SameSite` policy is different, i.e., no `SameSite` or None policy for one cookie, and Lax or Strict for the other. Since two session cookies may trivially have the same value (e.g., an integer), yet do not encode the same semantics, the script also apply certain heuristics, e.g., the length or type of the strings. Finally, it reports all cookie pairs that match these properties. For each cookie reported, we manually review and confirm the existence of a vulnerability to eliminate false positives.

In total, the script reported 22 cookie pairs of eight websites out of the 211 sites under test. However, manual investigation revealed that only three websites (nine cookie pairs) are vulnerable, i.e., GitHub, CNN, and Yahoo, accounting for 1.4% of the tested websites. For example, CNN sets a pair of duplicate cookies named `obuid` and `OB-USER-TOKEN` with the exact same value but with different SameSite cookie policies, i.e., no `SameSite` attribute and the Strict policy, respectively. Similarly, Yahoo sets duplicate session cookies with different policies, i.e., None and Lax. Finally, GitHub uses a pair of cookies named `user-session` and `-Host-user-session-same-site` with inconsistent policies, i.e., no `SameSite` policy and Strict, respectively.

6.3.2.8 Inconsistent Policy for Different User-Agents

To determine inconsistent policies based on the user-agent, we performed three web crawls on live instances of Alexa top 500K websites on June 2020, September 2020, and April 2021 using two different User-Agents for mobile and desktop clients. Accordingly, we compare if the SameSite cookie policy is set differently for the same cookie across the HTTP responses captured for desktop and mobile clients.

Table 6.8 summarizes our findings. In total, we identified 5,719, 9,215, and 9,951 vulnerable websites that allow a policy downgrade in the three web crawls, respectively. Note that not all entries in Table 6.8 may lead to a policy downgrade, i.e., pairs that have the Lax policy in one client, and do not set any policy in the other client do not lead to a policy downgrade assuming the new default policy. Finally, out of the 9,951 vulnerable websites, 138 are among the top 1K Alexa websites, showing that such inconsistencies are prevalent among popular sites. We refer interested readers to Table A.7 in Appendix A.2 which shows the number of policy inconsistencies grouped by site popularity.

Table 6.8: Summary of SameSite cookies' inconsistencies across mobile and desktop clients of Alexa top 500K websites. The total column shows the number of vulnerable sites where a policy downgrade can occur.

Crawl	(M, N)	(M, L)	(M, S)	(M, NS)	(M, I)	**Total**
June 2020						5,719
(D, N)	-	2,263	70	213	0	
(D, L)	2,382	-	244	157	0	
(D, S)	72	244	-	9	0	
(D, NS)	217	133	5	-	0	
(D, I)	0	0	0	0	-	
Sept. 2020						9,215
(D, N)	-	3,262	299	1,282	0	
(D, L)	3,167	-	381	759	0	
(D, S)	234	378	-	26	0	
(D, NS)	172	268	13	-	0	
(D, I)	0	0	0	1	-	
April 2021						9,951
(D, N)	-	3,572	328	1,166	0	
(D, L)	3,516	-	431	781	0	
(D, S)	302	432	-	35	0	
(D, NS)	135	278	33	-	0	
(D, I)	0	0	0	1	-	

Legend: D= Desktop; M= Mobile; N= None; L= Lax; S= Strict; NS= Not Set; I= Invalid.

6.4 Web Browsers and Web Frameworks

The final analysis of this chapter looks at the inconsistency between browsers when handling and enforcing the SameSite attribute properties (§6.4.1), and it looks at the default policies used by popular web frameworks (§6.4.2).

6.4.1 Evaluation of Web Browsers

Browsers exhibit a variety of behaviours when applying SameSite cookies. For example, the default policy in Chrome and Opera is Lax, whereas Firefox and Safari enforce the None policy by default. Even with regards to Chrome, the latest IOS version (87.0.4280.77) still uses the None policy by default [343]. Also, such inconsistencies not only apply to the default setting, but also to other corner cases where a request is sent in cross-site context, e.g., when `SameSite=None` cookies are used without a `Secure` flag, or when the `SameSite` attribute has an invalid or even the Lax or Strict value.

Methodology. We conducted our analysis against 14 web browsers and investigated their compliance with the new RFC 6265bis specification [28]. The list of popular browsers for testing is from MDN [175], and we add the iOS Chrome and Tor Browser. Furthermore, for Safari, we consider three different versions that are frequently used by the three recent macOS operating systems, since Safari cannot be upgraded standalone [351]. We automated the analysis by developing three webpages, two in the same origin and the third page in a different origin, where the first page performs same-site and cross-site requests from different contexts towards the second and third page, respectively. Then, the analysis of the logs of the web servers reveals which request was submitted with cookies.

Results. Table 6.9 summarizes our findings. In total, we identified seven distinct ways on how browsers enforce the SameSite cookie policy in same-site and cross-site context. We observed that, to date, none of the 14 tested browsers are fully compliant with the new RFC 6265bis specification [28], including Chrome. The most RFC-compliant browsers are Chrome, Chrome on Android, Opera, Opera on Android, and Edge, which comply for 11 out of the 12 possible cases of how the SameSite cookie attribute can be set in same-site and cross-site contexts, as shown in Table 6.9.

Table 6.9: Overview of web browser's compliance with RFC 6265bis [28]. Browsers with similar behaviours are grouped with the same color. The table highlights a total of seven distinct browsers' implementations when enforcing the SameSite cookie policy, each marked by a different color.

			Set-Cookie HTTP Header					
Browser/ Spec	**Version**	**Scope**	K=V	K=V; SS=None; Secure	K=V; SS=None	K=V; SS=Invalid	K=V; SS=Lax	K=V; SS=Strict
Specification	RFC 6265bis [28]	Same-Site	●	●	○	●	●	●
		Cross-Site	○	●	○	●	○	○
Tor Browser	10.0.12	Same-Site			✗			
		Cross-Site		✗		✗		
Chrome	89.0.4389.82	Same-Site						
		Cross-Site				✗		
Opera	74.0.3911.218	Same-Site						
		Cross-Site				✗		
Edge	89.0.774.54	Same-Site						
		Cross-Site				✗		
Firefox	86.0	Same-Site			✗			
		Cross-Site	✗		✗			
Safari	14.0.3	Same-Site			✗			
		Cross-Site	✗		✗			
Safari	12.0.3, 13.1.1	Same-Site			✗			
		Cross-Site	✗	✗		✗		
IE	11.0	Same-Site			✗			
		Cross-Site	✗		✗		✗	✗
Andr. Chrome	84.0.4147.124	Same-Site						
		Cross-Site				✗		
Andr. Opera	61.2.3076.56749	Same-Site						
		Cross-Site				✗		
Andr. Firefox	79.0.5	Same-Site			✗			
		Cross-Site	✗		✗			
iOS Safari	14.4	Same-Site			✗			
		Cross-Site	✗		✗			
Samsung Int.	13.2.1.70	Same-Site			✗			
		Cross-Site	✗		✗			
Andr. WebView	84.0.4147.105	Same-Site			✗			
		Cross-Site				✗		
iOS Chrome	87.0.4280.77	Same-Site			✗			✗
		Cross-Site	✗		✗			

Legend: K= Key; V= Value; SS= SameSite; ● = Cookie Sent; ○ = Cookie Not Sent;
✗ = Divergent From/Not Compliant with Specification.

As of today, web application developers need to be aware of all these seven behaviors if they want their website to (i) work with all these browsers and (ii) provide the same security guarantees. One way to achieve that is using user-agent-dependent SameSite policies. While this may seem a valid solution, we have seen in the past that header inconsistencies can be the root cause of vulnerabilities (see, e.g., [308] or our SSC User Agent Inconsistency vulnerabilities in §6.3.2.8).

Chrome on iOS and Lax-by-default. iOS/iPadOS browsers are required to use WebKit for rendering web pages, possibly limiting browser developers' liberty in changing the default SameSite cookies handling. However, even when browsers are required by the AppStore policy to use iOS WebKit, we observed different behaviors between iOS Safari and iOS Chrome. With `SameSite=Strict`, Safari attaches cookies only for SameSite requests (as per RFC 6265bis specification [28]) whereas iOS Chrome does not do that (Table 6.9). Based on that, we do not know whether Chrome developers have some form of liberty to control browser's behavior when handling SameSite cookies, or if Chrome developers are using a different version of iOS WebKit from the one used by Safari.

6.4.2 Evaluation of Web Frameworks

Table 6.10: Evaluation of SameSite cookie policy in top five frameworks of top five programming languages.

			SameSite Cookies		
Language	**Framework**	**Version**	**Support**	**Default**	**Reference**
Python	Flask	1.1.2	◐	Not Set	[352]
	Django	3.1.7	●	None	[309]
	Tornado	6.1	◐	Not Set	[353]
	Pyramid	2.0	●	None	[310]
	Web.py	0.62	●	None	[354]
JavaScript	Express	4.17.1	●	Not Set	[355]
	Meteor	2.1	○	Not Set	[356]
	Sails	1.4.1	●	None	[357, 358]
	Koa	6.1.0	●	Not Set	[359]
	Hapi	20.1.0	●	Strict	[360]
PHP	Laravel	8.16.1	◐	Lax	[361]
	Symfony	5.2	●	Lax	[362, 363]
	CakePHP	4.2.4	●	Not Set	[365, 364]
	Zend	1.12	●	Not Set	[366]
	Slim	4.7.0	●	Lax	[368, 367]
C#	ASP WebForms	4.7.2	●	None	[369]
	ASP MVC	4.7.2	●	None	[370]
	ASP Core	5.0	●	Not Set	[371]
	Nancy	1.4.4	○	Not Set	[372]
	Service Stack	5.1	●	Lax	[373]
Java	Spring	5.3.4	●	Lax	[374]
	Play	2.8	●	Lax	[375]
	Vaadin	8.0	○	Not Set	[376]
	Vert.x-Web	4.03	●	Not Set	[377]
	Spark	3.1.1	○	Not Set	[378]

Legend: ● = Fully Supported; ◐ = Partially Supported; ○ = Not Supported;

Even when browsers enforce a default Lax policy, web frameworks' built-in APIs can downgrade it to the None policy by default. Accordingly, we examined the top five frameworks of top five programming languages, with the overarching goal of identifying frameworks that relax the browser's default SameSite cookie policy when a cookie is set.

Methodology. First, we select the top five web programming languages based on GitHub's 2020 Octoverse report [379] (i.e., JavaScript, Python, Java, PHP, and C#). Then, we compile a list of frameworks for each language, and quantify their popularity based on a series of criteria (ordered): number of tagged questions in Stack Overflow [380], number of uses by other GitHub repositories, number of GitHub stars, forks and watches, and the number of downloads in package managers of each language. Accordingly, we pick the top five frameworks of each language, resulting in a total of 25 frameworks. Then, we resort to the documentation of each framework to see if it has built-in support for SameSite cookies. If so, we create a basic web application using the default configuration of the framework, and use the frameworks' cookie APIs to set a cookie. Finally, we run the application and investigate if the framework did set a `SameSite` attribute on the cookie by default.

Results. Table 6.10 summarizes our findings. First, out of the 25 frameworks, 21 frameworks provide built-in APIs to control the SameSite policy when setting a cookie, out of which in three frameworks, not all the three SameSite policies are supported. Then, we observe that six out of the 25 frameworks (i.e., 24%) specify the None policy by default for all cookies set. For example, when a developer uses the API `set_cookie(k, v)` in Django [309] or Pyramid [310], the framework sets the cookie `k=v; SameSite=None`, adding a None policy semi-transparently to the developer.

6.5 Summary

In this chapter, we performed, to the best of our knowledge, the first security evaluation of SameSite cookie policy, systematically covering the trend of its usage, the impact of the new default policy, and the threats against it, with the overarching goal of studying how effectively SameSite cookies can mitigate XS attacks. We quantified the prevalence of vulnerabilities of each threat in the wild, showing that (i) XS attacks can still be mounted in popular web applications leveraging requests that are not protected by the default Lax policy, thus requiring developers to be aware of the unprotected requests and the additional security risks, and (ii) even if developers use the default Lax policy correctly and as a defense-in-depth, application-level vulnerabilities, such as forgeable state-changing POST requests, or intra-page and inter-page cookie policy inconsistencies can continue to cause XS attacks, despite the presence of the Lax policy. Finally, we showed that browsers diverge when enforcing SameSite cookies, and that web frameworks' default APIs can undermine the browser's enabled-by-default Lax protection. Overall, we believe SameSite cookies are a powerful defense-in-depth that can help reduce the attack surface for XS attacks. However, their correct and secure use require developer's awareness and expertise.

7 Related Work

In this chapter, we provide an overview of the research domains closely tied to the central themes explored in this thesis. Firstly, we elaborate on research in the field of static and dynamic program analysis techniques, with a specific focus on the analysis of JavaScript programs and code properties, which serve as the fundamental building blocks for the systems that we design to study vulnerabilities on the Web platform (§7.1). Building upon this foundation, we explore prior research endeavors that leveraged these program analysis techniques to design and implement security testing solutions, and then compare the characteristics and capabilities of those solutions with our hybrid system, JAW, which plays a centeral role in this thesis. Moving forward, we discuss how these security testing solutions have been instantitated by researchers in the past, shifting our attention toward the exploration of security issues on the Web, particularly request forgery and code injection vulnerabilities. We track the historical evolution of these vulnerabilities and the research efforts dedicated to studying their characteristics, such as large-scale measurements of their prevalence and impact on the Internet, the identification of insecure coding patterns, and the recognition of vulnerability indicators (§7.2). Concluding our exploration, we delve into the research related to security mechanisms designed to safeguard the Web and their effective and consistent implementation, such as SameSite cookies and HTTP security headers like Content Security Policy (§7.3).

7.1 Static and Dynamic Program Analysis Techniques

In this thesis, we leveraged a wide array of program analysis concepts to tackle the task of uncovering vulnerabilities within client-side JavaScript programs. For example, in Chapters 3 to 5, we used static analysis to identify unvalidated data flows from program inputs to security-sensitive instructions and confirmed them using various runtime monitoring approaches like code instrumentation and dynamic forced execution. Of course, we are not pioneers in the application of such program analysis techniques to the realm of the Web (see, e.g., [381, 32, 16, 53, 3, 213]), as has been used in other domains like software and mobile security too (e.g., [383, 384, 382, 385]).

Static Program Analysis. Static analysis, a technique used in software engineering to analyze code without executing it, has evolved significantly over the years. The first scientific works on static analysis date back to the early 1970s, primarily driven by the need to ensure correctness and reliability in software, with Robert Floyd's work on program verification [386] and Donald Knuth's work on compiler theory [387]. Back then, simple techniques like syntax checking and type checking were the initial focus. One of the first major advances in static analysis came in 1977, when William Wulf and his team at Carnegie Mellon University developed the Flow Analysis System (FAS) [388]. FAS was a tool that could be used to analyze the control flow of a program and identify potential issues such as unreachable code and type mismatches. Shortly after, in 1979, Stephen C. Johnson released one of the earliest widely recognized static analysis tools, "lint," alongside Unix version 7, which was designed to check C programs for errors [389].

During the 1980s to 2000s, research on static analysis continued to advance. For example, Gary A. Kildall introduced data flow analysis to identify uninitialized variables and use-before-definition errors [390]. James C. King proposed a symbolic execution technique to symbolically execute a program and identify potential vulnerabilities such as buffer overflows and null pointer dereferences [391]. Clarke et. al. proposed a model checking approach to verify that a program satisfies a given formal specification [392]. Building upon these works, static analysis evolved with inter-procedural [393] and context-sensitive [394] approaches.

Moving forward and to the JavaScript domain, the 21st century saw the rise of a plethora of static analysis techniques, ranging from models capturing control [129, 43, 42] and data flow dependencies [44, 45] to complex type analyzers [47, 46, 48], points-to analysis [50, 49], DOM tree models [14] and models capturing run-time code evaluation [12]. While these techniques lay the foundation to build more complex approaches, they are not sufficient alone for finding the complex classes of vulnerabilities that we study in this thesis, such as client-side CSRF and DOM Clobbering. This is due to the fact that these methods are tailored for specific code analysis tasks, as discussed in §1.1.1. Detecting a vulnerability like DOM Clobbering requires understanding multiple aspects of a program at the same time, such as control flows, data flows, injection contexts and runtime behaviours, which the aforementioned models fail to provide as they create ad-hoc program representations.

To leverage the collective power of these approaches simultaneously, however, recent research proposed consolidating static models into canonical representations (e.g., [51, 15]), which is inspired by prior graph-based techniques to analyze the source code [180, 396, 395]. Notably, Yamaguchi et. al. [15] proposed the notion of CPGs for finding software bugs in C/C++ applications (i.e., a non-web-based execution environment). Backes et. al. [51] later extended this idea to detect vulnerabilities in the server-side of PHP web applications. However, these new ideas do not seamlessly adapt to JavaScript's nuances, such as the dynamic execution environment [14] and the asynchronous function calls [45]. To date, there are no established models for client-side JavaScript that offer a comprehensive foundation for both detecting vulnerabilities and conducting in-depth exploratory analysis of code. In contrast to these works, our approach adapts the concept of CPGs to the client-side of web applications, and extends them with dynamic information, i.e., state values. Furthermore, existing CPGs are poorly suited for large-scale analyses which is a needed feature to analyze web applications (a web application can have hundreds of pages to analyze, each with thousands of lines of JavaScript code). Backes et. al. [51] needed up to five days and 7 hours for a *single* query when analyzing 77M LoC. In comparison, JAW took three days (sequential execution) to model and query 228M LoC of Bitnamic applications (Chapter 3). This improvement is largely due to the introduction of the new notion of symbolic models for shared third-party code (§3.3.2). We believe that these contributions are key enablers to use graph-based analyses on web applications, at scale.

Dynamic Program Analysis. Dynamic testing is a well-known technique to analyze a program while it is running. Previous research in this area can be largely divided into black-box scanning and fuzzing techniques (e.g., [398, 397, 399]), which attempt to discover vulnerabilities by testing a list of vulnerability-inducing inputs and observing the program behaviour, and white-box testing techniques that range from code instrumentation and runtime monitoring [108, 110, 400] to more advanced methods like forced execution [401, 3], taint analysis [107, 402], and symoblic and concolic execution [53, 403].

As we will show next in §7.2, dynamic analysis techniques are invaluable for detecting or validating statically-found vulnerabilities in web applications. Nevertheless, a fundamental challenge when applying them in the Web domain lies in establishing a sound and sustainable implementation. For instance, a significant fraction of the taint analysis and symbolic/force execution engines heavily depend on modifications to web browsers (see, e.g., [107, 32, 53]). However, with the rapid evolution of browser code, such as the introduction of new features, deprecation of APIs, and continuous refactoring of the source code, any analysis closely tied to these modifications can quickly become outdated. Conversely, non-browser-based implementations, such as those relying on Jalangi [400], often lack the sophistication required to handle the intricacies of the JavaScript code or its ever-growing dynamic features. Consequently, these implementations often fall short when dealing with real-world, in-the-wild source code. We designed the dynamic testing components in our framework, JAW, with reusability in mind, offering both types of solutions for different applications. However, we note that JAW does not aim to solve these challenges, thus also not exempt from (some of) the outlined limitations above.

7.2 Security Testing of Web Applications

In this section, we review the related work in the area of security testing and program analysis for the Web. An orthogonal line of related work explored the applications of program analysis techniques outlined in §7.1 for vulnerability discovery, such as static analysis [404, 51, 405, 15], dynamic analysis [107, 8, 32, 16, 3], and hybrid approaches [381, P1, 204, 53, 213]. For example, several research efforts studied XSS [32, 52–54] and client-side input validation flaws [P1, 107, 32, 53, 2, 3, 56]. In particular, Klein et. al. [107] used a combination of dynamic taint tracking and symbolic string analysis to study the robustness of custom sanitization functions in the wild. Lekies et. al. [32] performed dynamic taint analysis to study the prevalence of client-side XSS vulnerabilities. The authors modified the JavaScript engine in Chromium and enhanced it with taint-tracking capabilities to track unvalidated data flows to security-sensitive XSS sinks. Steffens et. al. assessed the prevalence of persistent [2] and postMessage-based [3] XSS. Similarly, saxena et. al. proposed Kudzu [53], a tool that performs dynamic taint-tracking to identify sources and sinks in the current execution using a GUI explorer, and then generates XSS exploits by applying symbolic analysis to the detected source-sink data flows.

In general, these techniques could be useful to detect more advanced vulnerabilities like client-side CSRF provided their crawler can trigger the executions that are connecting sources to sinks. However, crawlers often fall short of visiting modern web UIs, providing low code coverage when compared with static analysis techniques. As opposed to approaches like [32, 53], JAW follows a hybrid approach, addressing shortcomings of JavaScript static analysis such as dynamic loading of script tags and point-to analysis for DOM elements.

Finally, other works studied the presence of script-less attacks in JavaScript programs using both static and dynamic approaches, e.g., mutation-based XSS [7] and script gadgets [8]. Our work aligns with these efforts by leveraging and combining common program analysis techniques. However, in contrast to these works, we focus on developing a general-purpose technique that can be used to detect also more sophisticated classes of taint-style vulnerabilities, such as client-side request hijacking and DOM Clobbering [204], to study such vulnerable program behaviors on the Web platform.

7.2.1 Request Forgery Vulnerabilities

Request forgery vulnerabilities have a long history and have been the the subject of numerous research endeavors in the past, e.g., SSRF [227, 406], CSWSH [210, 211, 209], CSRF [21, 16, 226, 20], and client-side CSRF [1]. Due to their nefarious consequences, research in this area has largely focused on request forgery defenses (e.g., [17, 138, 66, 158, 408, P4, 224, 70, 159, 407, 72, 71, 19]), with very few proposing detection techniques that can help security testing community to uncover CSRF exploits, i.e., dynamic analysis [16], ML-based solutions [21], and systematic manual inspection [225, 20]. Specifically, only a fraction of these works, most notably, Deemon [16], and Mitch [21], went beyond manual inspection by presenting semi-automated approaches. In this thesis, we propose JAW, a comprehensive framework that can be used for automatic detection or interactive exploration of client-side cross-site request forgery vulnerabilities (Cf. Chapter 3), as well as the broader issue of client-side request hijacking vulnerabilities in web applications (Cf. Chapter 4).

Closely related to client-side request hijacking, multiple studies considered the hijack of HTML tags such as scripts [32, 2] and iframes [83]. In contrast to HTML tags, this thesis focuses on JavaScript APIs that allow creating and sending network requests. Instead of hijacking requests, previous research also explored injecting new requests thorough DOM manipulations and dangling markup injections [410, 409, 78]. Our study complements the missing pieces of these works by proposing client-side CSRF (i.e., [P1]) and extending it to the larger issue of request hijacking (i.e., [P2]), quantifying their prevalence and impact in the wild.

7.2.2 HTML-only Injection Vulnerabilities

Reusing the webpages' legitimate JavaScript code to obtain arbitrary client-side code execution have been the topic of several research efforts in the past. Most notably, Lekies et. al. [8] described a new attack where small fragments of JavaScript code, known as *script gadgets*, are unexpectedly executed as a result of a non-script markup injected by attackers. The authors used a modified browser engine [32] to measure the prevalence of these gadgets, and demonstrated that they are prevalent and can bypass existing XSS mitigations, such as HTML sanitizers [78] and CSP [238, 92]. Later, Roth et. al. [411] quantified the impact of script gadgets on CSP in the wild. Similarly, Heiderich et. al. [7] discovered mutation-based XSS attacks (mXSS), showing how specific browser-based mutations of DOM content and insecure JavaScript that reads and rewrites HTML elements can transform initially secure DOM markup to code. While all these three attacks can transform non-script markup to executable code, the elements enabling DOM Clobbering is largely different, i.e., script gadgets rely on event handlers and mXSS attacks abuse `innerHTML` mutations, whereas DOM Clobbering is the result of a complex interplay of the default browser behaviors and insecure use of named property accesses in JavaScript programs. Contrary to these works, our study focuses on DOM Clobbering, systematically testing mobile and desktop browsers, identifying insecure coding patterns using both static and dynamic analysis techniques, and demonstrating their exploitability.

Multiple instances of DOM Clobbering vulnerabilities have been discovered in the last 12 years by both academics [412, 78, 83] and security analysts [88, 242, 89, 10, 9], with the first public instance identified in 2010 by Rydstedt et. al. as a way to circumvent frame busters [83]. The term 'DOM Clobbering' itself emerged in 2013, when Gareth Heyes [10] demonstrated how this class of vulnerabilities can escalate to client-side code execution. Due to such nefarious consequences of DOM Clobbering, prior academic studies has primarily focused on its defenses (e.g., [93, 78, 236]). Most notably, Heiderich et. al. proposed the JSAgents library [93] and later the DOMPurify sanitizer [78] to mitigate the security implications induced by markup injection, such as DOM Clobbering and client-side XSS [32, 54]. Our research completes the missing pieces of these works by systematically studying DOM Clobbering attack techniques, their prevalence, and effectiveness of the existing countermeasures.

7.3 Security Mechanisms for the Web

Previous research on Web security mechanisms centers around studying the correct implementation and effectiveness of already deployed mechanisms, such as Content Security Policy (CSP), or proposing new defenses that could mitigate various classes of Web attacks, such as XS attacks and request forgery discussed throughout this thesis. For example, multiple works studied the inconsistent deployment of HTTP security headers between the desktop and mobile variants of a website [308, 319]. Calzavara et. al. [413] proposed extensions to CSP to enable third-parties to contribute in creating a secure CSP configuration through advanced policy composition strategies. Roth et. al. [414] conducted a developer study to understand the challenges developers encounter when deploying a CSP. Other works addressed incoherencies in browser access control policies and SOP (see, e.g., [416, 133, 415, S2]). More closely related to the attacks presented in §6.3, Calzavara et. al. studied the inconsistent adoption of security mechanisms across different web pages of an application [307]. As opposed to these works, in this thesis, we uncovered inconsistencies of the browsers with regards to the SameSite cookie policy. Also, we studied the protective coverage of the new default Lax policy, and systematically identified and proposed attacks that can bypass it, primarily based on incoherencies in the `SameSite` cookie attribute.

A orthogonal line of the prior work studied the role of cookies in Web security, such as cookie integrity attacks (e.g., [417, 419, 420, 418, 156]), and third-party cookies (e.g., [422, 421]). As a response to the notorious role of third-party cookies in XS attacks, previous research proposed multiple approaches to automatically strip session cookies from cross-site requests, e.g., using server-side proxies [138, 231], browser extensions [66, 159, 407, 72], or both [224]. Franken et. al. [316] proposed a framework to evaluate the correct enforcement of these cookie stripping policies on cross-site requests, by analyzing the security mechanisms of browsers and browser extensions. Other works studied the usage of cookie security attributes. For example, Sivakorn et. al. explored the adoption of the `Secure` flag [136], and Singh et. al. measured its usage [415]. Similarly, Zhou and Evans studied the usage of the `HTTPOnly` attribute [423]. In contrast to these works, in this thesis, we focus on the `SameSite` attribute. Closely related to our work, Calvano [96, 424] analyzed the usage of SameSite cookie policies using HTTP archive. Similarly, the proprietary BuiltWith website reports the usage of SameSite cookies [425]. Our work completes the missing pieces from these analyses, systematically studying the trend of the adoption of valid and invalid SameSite cookie policies, and the impact and effectiveness of Lax-by-default cookies.

8 Concluding Remarks

This chapter provides concluding insights and a discussion of key aspects of our work. First, we discuss the ethical safeguards, considerations for experiment execution, and measures taken for vulnerability notifications concerning our study's findings (§8.1). Then, we outline the limitations of the approaches adopted in this thesis and their connection to recent research (§8.2). Building upon our main findings and identified pain points, we highlight and discuss open challenges for future work (§8.3). Following that, we demonstrate our commitment to open science by discussing our open-source research artifacts (§8.4). Finally, we present concluding remarks summarizing the research presented throughout this thesis and discuss the broader implications of our findings (§8.5).

8.1 Ethical Considerations

Our experiments on live websites do not target any real user. Tests requiring to persist data (e.g., store a markup) or state-changing operations (e.g., changing profile settings) are exclusively restricted to user accounts that we created on those sites. Also, to uphold the highest ethical standards, we excluded testing functionalities where we could not control the request impact or the visibility of the injected payload (e.g., publicly accessible posts and comments). Tests on public functionalities was performed without persistently injecting any markup. Throughout the testing process, we strictly adhered to best practices and guidelines outlined by websites' vulnerability disclosure programs on platforms like HackerOne [426] and BugCrowd [427], whenever such programs were available. Furthermore, we minimized the crawling load on resource servers by limiting the number of pages we visited per site (i.e., maximum 200) and also by distributing the load in time (round-robin strategy).

In this thesis, we test and measure the occurence of vulnerabilities in real systems. In all cases, we responsibly disclosed our findings to the affected parties, including web browsers, sanitizer libraries, web frameworks, IdPs and websites, following the best practices of vulnerability notification (see [428]). We prioritized our reports by severity, where we send an initial notification that includes the vulnerability details, or a proof-of-concept exploit, followed by an additional reminder every month to maximize the remediation rate. Since many of the security issues we found in Chapters 4 to 6 affected hundreds of websites, we created large-scale vulnerability notification campaigns, seeking the assistance of our national CSIRT[1]. We do not disclose the details of vulnerabilities unless they have been confirmed and patched by the affected vendor. As a result of our disclosure process, many sites patched their system, including popular ones like Microsoft Azure, GitHub, Starz, Vimeo, Fandom, TripAdvisor and SuveryMonkey, to name only a few examples. In addition, we even submitted patches ourself to the affected libraries, such as the famous DOMPurify sanitizer [429], maintaining the highest ethical standards throughout our research.

8.2 Limitations

In this section, we discuss potential limitations that affect our experiments presented throughout this thesis. First, our measurements on client-side CSRF, DOM Clobbering, and request hijacking vulnerabilities rely on our ability to represent JavaScript behaviour in terms of unvalidated data flows, largely via static analysis. However, JavaScript programs are incredibly challenging to be analyzed via static analysis due to their dynamic nature (e.g., [11, 12]). Second, our analyses on SameSite cookies, DOM Clobbering and request hijacking flaws are confined to the public surface of the investigated websites, as automated login and registration is a challenging task [318]. In the remainder of this section, we discuss each limitation in more detail.

[1] `https://www.trusted-introducer.org/`

8.2.1 Soundness of Static Analysis

The vulnerabilities found in this work are those captured by our model and traversals. However, it could happen that a vulnerability in the program (e.g., a forgeable request) is not found because the construction of the model is bound by the soundness properties offered by the individual static analysis tools we use for the construction of the property graph (e.g., CFG, PDG, etc). Accurately building these models by static analysis is a challenging task due to the streaming nature of JavaScript programs [11], and JavaScript dynamic code generation capabilities. We point out that, similarly to prior work (e.g., see [12]), JAW extracts the code executed by dynamic constructs, i.e., `eval`, `setTimeout` and `new Function()`, as long as the string parameter can be reconstructed statically. As a future work, we plan to incorporate a modified JavaScript engine (e.g., VisibleV8 [430]) into JAW, to provide better support for reflection and such dynamic constructs, and to minimize the potential side effects of function hooking, especially with respect to event handlers.

We observed that using dynamic state values together with traditional static analysis will help to remove spurious execution traces (§3.3.7). Nevertheless, our extensive manual verification uncovered that 1/516 requests in §3.3 was a false positive due to inaccurate pointer analysis of the `this` statement in dynamically called functions (see §3.3.3). We observed that such a request is using data values originating from the DOM tree, meaning that 1/83 requests of the `DOM-READ` forgeable request category may be a false positive. We addressed this shortcoming by incorporating a call-sensitive resolution of the `this` keyword into subsequent versisons of JAW (i.e., JAW-v2). To counter the limitations of static analysis, in this thesis, we leverage dynamic analysis and runtime monitoring to complement static analysis. For example, as we showed in §4.3, dynamic information plays a crucial role in identifying 67.3% of the discovered request hijacking data flows.

8.2.2 Web Crawling

The vulnerabilities discovered in this thesis affect those pages that JAW reached with our crawler. However, crawling is a challenging task (see, e.g., [431, 432]) and JAW may have missed pages with vulnerable code, as our crawlers only visit a portion of the websites under test. In a more concrete sense, we constrain our crawlers to visit a maximum number of 200 subpages of each site instead of exhaustive crawling attempts to manage the load on resource servers. Furthermore, our crawlers adopt a passive approach, merely visiting webpages without executing any actions that could potentially trigger execution of additional JavaScript code, which is contrary to other crawlers like BlackWidow [397].

Finally, in this thesis, we did not evaluate *at scale* pages after the login step accurately. To the best of our knowledge, Cookie Hunter [318] is the only recent approach able to handle the sign-up and sign-in automatically. However, the sign-up success rate is quite unsatisfactory, with 88% fail rate in creating accounts. In addition, Cookie Hunter relies on pattern matching which is too brittle to minor changes in the UIs, requiring creating and maintaining new patterns throughout the (longitudinal) analysis. For these reasons, we demonstrated the applicability of JAW at a smaller scale. For example, to study client-side CSRF vulnerabilities, we created Selenium login scripts for all (i.e., 106) bitnami applications. In addition, we evaluated SameSite adoption on a smaller scale by creating ad-hoc login scripts, focusing on the protected pages on the Top 500 sites having the login functionality (211 sites). We observed that 88% of the sites that do not use SameSite for their cookies on the home page also do not use it on their protected pages, further supporting the generalizability of our results.

To increase coverage, JAW provides support for the smooth integration of other crawlers. We emphasize that our results should be interpreted as lower-bound estimates of the actual threats posed by the studied vulnerabilities.

8.3 Open Challenges and Future Work

In this section, we discuss some of the open questions and new challenges that arise from our findings. First, in Sections 8.3.1 to 8.3.4, we focus on several open challenges and future work concerning the testing techniques to detect vulnerabilities at scale. Then, in Sections 8.3.5 and 8.3.6, our focus shifts to the application of these techniques, presenting new security problems and aspects that warrant further scrutiny.

8.3.1 Analysis of Shared Code in Web Applications

In this thesis, we showed that JAW can reduce by 60% the effort required to analyze client-side JavaScript programs via pre-built symbolic models. However, when looking at the unique application code, we observe that a large fraction of code is also shared between pages. For example, the 4,836 collected pages of Bitnami applications in Chapter 3 contain in total 104,720 scripts, of which only 4,559 are unique, suggesting that the shared code of different webpages can be modeled once, and reused through incremental program analysis to streamline the testing at scale.

Developing such an incremental static analysis technique poses challenges, necessitating alterations to the design and implementation of existing algorithms that generate program representations, including Program Dependence Graphs, Control Flow Graphs, and Call Graphs. This adaptation must (i) sufficiently maintain the soundness properties of static analysis models, and (ii) ensure that the effort required to construct these models for the entire webpage, starting from pre-built models of shared code, is no greater than the effort needed to analyze the webpage from scratch. We plan to address this problem in future work.

8.3.2 Testability Patterns and Automatic Transformation

Testability patterns are problematic code instructions that affect the capability of code analysis tools for accurate detection of vulnerabilities, such as hindering their detection or leading to false alarms, as shown by recent research [127]. One avenue for improvement is the automatic transformation of these patterns into simpler code structures that are more amenable to static analysis. For instance, consider the transformation of dynamic function calls, where the function name is provided as a string, into conventional function calls. This conversion can enhance the capabilities of static analyzers by simplifying the code, making it more conducive to thorough analysis. However, JavaScript programs can dynamically invoke functions through various features, including the prototype chain and event-driven calls, adding an additional layer of complexity to such transformations.

The primary challenges to realizing this approach lie in automatic identification of testability patterns as well as how transformations should look like for each pattern, ensuring the preservation of semantics during the transformation process. For example, transforming dynamically generated JavaScript code that uses code constructs like `eval()` is a challenging task [12]. Overall, this approach aims to enhance the testability of code, ultimately leading to more reliable and accurate results from security testing tools.

8.3.3 Automatic Assessment of Static Data Flows

Despite numerous advancements in static analysis, the verification of the analysis results remains a significant concern, especially considering the large number of alerts that are prone to false positives. In these cases, manual analysis is often necessary to investigate and validate these alerts, as shown in Chapters 3 to 5. To help confirming the presense of data flows automatically, in this thesis, we employed dynamic analysis techniques, such as code instrumentation and runtime monitoring (see, e.g., §4.2.4). However, dynamic analysis introduces additional challenges, including the automatic generation of suitable inputs for testing and coverage of various execution paths, such as conditional branches, which limit our ability to automatically reason about the associated risk for a significant fraction of the results.

An alternative approach for future work involves risk-based assessment of results using machine learning. For instance, a machine learning model can cluster similar data flows and employ classification techniques to assign a risk score to each flow, pruning potential false positive results (e.g., data flows with sanitization instructions). By incorporating machine learning algorithms, we can automate the assessment of the severity of potential vulnerabilities, which can streamline manual analysis efforts, allowing human analysts to prioritize investigations based on the alerts with the highest associated risk. Moreover, the integration of machine learning provides a scalable solution, enabling organizations to manage security assessments across a larger number of webpages and services.

Despite challenges with respect to the exact machine learning techniques to be used, such as translating insecure data flow logic into embeddings, creating a model necessitates a dataset comprising true positive and true negative samples. However, creating a representative dataset starting from the results of static analysis is challenging as it requires manual investigation of data flows for thousands of webpages. In Chapter 4, we have already collected a similar dataset for client-side JavaScript using dynamic analysis, where we visited webpages using a taint-aware browser. However, there are open questions about how well a model trained on dynamic data flows would perform when applied to static analysis considering various static analysis tools and their outputs. Moreover, the transferability of the results to programming languages beyond JavaScript is also a subject of consideration. Finally, answering these questions require a ground-truth dataset for model evaluation, presenting a challenge in itself. Overall, creating such a risk-based approach is challenging but could represent a strategic advancement in optimizing security workflows and enhancing the overall resilience of web applications against potential threats.

8.3.4 Web Crawling and Deep Application States

One of the fundamental steps of vulnerability discovery is crawling and data collection. However, crawlers often fall short of reaching deep application states as well as handling complex workflows such as following login and account registration procedures. For example, as outlined in the limitations of this thesis in §8.2, post-login pages were not captured by our large-scale measurements. To increase coverage over deep application states, one can leverage static analysis, possibly in combination with the power of large language models for text processing, to guide the crawling agent with feedback from webpages. Feedback information has proven valuable in other closely related domains, particularly fuzzing [398, 399].

A feedback-guided crawler that leverages static analysis to determine its next actions offers a strategic approach to enhance crawling coverage, especially in UI-intensive web applications. By utilizing static analysis, the crawler can intelligently decide actions based on event handlers associated with buttons and links, allowing it to navigate through intricate UI interactions. For example, such an approach can prove invaluable in identifying unique URLs or request parameters not found by typical crawlers, and in reaching deep application states, such as progressing through the steps of purchasing a product on a website. However, a notable challenge to realizing such an approach lies in maintaining an optimal crawling speed, as full-fledged static analysis, like the CPG generation presented in this thesis, can be computationally intensive. Perhaps, having an incremental static analysis approach, as suggested in §8.3.1, could potentially improve the situation. Striking a balance between thorough static analysis and efficient crawling speed remains a critical consideration for the effectiveness of this solution.

8.3.5 Characterization of Vulnerable Scripts

An interesting and orthogonal aspect to this thesis pertains to characterizing the first-party and third-party scripts responsible for vulnerabilities, such as client-side CSRF, request hijacking and DOM Clobbering vulnerabilities discussed in Chapters 3 to 5. Exploring these scripts could offer insights into the root causes of issues, possibly contributing to the development of more effective and compatible defenses. This is particularly important as our analyses of existing countermeasures in Chapter 6 and sections 4.4 and 5.3 suggest that they fall short in many situations.

Unfortunately, as of now, we still lack a clear understanding of first-party or third-party scripts introducing the vulnerable sinks, raising questions about whether vulnerabilities stem from developer mistakes, or, for instance, are introduced by established advertising providers or third-party frontend frameworks. For example, Lekies et. al. [8] found that script gadgets are prevalent in well-known JavaScript libraries, and Squarcina et. al. [433] discovered that frontend frameworks like Angular set custom HTTP headers for CSRF protection using attacker-controlled values (e.g., cookies from a subdomain). In the context of client-side JavaScript programs, characterizing such behavior is non-trivial, as according to recent research [434], over 40% of the applications in the wild use bundled scripts to optimise page loading speeds, making it challenging to attribute a specific line of code to either a first-party or third-party program.

8.3.6 The Unexpected Dangers of Code-less HTML Markups

The rapid evolution of client-side technologies has led to unforeseen interactions between JavaScript programs and their execution environment, particularly within browsers. These interactions can transform initially secure input markups into executable code, providing attackers with avenues for launching XSS attacks by exploiting the already-existing JavaScript code inside webpages. In Chapter 5, we investigated the problem of DOM Clobbering, representing a specific instance of the broader issue of code-less injection attacks. Previous research has also introduced various instances of such attacks, including script gadgets [8, 411], mXSS vulnerabilities [7, 32], and CSS-only XSS attacks [36]. Despite these individual explorations, we still lack a systematic and comprehensive study targeting these class of attacks as a whole in order to (i) understand the injection points in the DOM tree, (ii) the level of trust developers place in DOM content, and (iii) how this content is accessed and utilized in security-sensitive instructions. Addressing these aspects would contribute to a more thorough understanding of code-less injection attacks, enhancing our ability to design more effective defenses to mitigate associated vulnerabilities.

8.4 Open Science and Websites

To support the future research effort, we open-source the artifacts presented in this thesis. Specifically, we publicly release three versions of JAW[2] enabling security testing of client-side JavaScript programs at scale, including the detection and study of client-side CSRF, DOM Clobbering, and request hijacking vulnerabilities. In addition, we create and release a wiki consolidating our findings on SameSite cookie policies[3], including attacks that bypass the Lax-by-default protection. Finally, we release the automated browser testing pipeline[4] that identifies DOM clobbering markups (see §5.1), and an interactive catalog of clobbering markups[5]. We believe that making our artifacts available to the general public could assist developers

[2]https://github.com/SoheilKhodayari/JAW/releases
[3]https://soheilkhodayari.github.io/same-site-wiki
[4]https://github.com/SoheilKhodayari/DOMClobbering
[5]https://domclob.xyz

and security analysts in identifying security vulnerabilities in their systems and allows for reproducibility in the future.

8.5 Conclusion and Discussion

In this thesis, we proposed a security testing framework, JAW, to study the emerging security threats and defenses on the Web at scale. We demonstrated the efficacy and practicality of JAW by evaluating it on all (i.e., 106) bitnami applications, and the top 10K websites, covering multiple classes of vulnerabilities. We used JAW as our vehicle to conduct security testing at scale, where we assessed the prevalence and impact of client-side CSRF, DOM Clobbering, and request hijacking vulnerabilities in the wild, demonstrating an alarming threat landscape. Finally, we had a look at the adoption and adequacy of browser-based countermeasures, such as SameSite cookies, CSP, COOP and COEP. In the remaining, we summarize the main findings of this thesis for each research question outlined in §1.1 and discuss their wider implications.

8.5.1 Automatic Detection of Client-side CSRF

Hybrid Property Graphs. In this thesis, we proposed HPGs, a uniform representation for client-side of web applications, modeling both static and dynamic program behaviors. Contrary to CPGs, HPGs can capture the event-based transfer of control via the event registration, dispatch, and dependency graph, enabling the identification of +89% edges transferring the control flow (Cf. §3.3.2). In addition, HPGs introduce the concept of semantic types and symbolic models. By generating reusable symbolic models of shared libraries (Cf. §3.3.2), our approach reduces by more than half (-60.3%) the effort required to create program representations. Finally, HPGs provide dynamically observed information, such as environment properties and network messages.

Contribution of Run-time Monitoring. Our evaluation in §3.3 shows that dynamic information increases the transfer of control path by 0.26%. Despite its negligibility, our results shows that dynamic information is fundamental for the identification of the forgeable requests of 14 out of 87 vulnerable applications and three out of seven exploitable applications (an increase of +19.1% and +75%, respectively).

Properties of Client-side Forgeable Requests. In Chapter 3, we showed that 82% of the web applications have at least one web page with a client-side forgeable request that can be exploited to mount CSRF attacks, suggesting that forgeable requests are prevalent. We also showed that client-side CSRF can be used to mount other attacks, such as XSS and SQLi, which cannot be mounted via the traditional attack vectors. Then, the analysis of forgeable requests (Cf. §3.3.4) suggest that some client-side CSRF patterns are more prevalent than others, e.g., in 28.7% of vulnerable applications, the attacker can overwrite a parameter in the request body.

Interesting Properties of Vulnerable Applications. We found that 39 out of 106 targets in our testbed in Chapter 3 are single page applications (SPA), i.e., 36.7%. We manually examined the 87 vulnerable targets and observed that 44.8% of them are SPA's. Also, we found *exploits* in 17.9% of the tested SPAs (§3.3.5). We believe this sheds light into the fact that client-side CSRF instances are more prevalent among SPA applications.

Vulnerability Originates from the Same Code. In §3.3, the manual analysis of the 515 forgeable HTTP requests reveals that each vulnerability originates from different copies of the same code used across various pages. The templates for vulnerabilities range between one to four per application, with a majority of applications (i.e., 78.1%) having only a single template. These facts suggest that developers tend to repeat the same mistake across different pages.

8.5.2 Studying Request Hijacking Vulnerabilities in the Wild

Client-side CSRF Only Tip of the Iceberg. In this thesis, we have shown that client-side CSRF is only one facet of the larger problem of request hijacking in web applications. In fact, a considerable fraction of request hijacking data flows that we discovered (36.1%, i.e., 73.3K out of 202K) as well as more than half of the exploits that we created (Cf. §4.3.4) leverage the new vulnerability types and variants which have not been considered by previous works on client-side CSRF [1, P1, 69]. For example, we observed that over 21% of forgeable data flows affect the `sendBeacon` API [6].

Request Hijacking Data Flows Ubiquitous. Request hijacking data flows are pervasive in today's web, affecting over 9.6% of the websites in the wild and about 5.2% of the tested webpages in our dataset. Our measurement provides only a lower-bound estimate of the vulnerable data flows as we limited our tests to 50 unique pages of each website (Cf. §4.3.1).

Request Hijacking has Diverse Consequences. Our work uncovers the diverse range of security implications resulting from client-side request hijacking, where each vulnerability could be exploited in multiple ways depending on the affected request type and API, amplifying the associated risks. For example, we show that the hijack of asynchronous requests not only results in client-side CSRF, as outlined in previous research [P1], but also exposes the risk of information leakage, e.g., when attackers gain control over the endpoint of a request that contains sensitive information in its body. We observed that over 41% of the exploitable data flows (Cf. §4.3.4) could lead to client-side XSS and information leakage, and 25.3% and 22.4% lead to client-side CSRF and open redirections, respectively.

Existing Defenses Necessary but Insufficient. The analysis of existing countermeasures (§4.4) suggests that they are a necessary protection mechanism to prevent classical attacks (e.g., CSRF), but do not provide a complete protective coverage as each can only mitigate a fraction of the resulting attacks. For example, CSP does not mitigate over 41% of the XSS and information leakage exploitations of request hijacking, and is ineffective against CSRF exploitations, and COOP cannot prevent ∼93% of the discovered request hijacking vulnerabilities. In the absence of a full-fledged browser-level defense, developers have to be particularly careful when choosing or implementing a countermeasure, in order to balance security with usability. For example, over 9.6% of the applications have insufficient, missing or logically flawed input validation checks when offering their functionality.

8.5.3 Understanding DOM Clobbering Attacks and Defenses

Clobbering Markups Come In Many Forms. In this thesis, we proposed a systematic technique to identify DOM Clobbering markups, and showed that they come in many forms, with a total of 31,432 attack markups that rely on five different techniques, including 148 new instances and 30,803 new variants. We observed that browsers exhibit divergent behaviours when handling named properties. For example, for a significant fraction of the markups (i.e., 99%), there is at least one browser that disagrees with others, making it increasingly more challenging to enforce robust defenses.

DOM Clobbering is Ubiquitous. DOM Clobbering vulnerabilities are prevalent, affecting over 9.8% of the top 5K sites, with the consequences ranging from XSS to user state manipulation, request forgery and client-side open redirects in the majority of the cases, i.e., 83.7% (see §5.2.2).

Defenses Helpful but May not Completely Cut it. The evaluation of existing DOM Clobbering countermeasures (§5.3) suggests that each can only mitigate a fraction of the attacks. For example, 55% of the popular HTML sanitizers are vulnerable to at least one of the 31K clobbering markups by default, and CSP cannot mitigate over 85% of the identified vulnerabilities. Protecting such a fraction of the attack surface without switching named properties off completely is a more costly task, requiring developers to be aware of corner case behaviors of browsers and revisit the design and implementation of their systems, e.g., strict type checking, explicit variable declarations, or namespace isolation.

8.5.4 Studying the Effectiveness of SameSite Policies

The Hidden Costs of Pre-packaged Policies. In Chapter 6, we quantified a significant fraction of the attack surface that remains unprotected by the SameSite policy and exposed to XS attacks. Protecting such a fraction of the attack surface is a considerably harder and more costly task, requiring developers to revisit the design and implementation of their systems (e.g., removing state-changing GET) and being aware of both precise corner case behaviors of browsers and web frameworks. To date, developers must adapt their existing web applications to three predefined sets of admitted contexts, which is in stark contrast with other web security policies (e.g., CORS and CSP) where developers have fine-grained options to customize a security policy to their needs. We believe that such flexibility and customization could help developers fully protect their web applications. We hope that our work encourages researchers to take on the challenges of going beyond static, pre-packaged policies and exploring more flexible and customizable SameSite policies.

Correct and Secure Use Require Awareness. While the Lax-by-default policy is a relatively new mechanism that could help protect from XS attacks, it requires developers to know the precise cross-site request contexts that are and are not protected. In this thesis, we identified six cross-site contexts that are not covered by the Lax policy (Table 2.1), which are exposed to XS attacks. For example, we observed that over 10.3% of state-changing operations in Alexa top 1K sites use the GET method, and 2.6% of them can be trivially exploited to mount a CSRF attack.

Advertisement Services Affected the Most. Our functionality breakage analysis showed that as of February 2021, 19% of cross-site requests without the SameSite attribute are no longer working, affecting the most the advertising services (77.5%).

SameSite for Defense in Depth. Switching to the new Lax policy requires further care by developers as even the contexts covered by the Lax policy can be still be abused for XS attacks. For example, we showed that 1.5% of POST requests of top 1K Alexa sites that are seemingly protected by Lax can be successfully forged by replaying the request with the GET HTTP verb that is not protected by Lax. Also, SameSite policies should be used consistently across pages and across website versions to avoid introducing security gaps.

Browsers Diverge on SameSite Cookie Policy. Our analysis of 14 different web browsers uncovered seven distinct types of behaviours when enforcing SameSite cookies. These divergent enforcements may urge application developers to implement ad-hoc solutions to handle cookies of different user clients differently, e.g., by dynamically generating the SameSite policy per client, or by setting duplicate cookies, one for each intended client. For example, we discovered that the Lax policy can be bypassed in 1.4% and 3.3% of the tested top 500 sites due to cookies with inconsistent SameSite policies, either within a web page (duplicate cookies), or across multiple pages, respectively. We believe such divergence will narrow down over time.

Change of the Browser's Flag is the First Step. SameSite cookies are a robust defense-in-depth mechanism against some classes of XS attacks. However, developers needed to opt-into its protections by explicitly specifying a `SameSite` attribute. Accordingly, changing the default browser's SameSite cookie flag to Lax helps transition from an "opt-in" to an "opt-out" solution. While such change is a promising first step, it is not enough to complete this transition. For example, we observed that 24% of the top 25 web frameworks set the None policy by default when a cookie is set, which can downgrade the browser's default SameSite cookie policy, and requires developers to explicitly opt-into stricter policies. In addition, external functionalities, such as the integration of the application to third-party services, may be leveraged to compromise the Lax protection, and thus need to be reviewed. For example, for the top 10K Alexa sites, the Lax policy can be trivially bypassed in over 49% of the sites due to their integration with vulnerable identity providers.

Bibliography

Author's Papers for this Thesis

[P1] Khodayari, S. and Pellegrino, G. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In: *USENIX Security Symposium*. 2021.

[P2] Khodayari, S., Barber, T., and Pellegrino, G. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web. In: *IEEE S&P Symposium*. 2024.

[P3] Khodayari, S. and Pellegrino, G. It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses. In: *IEEE S&P Symposium*. 2023.

[P4] Khodayari, S. and Pellegrino, G. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In: *IEEE S&P Symposium*. 2022.

Other Papers of the Author

[S1] Likaj, X., Khodayari, S., and Pellegrino, G. Where We Stand (or Fall): An analysis of CSRF defenses in Web Frameworks. In: *International Symposium on Research in Attacks, Intrusions and Defenses*. 2021.

[S2] Sudhodanan, A., Khodayari, S., and Caballero, J. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In: *Network and Distributed Systems Security Symposium*. 2020.

Other references

[1] *Client-side CSRF*. `https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/`. 2018.

[2] Steffens, M., Rossow, C., Johns, M., and Stock, B. Don't trust the locals: investigating the prevalence of persistent client-side cross-site scripting in the wild. In: *NDSS*. 2019.

[3] Steffens, M. and Stock, B. PMForce: Systematically Analyzing postMessage Handlers at Scale. In: *CCS*. 2020.

[4] Stock, B., Johns, M., Steffens, M., and Backes, M. How the web tangled itself: uncovering the history of client-side web (in)security. In: *USENIX Security Symposium*. 2017.

[5] Push API Specification, W3C Working Draft (2023).

[6] Beacon, W3C Working Draft (2023).

[7] Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., and Yang, E. Z. mXSS Attacks: Attacking Well-secured Web Applications by Using innerHTML Mutations. In: *CCS*. 2013.

[8] Lekies, S., Kotowicz, K., Groß, S., Vela Nava, E. A., and Johns, M. Code-reuse attacks for the web: breaking cross-site scripting mitigations via script gadgets. In: *CCS*. 2017.

[9] Heyes, G. DOM Clobbering strikes back (2020). `https://portswigger.net/research/dom-clobbering-strikes-back`.

[10] Heyes, G. DOM Clobbering (2013). `http://www.thespanner.co.uk/2013/05/16/dom-clobbering/`.

[11] Guarnieri, S. and Livshits, B. GULFSTREAM: Staged Static Analysis For Streaming JavaScript Applications. In: *Proceedings of the 2010 USENIX conference on Web application development*. 2010.

[12] Jensen, S. H., Jonsson, P. A., and Møller, A. Remedying the Eval that Men Do. In: *ACM ISSTA*. 2012.

[13] Richards, G., Lebresne, S., Burg, B., and Vitek, J. An Analysis of the Dynamic Behavior of Javascript Programs. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2010.

[14] Jensen, S. H., Madsen, M., and Møller, A. Modeling the HTML DOM and Browser API in Static Analysis of Javascript Web Applications. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*. 2011.

[15] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In: *IEEE S&P Symposium*. 2014.

[16] Pellegrino, G., Johns, M., Koch, S., Backes, M., and Rossow, C. Deemon: Detecting CSRF with dynamic analysis and property graphs. In: *ACM CCS*. 2017.

[17] Barth, A., Jackson, C., and Mitchell, J. C. Robust defenses for cross-site request forgery. In: *ACM CCS*. 2008.

[18] Ferguson, D. Netflix Cross Site Request Forgery Vulnerability. *SecList Full Disclosure Mailing List* (2006). `https://seclists.org/fulldisclosure/2006/Oct/316`.

[19] Zeller, W. and Felten, E. W. Cross-Site Request Forgeries: Exploitation and Prevention. In: *Princeton University*. `https://www.cs.utexas.edu/~shmat/courses/cs378/zeller.pdf`. 2008.

[20] Sudhodanan, A., Carbone, R., Compagna, L., and Dolgin, N. Large-scale analysis & detection of authentication cross-site request forgeries. In: *2017 IEEE European Symposium on Security and Privacy*. 2017.

[21] Calzavara, S., Conti, M., Focardi, R., Rabitti, A., and Tolomei, G. Mitch: a machine learning approach to the black-box detection of csrf vulnerabilities. In: *IEEE EuroS&P Symposium*. 2019.

[22] *XMLHttpRequest API*. `https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest`.

[23] Fetch Living Standard (). `https://fetch.spec.whatwg.org`.

[24] *Bitnami Application Catalog*. `https://bitnami.com/stacks`.

[25] *DOM clobbering*. `https://portswigger.net/web-security/dom-based/dom-clobbering`.

[26] Jenkins, N. Sanitising HTML – The DOM Clobbering Issue (2015). `https://fastmail.blog/advanced/sanitising-html-the-dom-clobbering-issue/`.

[27] West, M. Content Security Policy Level 3. *W3C Working Draft* (2022). `https://w3c.github.io/webappsec-csp/`.

[28] Cookies: HTTP State Management Mechanism (2020).

[29] *SameSite cookie attribute, Chromium, Blink.* `https://www.chromestatus.com/feature/4672634709082112`. 2020.

[30] West, M. Incrementally Better Cookies (2019).

[31] *Top Website Statistics For 2023.* `https://www.forbes.com/advisor/in/business/software/website-statistics`. (Visited on 12/12/2023).

[32] Lekies, S., Stock, B., and Johns, M. 25 million flows later: large-scale detection of DOM-based XSS. In: *ACM CCS*. 2013.

[33] Hanna, S., Shin, R., Akhawe, D., Boehm, A., Saxena, P., and Song, D. The emperor's new apis: on the (in) secure usage of new client-side primitives. In: *Web 2.0 Security and Privacy*. 2010.

[34] Masas, R. *Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends.* 2018. URL: `https://www.imperva.com/blog/facebook-privacy-bug/`.

[35] Janc, A. and West, M. *How do we Stop Spilling the Beans Across Origins*. 2018.

[36] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless attacks: stealing the pie without touching the sill. In: *Proceedings of the ACM conference on computer and communications security*. 2012.

[37] Son, S. and Shmatikov, V. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In: *Proceedings of the Network and Distributed Systems Security Symposium*. 2013.

[38] Käfer, K. Cross site request forgery. In: *Hasso-Plattner-Institut, Technical report*. 2008.

[39] Hardy, N. The confused deputy: (or why capabilities might have been invented). In: *ACM SIGOPS Operating Systems Review*. 1988.

[40] *Intent to implement and ship: cookies with SameSite by default.* `https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/AknSSyQTGYs/SSB1rTEkBgAJ`. 2019.

[41] *Intent to implement: Cookie SameSite=lax by default and SameSite=none only if secure.* `https://groups.google.com/g/mozilla.dev.platform/c/nx2uP0CzA9k/m/BNVPWDHsAQAJ`. 2019.

[42] Park, C. and Ryu, S. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity (Artifact). In: *Proceedings of the European Conference on Object-Oriented Programming*. 2015.

[43] Jensen, S. H., Møller, A., and Thiemann, P. Interprocedural Analysis with Lazy Propagation. In: *International Static Analysis Symposium, Lecture Notes in Computer Science, vol 6337. Springer, Berlin, Heidelberg*. 2010.

[44] Madsen, M. and Møller, A. Sparse Dataflow Analysis with Pointers and Reachability. In: *International Static Analysis Symposium, Lecture Notes in Computer Science, vol 8723. Springer, Cham*. 2014.

[45] Sotiropoulos, T. and Livshits, B. Static Analysis for Asynchronous Javascript Programs. In: *Proceedings of the European Conference on Object-Oriented Programming*. 2019.

[46] Hackett, B., Lebresne, S., Burg, B., and Vitek, J. Fast and Precise Hybrid Type Inference for Javascript. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2012.

[47] Chandra, S., Gordon, C. S., Jeannin, J.-B., Schlesinger, C., Sridharan, M., Tip, F., and Choi, Y. Type Inference for Static Compilation of Javascript. In: *ACM SIGPLAN Notices*. 2016.

[48] Jensen, S. H., Møller, A., and Thiemann, P. Type Analysis for Javascript. In: *Proceedings of the 16th International Symposium on Static Analysis*. 2009.

[49] Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., and Tip, F. Correlation Tracking for Points-To Analysis of Javascript. In: *Proceedings of the European Conference on Object-Oriented Programmings*. 2012.

[50] Madsen, M., Livshits, B., and Fanning, M. Practical Static Analysis of Javascript Applications in the Presence of Frameworks and Libraries. In: *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2013.

[51] Backes, M., Rieck, K., Skoruppa, M., Stock, B., and Yamaguchi, F. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In: *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*. 2017.

[52] Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. Riding out domsday: towards detecting and preventing dom cross-site scripting. In: *2018 Network and Distributed System Security Symposium (NDSS)*. 2018.

[53] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A symbolic execution framework for JavaScript. In: *IEEE S&P Symposium*. 2010, 513–528.

[54] Steffens, M., Rossow, C., Johns, M., and Stock, B. Don't Trust the Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In: *NDSS*. 2019.

[55] Nicolay, J., Spruyt, V., and Roover, C. D. Static Detection of User-specified Security Vulnerabilities in Client-side JavaScript. In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS)*. 2016.

[56] Weissbacher, M., Robertson, W., Kirda, E., Kruegel, C., and Vigna, G. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In: *USENIX Security Symposium*. 2015.

[57] Saxena, P., Hanna, S., Poosankam, P., and Song, D. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In: *Proceedings of the Network and Distributed Systems Security Symposium*. 2010.

[58] Skrupsky, N., Monshizadeh, M., Bisht, P., Hinrichs, T., Venkatakrishnan, V., and Zuck, L. WAVES: Automatic Synthesis of Client-side Validation Code for Web Applications. In: *2012 International Conference on Cyber Security*. 2012.

[59] Somé, D. F. EmPoWeb: Empowering Web Applications with Browser Extensions. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2019.

[60] Calzavara, S., Bugliesi, M., Crafa, S., and Steffinlongo, E. Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions. In: *Programming Languages and Systems - 24th European Symposium on Programming (ESOP)*. 2015.

[61] Barth, A., Weinberger, J., and Song, D. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In: *Proceedings of the USENIX Security Symposium*. 2009.

[62] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. Thou shalt not depend on me: analysing the use of outdated javascript libraries on the web. *NDSS 2017* (2017).

[63] *JQuery library*. `https://jquery.com/`.

[64] Jensen, S. H., Madsen, M., and Møller, A. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* 2011, 59–69.

[65] *Usage statistics of JavaScript libraries for websites.* `https://w3techs.com/technologies/overview/javascript_library`. 2020.

[66] Johns, M. and Winter, J. *RequestRodeo: Client-side Protection Against Session Riding.* `https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf`. 2006.

[67] Burns, J. Cross site reference forgery: an introduction to a common web application weakness. In: *Information Security Partners, LLC.* 2005.

[68] *Critical CSRF Vulnerability on Facebook.* `https://www.acunetix.com/blog/web-security-zone/critical-csrf-vulnerability-facebook/`. 2019.

[69] *OWASP Cross-Site Request Forgery Prevention Cheat Sheet.* `https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html`.

[70] Likaj, X., Khodayari, S., and Pellegrino, G. Where we stand (or fall): an analysis of csrf defenses in web frameworks. In: *RAID Symposium.* 2021, 370–385.

[71] Wilander, J. Advanced csrf and stateless anti-csrf (2012).

[72] Ryck, P. D., Desmet, L., Joosen, W., and Piessens, F. Automatic and precise client-side protection against CSRF attacks. In: *ESORICS.* 2011.

[73] *Cross-Origin Opener Policy.* `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy`.

[74] *Cross-Origin Embedder Policy.* `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy`.

[75] Nadji, Y., Saxena, P., and Song, D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In: *NDSS.* 2009.

[76] Grossman, J., Fogie, S., Hansen, R., Rager, A., and Petkov, P. D. *XSS Attacks: Cross-Site Scripting Exploits and Defense.* Syngress, 2007.

[77] Dahse, J. and Holz, T. Static Detection of Second-Order Vulnerabilities in Web Applications. In: *USENIX Security.* 2014.

[78] Heiderich, M., Späth, C., and Schwenk, J. DOMPurify: Client-side Protection Against XSS and Markup Injection. In: *ESORICS.* 2017.

[79] Samuel, M., Saxena, P., and Song, D. Context-sensitive Auto-sanitization in Web Templating Languages Using Type Qualifiers. In: *CCS.* 2011.

[80] Saxena, P., Molnar, D., and Livshits, B. SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications. In: *CCS.* 2011.

[81] Bates, D., Barth, A., and Jackson, C. Regular Expressions Considered Harmful in Client-side XSS Filters. In: *WWW.* 2010, 91–100.

[82] Wurzinger, P., Platzer, C., Ludl, C., Kirda, E., and Kruegel, C. SWAP: Mitigating XSS attacks using a reverse proxy. In: *ICSE Workshop on Software Engineering for Secure Systems.* 2009.

[83] Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. Busting Frame Busting: A Study of Clickjacking Vulnerabilities at Popular Sites. *IEEE S&P* (2010).

[84] *document.cookie DOM property can be clobbered using DOM node named cookie.* https://bugzilla.mozilla.org/show_bug.cgi?id=1420032. 2018.

[85] *Pentest-Report DOMPurify.* https://cure53.de/pentest-report_dompurify.pdf. 2015.

[86] *Provide an opt-out for inputs overriding form DOM API.* https://github.com/whatwg/html/issues/2212.

[87] *Feature Proposal: no [OverrideBuiltins].* https://github.com/WICG/document-policy/issues/16.

[88] Bentkowski, M. XSS in GMail's AMP4Email via DOM Clobbering (2019). https://research.securitum.com/xss-in-amp4email-dom-clobbering/.

[89] *Clobbering the clobbered vol.2.* https://terjanq.medium.com/clobbering-the-clobbered-vol-2-fb199ad7ec41. 2019.

[90] *Disabling DOM clobbering.* https://github.com/w3c/webappsec-permissions-policy/issues/349.

[91] *Feature proposal: Disable named access on window.* https://github.com/WICG/document-policy/issues/32.

[92] West, M. Content Security Policy Level 3. *W3C Working Draft* (2022). https://w3c.github.io/webappsec-csp/#directive-script-src.

[93] Heiderich, M., Niemietz, M., and Schwenk, J. Waiting for CSP – Securing Legacy Web Applications with JSAgents. In: *ESORICS*. 2015, 23–42.

[94] *The Chromium Projects: SameSite Updates.* https://www.chromium.org/updates/same-site.

[95] *Feature: Cookies default to SameSite=Lax.* https://www.chromestatus.com/feature/5088147346030592. 2020.

[96] Calvano, P. SameSite Cookies - Are you Ready? (2020). https://dev.to/httparchive/samesite-cookies-are-you-ready-5abd.

[97] *Impact of the Changes to the SameSite Cookie Flag Default Behavior in Chrome.* https://wiki.resolution.de/doc/saml-sso/latest/all/knowledgebase-articles/technical/impact-of-the-changes-to-the-samesite-cookie-flag-default-behavior-in-chrome.

[98] Skokan, F. Upcoming Browser Behavior Changes: What Developers Need to Know (2020). https://auth0.com/blog/browser-behavior-changes-what-developers-need-to-know/.

[99] Geesink, B. Default cookie SameSite attribute behaviour change (2020). https://wiki.surfnet.nl/display/surfconextdev/Default+cookie+SameSite+attribute+behaviour+change.

[100] *PhenixID: SameSite cookie patch.* https://document.phenixid.net/m/87804/l/1201413-samesite-cookie-patch.

[101] Dixon, J. and Paine, M. Upcoming SameSite cookie changes and the impact for APEX Apps running in an iframe (2020). https://www.jmjcloud.com/blog/upcoming-samesite-cookie-changes-and-the-impact-for-apex-apps-running-in-an-iframe.

[102] Staicu, C. A. and Pradel, M. Leaky Images: Targeted Privacy Attacks in the Web. In: *USENIX Security Symposium*. 2019.

[103] *Bypass SameSite Cookies Default to Lax and get CSRF.* https://medium.com/@renwa/bypass-samesite-cookies-default-to-lax-and-get-csrf-343ba09b9f2b.

[104] *Defending against CSRF with SameSite cookies.* https://portswigger.net/web-security/csrf/samesite-cookies.

[105] Rabal, J. Same-Site cookies against CSRF attacks analysis (2017). https://www.tarlogic.com/en/blog/samesite-cookies-analysis/.

[106] *Project Foxhound.* https://github.com/SAP/project-foxhound.

[107] Klein, D., Barber, T., Bensalim, S., Stock, B., and Johns, M. Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions. In: *IEEE EuroS&P.* 2022.

[108] *Chrome DevTools Protocol.* https://chromedevtools.github.io/devtools-protocol/.

[109] Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In: *ACM CCS.* 2016, 1376–1387.

[110] Maier, F. *Iroh.js: Dynamic Code Analysis for JavaScript.* https://maierfelix.github.io/Iroh/. 2018.

[111] *The CacheStorage Web API.* https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage.

[112] Kotowicz, K. Prevent DOM-based cross-site scripting vulnerabilities with Trusted Types (2020). https://web.dev/trusted-types/.

[113] *DOM-based open redirection.* https://portswigger.net/web-security/dom-based/open-redirection.

[114] Banach, Z. Open redirect vulnerabilities and how to avoid them (2021). https://www.netsparker.com/blog/web-security/open-redirect-vulnerabilities-netsparker-pauls-security-weekly/.

[115] Berners-Lee, T. and Connolly, D. Hypertext Markup Language - 2.0 (RFC 1866). In: *Internet Engineering Task Force.* https://datatracker.ietf.org/doc/html/rfc1866. 1995.

[116] *HTML Living Standard.* https://html.spec.whatwg.org/multipage/.

[117] *The World Wide Web Consortium (W3C).* https://www.w3.org/. (Visited on 08/30/2023).

[118] *The Web Hypertext Application Technology Working Group (WHATWG).* https://whatwg.org. (Visited on 08/30/2023).

[119] Hors, A. L. and Jacobs, I. W3C HTML 4.01 Specification (1999). https://www.w3.org/TR/html401/.

[120] Goodman, D. *Dynamic HTML: The definitive reference: A comprehensive resource for HTML, CSS, DOM & JavaScript.* O'Reilly Media, Inc, 2002.

[121] *Chromium Blog. Saying goodbye to Flash in Chrome.* https://blog.google/products/chrome/saying-goodbye-flash-chrome/. (Visited on 08/30/2023).

[122] *Chromium Blog. So long, and thanks for all the Flash.* https://blog.chromium.org/2017/07/so-long-and-thanks-for-all-flash.html. (Visited on 08/30/2023).

[123] Keith, J. A brief history of javascript. *DOM Scripting: Web Design with JavaScript and the Document Object Model* (2005), 3–10.

[124] *Node.js: An Open-Source, Cross-Platform JavaScript Runtime Environment.* `https://nodejs.org`. (Visited on 08/30/2023).

[125] Chaniotis, I. K., Kyriakou, K.-I. D., and Tselikas, N. D. Is node. js a viable option for building modern web applications? a performance evaluation study. *Computing* (2015), 1023–1044.

[126] ECMAScript language specification, 14th edition (August 2023) (2023). `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`.

[127] Al Kassar, F., Clerici, G., Compagna, L., Balzarotti, D., and Yamaguchi, F. Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications. In: *NDSS Symposium.* 2022.

[128] Yamaguchi, F., Lottmann, M., and Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees. In: *Proceedings of the Annual Computer Security Applications Conference.* 2012.

[129] Gallaba, K., Mesbah, A., and Beschastnikh, I. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In: *Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 2015.

[130] *What is the Document Object Model?* `https://www.w3.org/TR/WD-DOM/introduction.html`. (Visited on 08/30/2023).

[131] *The Document Interface.* `https://developer.mozilla.org/en-US/docs/Web/API/Document`. (Visited on 08/30/2023).

[132] *Web APIs.* `https://developer.mozilla.org/en-US/docs/Web/API`. (Visited on 08/30/2023).

[133] Schwenk, J., Niemietz, M., and Mainka, C. Same-Origin Policy: Evaluation in Modern Browsers. In: *Proceedings of the 26th USENIX Conference on Security Symposium.* 2017.

[134] Same-origin policy (). `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`.

[135] Barth, A. Http state management mechanism (2011). `https://tools.ietf.org/html/rfc6265`.

[136] Sivakorn, S., Polakis, I., and Keromytis, A. D. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In: *IEEE Symposium on Security and Privacy.* 2016.

[137] Lekies, S., Stock, B., Wentzel, M., and Johns, M. The Unexpected Dangers of Dynamic JavaScript. In: *USENIX Security Symposium.* 2015.

[138] Czeskis, A., Moshchuk, A., Kohno, T., and Wang, H. J. Lightweight Server Support for Browser-based CSRF Protection. In: *International Conference on World Wide Web.* 2013.

[139] *Account Take Over in US Dept of Defense.* `https://hackerone.com/reports/410099`. 2019. (Visited on 08/30/2023).

[140] *Critical CSRF Vulnerability on Facebook.* `https://www.acunetix.com/blog/web-security-zone/critical-csrf-vulnerability-facebook/`. (Visited on 08/30/2023).

[141] *WordPress CVE-2014-9033.* `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9033`. (Visited on 08/30/2023).

[142] Cardwell, M. Abusing HTTP Status Codes to Expose Private Information (2011). https://www.grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information.

[143] Yoneuchi, T. Detect the Same-Origin Redirection with a bug in Firefox's CSP Implementation (2018). https://diary.shift-js.info/csp-fingerprinting/.

[144] Linus, R. Your Social Media Fingerprint (). https://github.com/RobinLinus/socialmedia-leak.

[145] Van Goethem, T., Joosen, W., and Nikiforakis, N. The Clock is Still Ticking: Timing Attacks in the Modern Web. In: *ACM SIGSAC Conference on Computer and Communications Security.* 2015.

[146] *XS-Leaks Wiki.* https://xsleaks.com/.

[147] Chen, S., Wang, R., Wang, X., and Zhang, K. Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In: *IEEE Symposium on Security and Privacy.* 2010.

[148] Mao, J., Chen, Y., Shi, F., Jia, Y., and Liang, Z. Toward Exposing Timing-Based Probing Attacks in Web Applications. In: *International Conference on Wireless Algorithms, Systems, and Applications.* 2016.

[149] Weinberg, Z., Chen, E. Y., Jayaraman, P. R., and Jackson, C. I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks. In: *IEEE Symposium on Security and Privacy.* 2011.

[150] Johns, M. The three faces of csrf. talk at the deepsec2007 conference. (2007). https://deepsec.net/archive/2007.deepsec.net/speakers/index.html.

[151] *Two Factor Authentication Cross Site Request Forgery (CSRF) Vulnerability in Wordpress. CVE-2018-20231.* https://www.privacy-wise.com/two-factor-authentication-cross-site-request-forgery-csrf-vulnerability-cve-2018-20231/. 2018.

[152] *CSRF: Adding Optional Two Factor Mobile Number in Slack.* https://hackerone.com/reports/155774. 2016.

[153] *YUI library.* https://yuilibrary.com/.

[154] *window.open() API.* https://developer.mozilla.org/en-US/docs/Web/API/Window/open.

[155] *window.name API.* https://developer.mozilla.org/en-US/docs/Web/API/Window/name.

[156] Zheng, X., Jiang, J., Liang, J., Duan, H., Chen, S., and Wan, T. Cookies Lack Integrity: Real-World Implications. In: *USENIX Security Symposium.* 2015.

[157] Sivakorn, S., Polakis, I., and Keromytis, A. D. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In: *Proceedings of the IEEE European Symposium on Security and Privacy.* 2016.

[158] Jovanovic, N., Kirda, E., and Kruegel, C. Preventing cross site request forgery attacks. In: *Second International Conference on Security and Privacy in Communication Networks and the Workshops (SecureComm).* 2006.

[159] Mao, Z., Li, N., and Molloy, I. Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. In: *13th International Conference on Financial Cryptography and Data Security.* 2009.

[160] Schwenk, J., Niemietz, M., and Mainka, C. Same-Origin Policy: Evaluation in Modern Browsers. In: *USENIX Security Symposium.* 2017.

[161] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. Towards a formal foundation of web security. In: *IEEE CSF*. 2010.

[162] *Cross-Origin Resource Sharing*. `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

[163] *CSP connect-src Directive*. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/connect-src`.

[164] Braun, F., Heiderich, M., and Vogelheim, D. HTML Sanitizer API, Section 4.2, DOM Clobbering. *W3C Draft Community Group Report* (2021). `https://wicg.github.io/sanitizer-api/#dom-clobbering`.

[165] *Undefined primitive type*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined`.

[166] Rascia, T. Understanding Variables, Scope, and Hoisting in JavaScriptt (2021). `https://www.digitalocean.com/community/tutorials/understanding-variables-scope-hoisting-in-javascript`.

[167] *HTML Living Standard: Named Access on the Window Object*. `https://html.spec.whatwg.org/multipage/window-object.html#named-access-on-the-window-object`.

[168] *HTML Living Standard: DOM Tree Accessors*. `https://html.spec.whatwg.org/multipage/dom.html#dom-tree-accessors`.

[169] Etemad, E. J., Jr., T. A., Çelik, T., Glazman, D., Hickson, I., Linss, P., and Williams, J. Selectors Level 4, W3C Working Draft (2018).

[170] *Web IDL Living Standard - Named Property Visibility Algorithm, Sections 3.4.7 and 3.9.7*. `https://webidl.spec.whatwg.org/#legacy-platform-object-abstract-ops`.

[171] Dynamic email in Gmail becoming generally available on July 2019 (2019). `https://workspaceupdates.googleblog.com/2019/06/dynamic-email-in-gmail-becoming-GA.html`.

[172] Peek, J. GitHub Handling of Named HTML Elements Generated by Repository Markdown Code (2014). `https://github.com/gjtorikian/html-pipeline/pull/111`.

[173] Puzrin, V. DOM Clobbering through Markdown Header anchors (2015). `https://github.com/markdown-it/markdown-it/issues/28`.

[174] West, M. Same-site Cookies (2016).

[175] *SameSite Cookies*. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite`.

[176] *Safe HTTP Methods*. `https://developer.mozilla.org/en-US/docs/Glossary/safe`.

[177] *Site compatibility-impacting changes coming to Microsoft Edge*. `https://docs.microsoft.com/en-us/microsoft-edge/web-platform/site-impacting-changes`.

[178] *Can I use SameSite cookie attribute?* `https://caniuse.com/?search=samesite`.

[179] *Bitnami application catalog*. `https://bitnami.com/stacks`.

[180] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. In: *ACM Transactions on Programming Languages and Systems*. 1987.

[181] Mozilla. *Introduction to the DOM.* https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. 2020.

[182] *Selenium-Python.* https://selenium-python.readthedocs.io/index.html.

[183] *Selenium browser automation.* https://www.selenium.dev.

[184] *Selenium IDE.* https://www.selenium.dev/projects.

[185] *Headless chromium.* https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md.

[186] *Neo4j Graph Database.* https://neo4j.com/.

[187] *Library Detector For chrome.* https://www.npmjs.com/package/js-library-detector.

[188] *Esprima.* https://esprima.org/.

[189] *Escontrol library.* https://www.npmjs.com/package/escontrol.

[190] *Styx library.* https://www.npmjs.com/package/styx.

[191] *ast-flow-graph library.* https://www.npmjs.com/package/ast-flow-graph.

[192] *Esgraph CFG generator.* https://github.com/Swatinem/esgraph.

[193] *Dujs library.* https://github.com/chengfulin/dujs.

[194] Lam., M. S., Avaya, R. S., and Ullman, J. D. Compilers: principles, techniques, and tools (2nd edition). In: *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.* 2006.

[195] *Function.prototype.call().* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call.

[196] *Function.prototype.apply().* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply.

[197] *Cypher Query Language.* https://neo4j.com/developer/cypher/.

[198] *Usage statistics of content management systems.* https://w3techs.com/technologies/overview/content_management.

[199] HTML Living Standard (2023).

[200] XMLHttpRequest Living Standard (). https://xhr.spec.whatwg.org/.

[201] WebSockets Living Standard (2023).

[202] WHATWG Specifications (). https://spec.whatwg.org/.

[203] W3C Standards and Drafts (). https://www.w3.org/TR/.

[204] Khodayari, S. and Pellegrino, G. It's (dom) clobbering time: attack techniques, prevalence, and defenses. In: *IEEE S&P Symposium.* 2023.

[205] Push API: CSRF on PushManager Subscriptions ().

[206] Subramani, K., Jueckstock, J., Kapravelos, A., and Perdisci, R. Sok: workerounds-categorizing service worker attacks and mitigations. In: *IEEE EuroS&P Symposium.* 2022.

[207] Watanabe, T., Shioji, E., Akiyama, M., and Mori, T. Melting pot of origins: compromising the intermediary web services that rehost websites. In: *NDSS Symposium.* 2020.

[208] Hickson, I. Server-sent Events. In: *W3C Working Draft.* 2012.

[209] Schneider, C. Cross-Site WebSocket Hijacking (CSWSH). In: 2019.

[210] Cross-Site WebSocket Hijacking. In:

[211] Mei, W. and Long, Z. Research and Defense of Cross-Site WebSocket Hijacking Vulnerability. In: *IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. 2020.

[212] Murley, P., Ma, Z., Mason, J., Bailey, M., and Kharraz, A. WebSocket Adoption and the Landscape of the Real-Time Web. In: *WWW Web Conference*. 2021.

[213] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross site scripting prevention with dynamic data tainting and static analysis. In: *NDSS Symposium*. 2007.

[214] Exposure of Sensitive Information to Unauthorized Actors in EventSource (2022).

[215] *The WebSocket API*. URL: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`.

[216] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. Tranco: a research-oriented top sites ranking hardened against manipulation. In: *NDSS Symposium*. 2019.

[217] *Playwright browser automation framework*. `https://playwright.dev/`.

[218] *Firefox developer tools*. `https://firefox-dev.tools/`.

[219] Pletinckx, S., Borgolte, K., and Fiebig, T. Out of Sight, Out of Mind: Detecting Orphaned Web Pages at Internet-Scale. In: *ACM CCS*. 2021.

[220] Henzinger, M. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: *ACM SIGIR conference on Research and development in information retrieval*. 2006.

[221] *setTimeout global function*. `https://developer.mozilla.org/en-US/docs/Web/API/setTimeout`.

[222] *Ineo: neo4j instance and version manager*. `https://github.com/cohesivestack/ineo`.

[223] *JavaScript Function() constructor*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/Function`.

[224] Lekies, S., Tighzert, W., and Johns, M. Towards Stateless, Client-side Driven Cross-site Request Forgery Protection for Web Applications. *SAP Research* (2012).

[225] Shahriar, H. and Zulkernine, M. Client-side detection of cross-site request forgery attacks. In: *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*. 2010.

[226] Shernan, E., Carter, H., Tian, D., Traynor, P., and Butler, K. More guidelines than rules: csrf vulnerabilities from noncompliant oauth 2.0 implementations. In: *DIMVA*. 2015.

[227] Jabiyev, B., Mirzaei, O., Kharraz, A., and Kirda, E. Preventing server-side request forgery attacks. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, 1626–1635.

[228] Sudhodanan, A., Khodayari, S., and Caballero, J. Cross-origin state inference (COSI) attacks: Leaking web site states through xs-leaks. In: *NDSS Symposium*. 2020.

[229] *W3C Standards and Drafts*. `https://www.w3.org/TR/`.

[230] Alkhalaf, M., Bultan, T., and Gallegos, J. L. Verifying client-side input validation functions using string analysis. In: *ICSE*. 2012.

[231] Kerschbaum, F. Simple cross-site attack prevention. In: *International Conference on Security and Privacy in Communications Networks and the Workshops (SecureComm)*. IEEE. 2007.

[232] Balduzzi, M., Gimenez, C. T., Balzarotti, D., and Kirda, E. Automated discovery of parameter pollution vulnerabilities in web applications. In: *NDSS Symposium*. 2011.

[233] *Fetch MetaData Sec-Fetch-Dest Header*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Dest.

[234] *Bypassing sanitization using DOM clobbering in HTML-Janitor*. https://hackerone.com/reports/308158. 2018.

[235] *DOM Clobbering affecting Google Analytics script*. https://twitter.com/zachleat/status/1387460811522813953.

[236] Janc, A. and West, M. Oh, the Places You'll Go! Finding Our Way Back from the Web Platform's Ill-conceived Jaunts. In: *IEEE EuroS&P Workshops*. 2020, 673–680.

[237] *Chrome Platform Status: DOM Clobbered Variable Accessed*. https://chromestatus.com/metrics/feature/timeline/popularity/1824.

[238] Roth, S., Barron, T., Calzavara, S., Nikiforakis, N., and Stock, B. Complex security policy? a longitudinal analysis of deployed content security policies. In: *NDSS*. 2020.

[239] Stamm, S., Sterne, B., and Markham, G. Reining in the Web with Content Security Policy. In: *WWW*. 2010, 921–930.

[240] *DOM Clobbering Vulnerability Reports in HackerOne*. https://hackerone.com/hacktivity?querystring=dom%20clobbering.

[241] *DOM Clobbering Vulnerability Reports in Mitre*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=dom+clobbering.

[242] *Clobbering the clobbered — Advanced DOM Clobbering*. https://terjanq.medium.com/dom-clobbering-techniques-8443547ebe94. 2019.

[243] Nafeez, A. DomFlow - Untangling the DOM for easy juicy bugs (2015). https://www.blackhat.com/docs/us-15/materials/us-15-Nafeez-Dom-Flow-Untangling-The-DOM-For-More-Easy-Juicy-Bugs.pdf.

[244] *DOM Living Standard*. https://dom.spec.whatwg.org/.

[245] *WHATWG DOM repository issues*. https://github.com/whatwg/dom/issues.

[246] *BrowserStack*. https://www.browserstack.com/.

[247] H., S. How to Update Safari without upgrading MacOS? (2021). https://browserhow.com/how-to-update-safari-without-upgrading-macos/.

[248] *The Window Interface*. https://developer.mozilla.org/en-US/docs/Web/API/Window.

[249] *The HTMLCollection Interface*. https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection. 2021.

[250] *The Blink Rendering Engine*. https://www.chromium.org/blink/.

[251] Charlton, H. Should Apple Continue to Ban Rival Browser Engines on iOS? (2022). https://www.macrumors.com/2022/02/25/should-apple-ban-rival-browser-engines/.

[252] *The Notification Web API*. https://developer.mozilla.org/en-US/docs/Web/API/notification.

[253] *The WebStorage API*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.

[254] *Puppeteer*. https://github.com/puppeteer/puppeteer.

[255] *Rest Parameters.* `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters`.

[256] *Spread Operator Syntax.* `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax`.

[257] Guarnieri, S. and Livshits, V. B. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In: *USENIX Security.* 2009.

[258] *The arguments object.* `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments`.

[259] *DOM-based WebSocket-URL poisoning.* `https://portswigger.net/web-security/dom-based/websocket-url-poisoning`.

[260] Polop, C. Cross-site WebSocket hijacking (2022). `https://book.hacktricks.xyz/pentesting-web/cross-site-websocket-hijacking-cswsh`.

[261] *DOM-based document-domain manipulation.* `https://portswigger.net/web-security/dom-based/document-domain-manipulation`.

[262] Nideck, T. A. What Are JSON Injections? (2019). `https://www.acunetix.com/blog/web-security-zone/what-are-json-injections`.

[263] *Client-side JSON injection.* `https://portswigger.net/kb/issues/00200370_client-side-json-injection-dom-based`.

[264] Staicu, C.-A. and Pradel, M. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In: *USENIX Security.* 2018, 361–376.

[265] Davis, J. C., Coghlan, C. A., Servant, F., and Lee, D. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In: *ESEC/FSE.* 2018, 246–256.

[266] *DOM-based Local File-path Manipulation.* `https://portswigger.net/web-security/dom-based/local-file-path-manipulation`.

[267] *Boomerang Library.* `https://developer.akamai.com/tools/boomerang`.

[268] *Google Closure Library HTML Sanitizer.* `https://github.com/google/closure-library/blob/master/closure/goog/html/sanitizer/htmlsanitizer.js`.

[269] *JS-XSS HTML Sanitizer.* `https://github.com/leizongmin/js-xss`.

[270] *Sanitize-HTML Library.* `https://github.com/apostrophecms/sanitize-html`.

[271] *Google Caja Sanitizer.* `https://code.google.com/archive/p/google-caja/wikis/JsHtmlSanitizer.wiki`.

[272] *Insane HTML Sanitizer.* `https://github.com/bevacqua/insane`.

[273] *JavaScript Bleach Sanitizer.* `https://www.npmjs.com/package/bleach`.

[274] *Angular-sanitize Library.* `https://www.npmjs.com/package/bleach`.

[275] *HTML-Purify Library.* `https://www.npmjs.com/package/html-purify`.

[276] *Arcgis HTML Sanitizer.* `https://www.npmjs.com/package/@esri/arcgis-html-sanitizer`.

[277] *Python Mozilla Bleach Sanitizer.* `https://pypi.org/project/bleach/`.

[278] *LXML Library.* `https://pypi.org/project/lxml/`.

[279] *Python HTML-sanitizer Library.* `https://pypi.org/project/html-sanitizer/`.

[280] *HTMLLaundry Library.* https://pypi.org/project/htmllaundry/.

[281] *Django HTML Sanitizer.* https://pypi.org/project/django-html_sanitizer/.

[282] *PHP HTML Purifier.* https://packagist.org/packages/ezyang/htmlpurifier.

[283] *PHP HTML-Sanitizer.* https://packagist.org/packages/tgalopin/html-sanitizer.

[284] *Symfony HTML Sanitizer.* https://packagist.org/packages/symfony/html-sanitizer.

[285] *HTMLawed Library.* https://packagist.org/packages/htmlawed/htmlawed.

[286] *Typo3 HTML Sanitizer.* https://packagist.org/packages/typo3/html-sanitizer.

[287] *HTML Encoder of AntiXSS Library.* https://docs.microsoft.com/en-us/dotnet/api/system.web.security.antixss.antixssencoder.htmlencode?view=netframework-4.8.

[288] *C# HtmlSanitizer.* https://www.nuget.org/packages/HtmlSanitizer.

[289] *ASP.NET Ajax Control Toolkit.* https://www.nuget.org/packages/AjaxControlToolkit.HtmlEditor.Sanitizer/.

[290] *NSoup HTML Parser and Sanitizer for .NET Framework.* https://www.nuget.org/packages/NSoup/.

[291] *HTMLRuleSanitier Library.* https://www.nuget.org/packages/Vereyon.Web.HtmlSanitizer.

[292] *JSoup: Java HTML Parser.* https://github.com/jhy/jsoup.

[293] *OWASP Java HTML Sanitizer.* https://github.com/OWASP/java-html-sanitizer.

[294] *Java AntiSamy Library.* https://github.com/nahsra/antisamy.

[295] *HtmlCleaner Library.* http://htmlcleaner.sourceforge.net/index.php.

[296] *GitHub Octoverse report.* https://octoverse.github.com/.

[297] *Object Freeze API.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze.

[298] *The instanceof Operator.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof.

[299] *The typeof Operator.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof.

[300] *The globalThis object.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis.

[301] *The MutationObserver API.* https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver.

[302] Helme, S. CSRF is (really) dead (). https://scotthelme.co.uk/csrf-is-really-dead/.

[303] Rees-Carter, S. CSRF is dead, long live SameSite=Lax (or is it?) (). https://stephenreescarter.net/csrf-is-dead-long-live-samesite-lax/.

[304] *Using the Same-Site Cookie Attribute to Prevent CSRF Attacks.* https://www.netsparker.com/blog/web-security/same-site-cookie-attribute-prevent-cross-site-request-forgery/.

[305] Sharma, R. Preventing Cross-Site Attacks Using SameSite Cookies (). https://dropbox.tech/security/preventing-cross-site-attacks-using-same-site-cookies.

[306] Riramar, P. K. *OWASP: SameSite Attribute.* https://owasp.org/www-community/SameSite.

[307] Calzavara, S., Urban, T., Tatang, D., Steffens, M., and Stock, B. Reining in the Web's Inconsistencies with Site Policy. In: *Network and Distributed Systems Security Symposium.* 2021.

[308] Mendoza, A., Chinprutthiwong, P., and Gu, G. Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites. In: *World Wide Web Conference.* 2018.

[309] *Django HttpResponse.set_cookie() API.* https://docs.djangoproject.com/en/3.1/ref/request-response/.

[310] *Pyramid Response.set_cookie() API.* https://docs.pylonsproject.org/projects/pyramid/en/latest/api/response.html.

[311] *Chrome DevTools Protocol Audits.* https://chromedevtools.github.io/devtools-protocol/tot/Audits/.

[312] *Web Shrinker API.* https://www.webshrinker.com/.

[313] *EasyList.* https://easylist.to/.

[314] *Host BlackList.* https://github.com/anudeepND/blacklist.

[315] *Host BlockList.* https://github.com/notracking/hosts-blocklists.

[316] Franken, G., Van Goethem, T., and Joosen, W. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-party Cookie Policies. In: *27th USENIX Security Symposium.* 2018.

[317] Franken, G., Van Goethem, T., and Joosen, W. Exposing Cookie Policy Flaws Through an Extensive Evaluation of Browsers and Their Extensions. In: *IEEE Symposium on Security and Privacy.* 2019.

[318] Drakonakis, K., Ioannidis, S., and Polakis, J. The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws. In: *ACM SIGSAC Conference on Computer and Communications Security.* 2020.

[319] Van Goethem, T., Le Pochat, V., and Joosen, W. Mobile Friendly or Attacker Friendly? A Large-Scale Security Evaluation of Mobile-First Websites. In: *ACM Asia Conference on Computer and Communications Security.* 2019.

[320] *StackExchange Security Community.* https://security.stackexchange.com/.

[321] *Dev Security Community.* https://dev.to/t/security.

[322] *Issue 831725: SameSite cookie bypass via prerender.* https://bugs.chromium.org/p/chromium/issues/detail?id=831725.

[323] *Cookies with SameSite=None or SameSite=invalid treated as Strict.* https://bugs.webkit.org/show_bug.cgi?id=198181.

[324] *Mozilla CVE-2018-12370.* https://www.cvedetails.com/cve/CVE-2018-12370/.

[325] *CVE-2018-18351.* https://nvd.nist.gov/vuln/detail/CVE-2018-18351.

[326] *CVE-2019-5880: SameSite cookie bypass.* https://bugzilla.redhat.com/show_bug.cgi?id=1762378.

[327] *SameSite cookies aren't sent on credentialed CORS requests.* https://github.com/whatwg/fetch/issues/769.

[328] Sabol, C. It's Okay, We're All On the SameSite (2020). https://securityboulevard.com/2020/02/its-okay-were-all-on-the-samesite/.

[329] Rees-Carter, S. *SameSite Cookies Deep Dive / CSRF is dead (or is it?)* https://stephenreescarter.net/talks/samesite-cookies/.

[330] Li, V. Bypassing CSRF Protection (2020). https://vickieli.dev/csrf/bypass-csrf-protection/.

[331] *SameSite cookies.* https://makandracards.com/makandra/71018-samesite-cookies.

[332] Walton, J. Avoiding CSRF Attacks with API Design (2020). http://www.thedreaming.org/2020/05/26/avoid-csrf-attacks-with-api-design/.

[333] *SameSite Cookies and CSRF Attacks.* https://symfonycasts.com/screencast/api-platform-security/samesite-csrf.

[334] *XS-Leaks Wiki: SameSite Cookies.* https://xsleaks.dev/docs/defenses/opt-in/same-site-cookies/.

[335] *SameSite Cookie Attribute and Synchronizer Token Pattern.* https://security.stackexchange.com/questions/201396/samesite-cookie-attribute-and-synchronizer-token-pattern.

[336] *How is the lack of the "SameSite" cookie flag a risk?* https://security.stackexchange.com/questions/154106/how-is-the-lack-of-the-samesite-cookie-flag-a-risk.

[337] *Will same-site cookies be sufficent protection against CSRF and XSS?* https://security.stackexchange.com/questions/121971/will-same-site-cookies-be-sufficent-protection-against-csrf-and-xss.

[338] Jubeau, D. Secure your cookies to the next level with SameSite attribute (2017). https://dev.to/damienjubeau/secure-your-cookies-to-the-next-level-with-samesite-attribute.

[339] Valverde, F. Everybody hates CSRF (2020). https://dev.to/fdoxyz/everybody-hates-csrf-4fek.

[340] *Feature: Reject insecure SameSite=None cookies.* https://www.chromestatus.com/feature/5633521622188032.

[341] Guan, C., Sun, K., Wang, Z., and Zhu, W. Privacy Breach by Exploiting postMessage in HTML5: Identification, Evaluation, and Countermeasure. In: *ACM Asia Conference on Computer and Communications Security.* 2016.

[342] Ballarano, A., Colace, F., De Santo, M., and Greco, L. The Postman Always Rings Twice": Evaluating E-Learning Platform a Decade Later. *International Journal of Emerging Technologies in Learning* (2016).

[343] *SameSite Frequently Asked Questions (FAQ).* https://www.chromium.org/updates/same-site/faq.

[344] *SameSite Cookie Attribute explained.* https://cookie-script.com/documentation/samesite-cookie-attribute-explained.

[345] Merewood, R. SameSite cookie recipes (). https://web.dev/samesite-cookie-recipes/.

[346] Likaj, X., Khodayari, S., and Pellegrino, G. Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks. In: *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 2021.

[347] Parsovs, A. Practical Issues with TLS Client Certificate Authentication. In: *Network and Distributed Systems Security Symposium*. 2014.

[348] Caretton, L. Node.js Connect CSRF Bypass Abusing Method Override Middleware (). `http://blog.nibblesec.org/2014/05/nodejs-connect-csrf-bypass-abusing.html`.

[349] *Often Misused: HTTP Method Override.* `https://vulncat.fortify.com/en/detail?id=desc.dynamic.xtended_preview.often_misused_http_method_override`.

[350] *Mitch Dataset.* `https://github.com/alviser/mitch/tree/master/dataset`.

[351] *Update or reinstall Safari for your computer.* `https://support.apple.com/en-us/HT204416`.

[352] *Flask Response.set_cookie() API.* `https://tedboy.github.io/flask/generated/generated/flask.Response.set_cookie.html`.

[353] *Tornado RequestHandler.set_cookie() API.* `https://www.tornadoweb.org/en/stable/web.html?highlight=set_cookie#tornado.web.RequestHandler.set_cookie`.

[354] *Web.py setcookie() API.* `https://webpy.org/cookbook/cookies`.

[355] *Express SameSite Cookie Attribute.* `https://expressjs.com/en/resources/middleware/session.html`.

[356] *Meteor Cookie.set() API.* `https://docs.meteor.com/`.

[357] *Sails Response.cookie() API.* `https://sailsjs.com/documentation/reference/response-res/res-cookie`.

[358] *Sails SameSite Cookies.* `https://github.com/balderdashy/sails/issues/6942`.

[359] *Koa SameSite Attribute.* `https://github.com/koajs/session/issues/174`.

[360] *Hapi Server.state() API and isSameSite Option.* `https://hapi.dev/api/?v=20.1.0`.

[361] *Laravel SameSite Cookie Attribute.* `https://laracasts.com/discuss/channels/laravel/some-cookies-are-misusing-the-recommended-samesite-attribute`.

[362] *Symfony Cookie API.* `https://symfony.com/doc/current/components/http_foundation.html#setting-cookies`.

[363] *Symfony Default SameSite Cookie Attribute.* `https://github.com/symfony/symfony/blob/c377a795f579e5417d106c94ae5d5fe4b4300dca/src/Symfony/Component/HttpFoundation/Cookie.php`.

[364] *CakePHP withCookie() API.* `https://book.cakephp.org/4/en/controllers/request-response.html#setting-cookies`.

[365] *CakePHP Default SameSite Cookie Attribute.* `https://github.com/cakephp/cakephp/blob/d4b68a6dd2404d0b8cc7431838a39ec44b3f5f6b/src/Http/Cookie/Cookie.php`.

[366] *Zend Default SameSite Cookie Attribute.* `https://github.com/zendframework/zend-http/commit/0d99103d391f4f746e267a00507d753660550f7b`.

[367] *Slim setCookie() API.* `https://www.slimframework.com/docs/v2/response/cookies.html`.

[368] *Slim SameSite Cookie Attribute.* `https://github.com/bryanjhv/slim-session/issues/54`.

[369] *ASP.NET WebForms HttpCookie.* `https://docs.microsoft.com/en-us/aspnet/samesite/csharpwebforms`.

[370] *ASP.NET HttpCookie.* `https://docs.microsoft.com/en-us/aspnet/samesite/csmvc`.

[371] *ASP.NET Core CookieBuilder.* `https://docs.microsoft.com/en-us/aspnet/core/security/samesite?view=aspnetcore-5.0`.

[372] *Nancy Web Framework.* `https://github.com/NancyFx/Nancy`.

[373] *Service Stack UseSameSiteCookies Configuration.* `https://docs.servicestack.net/sessions`.

[374] *Spring SameSite Cookie Attribute.* `https://docs.spring.io/spring-session/docs/current/reference/html5/guides/java-custom-cookie.html`.

[375] *Play Cookie.builder() API.* `https://www.playframework.com/documentation/2.8.x/Migration26#SameSite-attribute,-enabled-for-session-and-flash`.

[376] *Vaadin Cookie.SetCookie() API.* `https://vaadin.com/docs/v8/framework/articles/SettingAndReadingCookies`.

[377] *Vert.X-Web setCookieSameSite API.* `https://github.com/vert-x3/vertx-web/blob/f7902ccd4f5da70908a68611119d77ef4aa3f8d4/vertx-web/src/main/java/io/vertx/ext/web/handler/SessionHandler.java`.

[378] *Spark Response.cookie() API.* `https://sparkjava.com/documentation#getting-started`.

[379] *GitHub 2020 Octoverse Report.* `https://octoverse.github.com/`.

[380] *Stackoverflow Tags.* `https://stackoverflow.com/help/tagging`.

[381] Alhuzali, A., Gjomemo, R., Eshete, B., and Venkatakrishnan, V. NAVEX: precise and scalable exploit generation for dynamic web applications. In: *USENIX Security Symposium.* 2018.

[382] Nielsen, B. B., Hassanshahi, B., and Gauthier, F. Nodest: feedback-driven static analysis of node. js applications. In: *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2019.

[383] Landman, D., Serebrenik, A., and Vinju, J. J. Challenges for static analysis of java reflection-literature review and empirical study. In: *IEEE/ACM 39th International Conference on Software Engineering (ICSE).* 2017.

[384] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. Static analysis of android apps: a systematic literature review. *Information and Software Technology* (2017).

[385] Seo, S.-H., Gupta, A., Sallam, A. M., Bertino, E., and Yim, K. Detecting mobile malware threats to homeland security through static analysis. *Journal of Network and Computer Applications* (2014).

[386] Floyd, R. W. Assigning meanings to programs. In: *Proceedings of Symposium on Applied Mathematics.* 1967.

[387] Knuth, D. E. *The art of computer programming.* The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, 1968.

[388] Wulf, W. A., London, R. L., and Shaw, M. *The Flow Analysis of Computer Programs.* Prentice-Hall, 1976.

[389] Johnson", " C. *"Lint, a C Program Checker".* 1979.

[390] Kildall, G. A. A unified approach to global program optimization. In: *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* ACM. 1973.

[391] King, J. C. Symbolic execution and program testing. In: *Communications of the ACM.* 1976.

[392] Clarke, E. M., Emerson, E. A., and Sifakis, J. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic.* 1986.

[393] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. In: *ACM Transactions on Programming Languages and Systems.* 1990.

[394] Wilson, R. P. and Lam, M. S. Efficient context-sensitive pointer analysis for c programs. In: *ACM Sigplan Notices.* 1995.

[395] Reps, T. Program analysis via graph reachability. In: *Information and Software Technology, 40(11):701–726.* 1998.

[396] Kinloch, D. A. and Munro, M. Understanding c programs using the combined c graph representation. In: *Proceedings of the International Conference on Software Maintenance.* 1994.

[397] Eriksson, B., Pellegrino, G., and Sabelfeld, A. Black widow: blackbox data-driven web scanning. In: *IEEE Symposium on Security and Privacy.* 2021.

[398] Duchene, F., Rawat, S., Richier, J.-L., and Groz, R. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In: *Proceedings of the 4th ACM conference on Data and application security and privacy.* 2014.

[399] Trickel, E., Pagani, F., Zhu, C., Dresel, L., Vigna, G., Kruegel, C., Wang, R., Bao, T., Shoshitaishvili, Y., and Doupé, A. Toss a fault to your witcher: applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In: *IEEE symposium on security and privacy.* 2023.

[400] Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 2013.

[401] Hu, X., Cheng, Y., Duan, Y., Henderson, A., and Yin, H. Jsforce: a forced execution engine for malicious javascript detection. In: *13th International Conference on Security and Privacy in Communication Networks (SecureComm).* 2018.

[402] Newsome, J. and Song, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: *NDSS.* 2005.

[403] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: *IEEE symposium on Security and privacy.* IEEE. 2010.

[404] Al Kassar, F., Clerici, G., Compagna, L., Yamaguchi, F., and Balzarotti, D. Testability tarpits: the impact of code patterns on the security testing of web applications (2022).

[405] Brito, T., Lopes, P., Santos, N., and Santos, J. F. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security* (2022).

[406] Pellegrino, G., Catakoglu, O., Balzarotti, D., and Rossow, C. Uses and abuses of server-side requests. In: *RAID Symposium.* 2016.

[407] Ryck, P. D., Desmet, L., Heyman, T., Piessens, F., and Joosen, W. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In: *International Symposium on Engineering Secure Software and Systems*. 2010.

[408] Kerschbaum, F. Simple cross-site attack prevention. In: *Third International Conference on Security and Privacy in Communications Networks and the Workshops (SecureComm)*. 2007.

[409] Hantke, F. and Stock, B. HTML violations and where to find them: a longitudinal analysis of specification violations in HTML. In: *ACM Internet Measurement Conference*. 2022.

[410] *Dangling markup injection*. `https://portswigger.net/web-security/cross-site-scripting/dangling-markup`.

[411] Roth, S., Backes, M., and Stock, B. Assessing the impact of script gadgets on csp at scale. In: *ACM Asia CCS*. 2020, 420–431.

[412] Heiderich, M. ToStaticHTML for Everyone! About DOMPurify, Security in the DOM, and Why We Really Need Both (2016).

[413] Calzavara, S., Rabitti, A., and Bugliesi, M. CCSP: controlled relaxation of content security policies by runtime policy composition. In: *USENIX Security Symposium*. 2017.

[414] Roth, S., Gröber, L., Backes, M., Krombholz, K., and Stock, B. 12 angry developers-a qualitative study on developers' struggles with csp. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021.

[415] Singh, K., Moshchuk, A., Wang, H. J., and Lee, W. On the Incoherencies in Web Browser Access Control Policies. In: *IEEE Symposium on Security and Privacy*. 2010.

[416] Aggarwal, G., Bursztein, E., Jackson, C., and Boneh, D. An Analysis of Private Browsing Modes in Modern Browsers. In: *USENIX security symposium*. 2010.

[417] Bortz, A., Barth, A., and Czeskis, A. Origin Cookies: Session Integrity for Web Applications. In: *ACM Transactions on Internet Technology (TOIT)*. 2012.

[418] Nikiforakis, N., Meert, W., Younan, Y., Johns, M., and Joosen, W. SessionShield: Lightweight protection against session hijacking. In: *International Symposium on Engineering Secure Software and Systems*. 2011.

[419] Bugliesi, M., Calzavara, S., Focardi, R., and Khan, W. CookiExt: Patching the browser against session hijacking attacks. In: *Journal of Computer Security*. 2015.

[420] Calzavara, S., Rabitti, A., and Bugliesi, M. Sub-session hijacking on the web: Root causes and prevention (2019).

[421] Roesner, F., Kohno, T., and Wetherall, D. Detecting and defending against third-party tracking on the web. In: *9th USENIX Symposium on Networked Systems Design and Implementation*. 2012.

[422] Dhawan, M., Kreibich, C., and Weaver, N. Priv3: A third party cookie policy. In: *W3C Workshop: Do Not Track and Beyond*. 2012.

[423] Zhou, Y. and Evans, D. Why aren't HTTP-only cookies more widely deployed. In: *Proceedings of 4th Web 2.0 Security and Privacy Workshop*. 2010.

[424] Calvano, P. SameSite Cookies Analysis (2020). `https://discuss.httparchive.org/t/samesite-cookies-analysis/1988`.

[425] *SameSite Strict Usage Statistics*. `https://trends.builtwith.com/docinfo/SameSite-Strict`.

[426] *Hackerone*. `https://hackerone.com`.

[427] *Bugcrowd*. `https://www.bugcrowd.com`.

[428] Stock, B., Pellegrino, G., Rossow, C., Johns, M., and Backes, M. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In: *USENIX Security Symposium*. 2016.

[429] *Extra DOM Clobbering protection*. `https://github.com/cure53/DOMPurify/pull/710`.

[430] Jueckstock, J. and Kapravelos, A. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In: *Proceedings of the ACM Internet Measurement Conference (IMC)*. 2019.

[431] Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In: *Proceedings of the 21st USENIX Security Symposium*. 2012.

[432] Pellegrino, G., Tschürtz, C., Bodden, E., and Rossow, C. Jäk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In: *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses*. 2015.

[433] Squarcina, M., Adão, P., Veronese, L., and Maffei, M. Cookie crumbles: breaking and fixing web session integrity. In: *USENIX Security Symposium*. 2023.

[434] Rack, J. and Staicu, C.-A. Jack-in-the-box: an empirical study of javascript bundling on the web and its security implications. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2023.

A
Appendix

A.1 Testbed of Bitnami Applications

This appendix contains the complete list of the Bitnami web applications (alphabetically ordered) and the specific versions that we used for our experiments in §3.3.

Bitnami Applications. AbanteCart 1.2.16, Akeneo 3.2.26, Alfresco Community 201911, Apache Airflow UI 1.10.8, Axelor 5.3.0, Bonita 7.6, CMS Made Simple 2.2.14, CanvasLMS 2020.01.01.05, CiviCRM 5.25.0, Ckan 2.8.0, Collabtive 3.1, Composr 10.0.30, Concrete5 8.5.2, Coppermine 1.6.08, Cotonti 0.9.19, Diaspora 0.7.13.0, Discourse 2.4.5, DokuWiki 20180422c, Dolibarr 11.0.4, DreamFactory 4.2.2, Drupal 8.8.6, ELK 7.6.0, ERPNext 12.9.3, EspoCRM 5.9.1, FatFreeCRM 0.18.1, Fluentd UI 1.10.3, Ghost 3.17.1, Gitlab CE 13.0.3, Grafana 6.5.2, Horde Groupware Webmail 5.2.22, JFrog Artifactory Open Source 6.19.1, JasperReports 7.5.0, Jenkins 2.204.1, JetBrains YouTrack 2019.3.62973, Joomla 3.9.18, Kibana 7.5.1, Kong Admin UI 0.4.1, Kubeapps 1.9.0, Let's Chat 0.4.8, Liferay 7.2.1, LimeSurvery 4.2.5, Live Helper Chat 3.27, LotusCMS 3.0.5, Magento 2.3.5, Mahara 19.10.1, Mantis 2.24.1, Matomo 3.13.1, Mattermost 5.14.0, Mautic 2.16.2, MediaWiki 1.34.1, Moalyss 7.3.0.0, Modx 2.7.3pl, Moodle 3.8.3, MyBB Forum 1.8.22, Neos 5.2.0, OXID eShop 6.2.1, Odoo 13.0.20200515, Open Atrium 2.646, Open edX ironwood.2.8, OpenCart 3.0.3.2, OpenProject 10.5.1, Openfire 4.4.4.1, OrangeHRM 4.4, OroCRM 4.1.4, Osclass 3.9.0, Parse Server 4.2.0, ParseDashboard 2.0.5, Phabricator 2020.21, Pimcore 6.6.4, Plone 5.2.1, Pootle 2.8.2, PrestaShop 1.7.6.2, ProcessMaker Community 3.3.6, ProcessWire 3.0.148, Prometheus 2.18.1, Publify 9.1.0, Re:dash 8.0.0, Redmine 4.1.1, Report Server Community 3.1.1.6020, Report Server Enterprise 3.1.1.6020, ResourceSpace 9.2.14719, ReviewBoard 3.0.17, Roundcube 1.4.5, SEO Panel 4.3.0, Shopware 6.1.0, Silverstripe 4.5.2, Simple Machines Forum 2.0.17, SonarQube 8.2.0.32929, Spree 4.1.6, SugarCRM 6.5.13, SuiteCRM 7.1.1, TestLink 1.9.20, Tiki Wiki CMS Groupware 21, Tiny Tiny RSS 202006, Trac 1.5.1, Typo3 10.4.3, Weblate 4.0.3, Webmail Prop PHP 8.3.20, Wordpress 5.4.1, Xoops 2.5.10, Zurmo 3.2.7, eXo Platform 5.3.0, ownCloud 10.4.1, phpBB 3.3.0, phpList 3.5.4, and phpMyAdmin 5.0.1.

A.2 Additional Evaluation Details

This appendix contains the additional evaluation details for Chapters 4 to 6.

Table A.1: Summary of primitive JavaScript sinks supported by JAW-v2. Rows marked with ⊕ show APIs for which we implemented extra instrumentation in Foxhound$^+$.

Category		JavaScript Sink
Request Hijacking [P1, 5, 6, 154, 204, 208, 211]	⊕	navigator.sendBeacon(T_1, T_2)
		fetch(T_1, T_2)
		XMLHttpRequest.open(T)
		xhr.send(T)
		xhr.setRequestHeader(T_1, T_2)
	⊕	new WebSocket(T)
	⊕	socket.send(T)
	⊕	new EventSource(T)
	⊕	PushManager.subscribe(T)
		window.open(T)
		location.href = T
		location.replace(T)
		location.assign(T)
Code Execution [2, 32, 53, 107]		eval(T)
		new Function(T)
		setInterval(T)
		setTimeout(T)
		script.text = T
		script.src = T
		script.innerHTML = T
Markup Injection [7, 8, 204]		document.write(T)
		document.writeln(T)
		elm.innerHTML = T
		elm.outerHTML = T
		elm.insertAdjacentHTML(T)
		elm.insertAdjacentText(T)
State Manipulation [2, 204]		document.cookie = T
		localStorage.setItem(T)
		sessionStorage.setItem(T)
PostMessage Spoofing [3]		postMessage(T)

Legend: T_i= Tainted Variable.

Table A.2: Summary of primitive JavaScript sinks and semantic types supported by JAW-v3 grouped by the security risk of manipulating the sink object. The list is obtained by aggregating the client-side JavaScript sinks considered in existing literature.

Security Threat	**Semantic Type**	**Description**	**Reference**	**JavaScript Sink**
Client-side Open Redirect	WIN_LOC_WRITE	Redirecting the Window URL	[113, 114]	`window.location = T`
Websocket Hijacking	WEBSOCK_URL_WRITE	Hijacking Websocket Connections	[259, 260]	`new WebSocket(T)`
Cookie Manipulation	DOC_COOKIE_WRITE	Manipulating Cookie State	[3, 32, 54]	`document.cookie = T`
Doc. Domain Manipulation	DOC_DOMAIN_WRITE	Bypassing SOP	[160, 261]	`document.domain = T`
Client-side JSON Injection	JSON_PARSE	Parsing Untrusted JSON	[53, 262, 263]	`JSON.parse(T)`
RegEx Injection	REGEX_BUILD	Injecting Regex for ReDoS	[264, 265]	`new RegExp(T)`
postMessage Manipulation	POST_MSG_WRITE	Manipulating postMessages	[3]	`window.postMessage(T)`
Local File Path Manipulation	FILE_PATH_WRITE	Manipulating Path of Read Files	[266]	`new FileReader().readAsText(T)`
Cross-site Scripting (XSS)	CODE_LOADING	Loading New Scripts	[25, 32, 88]	`script.src = T`

	CODE_EXEC	Executing Arbitrary JavaScript	[32, 78]	`script.textContent = T`
				`eval(T)`
				`setTimeout(T)`
				`setInterval(T)`
				`new Function(T)`

	DOM_NODE_INJECT	Injecting DOM Elements	[8, 32, 53, 78]	`document.write(T)`
				`document.writeln(T)`
				`elm.innerHTML = T`
				`elm.outerHTML = T`
				`elm.insertAdjacentHTML(T)`
				`elm.insertAdjacentElement(T)`
				`elm.replaceChild(T)`
				`elm.append(T)`
				`elm.appendChild(T)`

Web Storage Manipulation	DOC_STORAGE_WRITE	Manipulating Storage State	[3, 32, 54]	`localStorage.setItem()`
				`sessionStorage.setItem()`

Client-side Request Forgery	REQ	Manipulating Asynchronous Reqs.	[P1, 1]	`fetch(T)`
				`XMLHttpRequest.open(T)`
				`asyncRequest(T)`
				`$.ajax(T)`

Legend: T= Tainted Variable;

Table A.3: List of HTML tags used in §5.1.2.1 that share the same DOM Clobbering behaviour.

Name	</> HTML Tags
TS1	a, abbr, acronym, address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, iframe, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp
TS2	blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, plaintext, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, svg, table, template, textarea, time, title, track, tt, u, ul, var
TS3	button, fieldset, input, output, select, textarea
TS4	image, img, object
TS5	a, abbr, acronym, address, applet, area
TS6	basefont, bgsound, blink
TS7	noembed, noframes, noscript, script, style, template, textarea, title, xmp
TS8	ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, object, ol, optgroup, option, output, p, param, picture, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, section, select, shadow, slot, small, source, spacer, span, strike, strong, sub, summary, sup, svg, table, time, track, tt, u, ul, var, video, wbr
TS9	fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image
TS10	form, iframe, image, img, script, style, table, template
TS11	caption, col, colgroup, tbody, td, tfoot, th, thead, tr
TS12	p, param, picture, plaintext, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp, a, abbr, acronym, address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i
TS13	h1, header, hgroup, hr, i, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, plaintext, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp, a, abbr, acronym, address, applet, area, article, aside, audio
TS14	form, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer
TS15	data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, plaintext, pre, progress, q, rb, rp, rt, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, rtc, ruby, s, samp, script, section, select, shadow
TS16	rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp, a, abbr, acronym, address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image
TS17	br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i
TS18	address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir
TS19	form, iframe, script, style, template
TS20	image, img, input, noembed, noframes, noscript

Table A.4: Overview of the categorization of the affected cross-site requests and types of third-party functionalities.

IAB ID	**Category**	**Sub-categories**	**# Requests**	**# Cookies**	**# Websites**
IAB19	Technology & Computing	File Sharing, Web Search, Email / Chat / Messaging, Data Centers, Desktop Publishing	10,026	37,612	188
IAB3	Business	Advertising, Marketing, Business Software	6,354	19,917	186
IAB12	News / Weather / Information	News / Weather / Information	1,969	10,795	32
IAB9	Hobbies & Interests	Video & Computer Games, Freelance Writing / Getting Published, Photography	1,365	10,508	34
IAB5	Education	Distance Learning	1,120	10,705	23
IAB1	Arts & Entertainment	Books & Literature, Movies, Music & Audio, Television & Video	867	4,341	25
IAB4	Careers	Job Search	399	2,005	21
IAB6	Family & Parenting	Babies & Toddlers	351	1,918	18
IAB24	Uncategorized	Uncategorized	303	1,025	17
IAB21	Real Estate	Buying / Selling Homes	62	489	10
IAB22	Shopping	Content Server, Streaming Media, Adult Content, Contests & Freebies	57	411	17
IAB14	Society	Social Networking, Weddings	46	283	13
IAB18	Fashion	Jewelry, Clothing	40	142	16
IAB11	Law, Government, & Politics	Politics	12	55	8
IAB13	Personal Finance	Credit / Debit & Loans	9	22	5
IAB2	Automotive	Buying/Selling Cars	8	14	4
IAB7	Health & Fitness	Exercise / Weight Loss	4	9	3
Total	16	32	22,992	89,743	211

Table A.5: Top ten invalid SameSite cookie policies in Alexa top 500K sites.

Invalid Policies	**# Websites**
SameSite=secure	287
SameSite=1	245
SameSite=true	138
SameSite=undefined	124
SameSite=;	106
SameSite=	72
SameSite=false	68
SameSite=-1	55
SameSite:Lax	53
SameSite=0	40

Table A.6: Overview of the IdPs that enable bypass of the new default SameSite cookie policy and the number of affected websites.

IdP	**Vuln.**	**# Websites**
Google	●	3,450
Amazon	○	679
Facebook	●	3,328
Apple	○	1,593
Microsoft	●	1,921
Linkedin	●	983
GitHub	●	198
Twitter	○	2,591
VK	●	1,241
Mail.ru	○	49
Twitch	○	168
Yahoo	○	379
Instagram	○	1,485
Total Vuln.	6	4,935
Total	13	9,485

Legend: ● = vuln. ; ○ = not vuln.

Table A.7: SameSite cookie policy inconsistencies for different user-agents grouped by site popularity.

Testbed	**June 2020**	**Sept. 2020**	**April 2021**
Alexa Top 500K	5,719	9,215	9,951
Alexa Top 100K	1,903	2,949	3,242
Alexa Top 10K	339	565	645
Alexa Top 1K	64	128	138