Origin Story: A Comprehensive Lifecycle Analysis of Same-Origin Policy Bugs

Jakub Szymsza
DistriNet, KU Leuven
Leuven, Belgium
jakub@szymsza.cz

Gertjan Franken
DistriNet, KU Leuven
Leuven, Belgium
gertjan.franken@kuleuven.be

Vik Vanderlinden
DistriNet, KU Leuven
Leuven, Belgium
vik.vanderlinden@kuleuven.be

Tom Van Goethem
DistriNet, KU Leuven
Leuven, Belgium
tom.vangoethem@kuleuven.be

Mathy Vanhoef
DistriNet, KU Leuven
Leuven, Belgium
mathy.vanhoef@kuleuven.be

Lieven Desmet
DistriNet, KU Leuven
Leuven, Belgium
lieven.desmet@kuleuven.be

Abstract

The Same-Origin Policy (SOP) serves as the web's fundamental security mechanism, isolating resources between different web origins to prevent malicious cross-site interactions. Despite its critical role, the SOP lacks a unified formal specification and has evolved informally over decades, resulting in a fragmented implementation landscape prone to bypasses and inconsistencies. While prior research has examined specific SOP flaws, no study has systematically analyzed how these bugs are introduced and resolved throughout their lifecycle. We address this gap by conducting the first comprehensive lifecycle analysis of 97 SOP security bugs across Chromium and Firefox, enabled by our extensions to BugHog—a framework previously used on Content Security Policy (CSP) bugs.

Our findings reveal that 94% of all SOP bugs stem from non-security code revisions, such as functional bug fixes or the addition of new web features, which contrasts sharply with previously identified CSP bug causes. This, compounded by inconsistent bug labeling, cross-vendor disagreement on what constitutes an SOP bug, and the vast scope of the SOP, makes it exceptionally difficult for developers to foresee the security implications of their changes. The severity of this issue is further underscored by our discovery of three publicly disclosed vulnerabilities that remained active across the latest versions of Chromium, Firefox, and Safari. Beyond showing that bug patterns are not generalizable across different browser security policies, we propose actionable recommendations for standardization and improved bug categorization.

CCS Concepts

• **Security and privacy → Browser security**; **Vulnerability management**; • **Software and its engineering → Software defect analysis**.

Keywords

Same-Origin Policy, Browser Security, Bug Lifecycle Analysis

SecDev '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2602-6/2026/07
https://doi.org/10.1145/3805773.3805993

ACM Reference Format:
Jakub Szymsza, Gertjan Franken, Vik Vanderlinden, Tom Van Goethem, Mathy Vanhoef, and Lieven Desmet. 2026. Origin Story: A Comprehensive Lifecycle Analysis of Same-Origin Policy Bugs. In *Proceedings of the 2026 ACM Secure Development Conference (SecDev '26), July 5–6, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3805773.3805993

1 Introduction

Introduced by the Netscape Navigator browser in 1995, the SOP stands as one of the oldest and most fundamental web security policies [33, 49, 51, 57, 68]. With this, the browser's developers sought to establish a boundary of trust: preventing malicious scripts on one web origin from reading sensitive data hosted on another web origin [2]. This isolation was, and still is today, the primary defense against data exfiltration attacks in all modern browsers. However, since its inception, the policy has evolved significantly to match the complexity of the modern web.

On the one hand, newly introduced features and functionality that pushed the web forward also had to adhere to the SOP. What began as a simple restriction on Document Object Model (DOM) access has expanded to govern network requests (e.g., via `XMLHttpRequest` and `fetch`), local storage APIs, and media elements [49, 68]. On the other hand, newly discovered attacks warranted extensions to the SOP protection model. Among the more recent additions are SOP-related policies that enforce trusted, isolated environments in response to side-channel attacks [36, 41].

Unfortunately, while some SOP-related policies have formal specifications, such as Cross-Origin Resource Sharing (CORS) [69], this is not the case for the SOP as a whole. Instead, much of its definition remains *de facto*, shaped by years of independent browser vendor experimentation and retroactive documentation. Consequently, security researchers have repeatedly uncovered shortcomings and inconsistencies in how different browsers enforce the SOP, resulting in a history of bypasses [5, 7, 39, 40, 56, 58, 60].

While prior work has examined the SOP, no study has systematically analyzed the lifecycles of bugs compromising its enforcement. To address this gap, we conduct the first comprehensive lifecycle analysis of SOP bugs, characterizing how they are introduced and resolved. We achieve this by employing and significantly extending BugHog, an open-source framework that automates revision-level bisection based on a bug's proof-of-concept (PoC), originally used in a study on CSP bug lifecycles in browsers [25, 34].

Our analysis of 97 SOP bugs in Chromium and Firefox reveals that security bug patterns do not generalize across browser policies. Contrasting the prior CSP study, where bugs were predominantly linked to security revisions, we show that 94% of SOP bugs are introduced by non-security-related revisions. Furthermore, we identified three persistent vulnerabilities affecting the latest versions of Chromium, Firefox, and Safari, which we responsibly disclosed to the affected vendors. We also observed divergent vendor classifications of SOP issues, pointing to an urgent need for unified standards and reconciliation efforts.

In summary, we make the following contributions:

- We significantly extended the open-source BugHog framework by adding support for PoCs with simulated user interactions and custom server logic, enabling the automated analysis of complex SOP PoCs [25].
- We curated and reproduced a set of 97 unique SOP vulnerabilities to perform a comprehensive lifecycle analysis across Chromium and Firefox.
- We discovered and reported three publicly disclosed vulnerabilities persisting across the latest versions of Chromium, Firefox, and Safari at the time of evaluation.
- Finally, we propose a pragmatic path toward standardization, urging browser vendors to reconcile their divergent definitions and handling of SOP security issues.

2 Background

In this section, we describe the SOP and BugHog.

2.1 Same-Origin Policy

According to the World Wide Web Consortium (W3C), the main standards organization for the web, *"there is no single same-origin policy"* [68]. Rather than arising from a unified formal specification, the SOP has been informally adopted and iteratively evolved by browser vendors since its inception in Netscape Navigator in 1995 [33, 51]. As demonstrated by the 97 public bug reports we reproduce in this work, alongside with the academic research identifying flaws discussed in Section 5, the implementation of the SOP remains fragile and error-prone.

2.1.1 Definition. In short, the SOP restricts how a document or script loaded from one origin can interact with a resource from another. We define an origin O as a tuple (S, H, P), consisting of a URL's scheme S, host H, and port P [2, 49]. Two resources are considered to be *same-origin* if and only if all three elements of their origin tuples are identical. Table 1 illustrates this logic using `https://foo.com/home.html` (implying the default HTTPS port 443) as the reference origin.

Table 1: SOP comparison against the reference URL `https://foo.com/home.html`.

Comparison URL	Same-origin
`https://foo.com/page2.html`	✓
`http://foo.com/home.html`	✗
`https://bar.com/home.html`	✗
`https://foo.com:8080/home.html`	✗

While this fundamental definition appears straightforward, the SOP has evolved significantly over time. Since its inception, the policy has been extended by both relaxations and tightenings to balance functionality with security, some of which have been formalized in official standards. For example, to allow consenting origins to share data, the CORS mechanism was introduced, becoming a W3C Recommendation in 2014 [69]. This allows servers to explicitly opt-in to cross-origin resource sharing via specific HTTP headers. Conversely, the discovery of transient execution attacks like Spectre and Meltdown in 2018 necessitated stricter isolation primitives [43, 45]. To address these threats, new mechanisms were integrated into WHATWG specifications: the Cross-Origin Resource Policy (CORP) was added to the Fetch standard to prevent cross-origin embedding [71], while the Cross-Origin Opener Policy (COOP) and the Cross-Origin Embedder Policy (COEP) were introduced to the HTML standard to enable process-level isolation of browsing contexts [72].

In this study, we adopt a broad definition of the SOP, encompassing all security policies that enforce access controls or isolation boundaries based on a resource's origin. This scope includes DOM access (i.e., `SOP-DOM`), cross-window communication (e.g., via `window.opener`), Web API restrictions (e.g., `CacheStorage`, `IndexedDB`, `Storage`), network requests (e.g., `fetch`, `sendBeacon`, `EventSource`), HTTP cookie handling, pseudo-protocol behavior (e.g., `about:`, `data:`), and browser plugin isolation (e.g., PDF readers, Adobe Flash). While certain aspects of this landscape, such as cookie scoping and cross-origin fetch behavior [1, 71], are relatively well-defined, others, particularly pseudo-protocols, lack standardization and depend on vendor-specific implementation choices.

2.1.2 Comparison with the Content Security Policy. To contextualize our lifecycle analysis of SOP bugs against that of CSP bugs [34], we outline the fundamental differences between these two mechanisms. Although both rely on the concept of web origins, they address distinct threat models and operate in opposing directions.

While the SOP is a mandatory, always-on policy, the CSP is an opt-in mechanism designed to preserve integrity and mitigate code injection attacks, such as Cross-Site Scripting (XSS) [47, 62]. Through the `Content-Security-Policy` HTTP response header, a webpage defines an allowlist of trusted origins from which the browser is permitted to load resources (e.g., scripts, images, and stylesheets). Whereas the SOP restricts what a script can access on other web origins, CSP restricts what scripts can run on the current webpage. For example, the CSP defined by the header below prohibits any script from executing, unless it shares the same web origin as the current webpage:

```
Content-Security-Policy: script-src 'self';
```

Furthermore, their relationship to browser capabilities differs. Mechanisms like CORS are designed to relax the SOP, granting permissions that are otherwise forbidden. CSP, conversely, is designed to restrict browser capabilities, disabling features that are otherwise allowed (such as inline scripts or `eval()`). Consequently, while a rigorous CSP can prevent an attacker from executing a malicious script exploiting a SOP bypass, it does not inherently repair the logical flaws in origin isolation itself.

Like the SOP, the CSP has evolved through several iterations, introducing new directives and functionality. For example, CSP Level

3 covers automatic HTTP-to-HTTPS upgrades and framing defenses via the `upgrade-insecure-requests` and `frame-ancestors` directives, respectively [47].

2.2 BugHog

Our lifecycle analysis methodology is supported by BugHog [25], an open-source framework originally introduced to identify CSP bug lifecycles [34]. When provided with a PoC that reproduces a given (security) bug, BugHog performs automated dynamic evaluations across a range of browser revision executables to pinpoint the precise introducing and fixing revisions of that bug.

Conceptually, BugHog operates by performing a binary-like search (bisection) across the revision history of a browser within a Git or Mercurial version control system. BugHog downloads browser executables corresponding to specific revisions and executes the provided PoC against them. By observing the browser's behavior, for example by observing all sent requests, BugHog determines whether the bug is reproducible in the software state that corresponds to that revision. This automated bisection effectively narrows down the introduction and fix windows from thousands of revisions to only a single or a few causal revision(s), depending on executable availability.

The architecture of a BugHog-compatible PoCs generally consists of one or more resources (e.g., HTML pages, CSS stylesheets, or JavaScript files) that attempt to trigger the investigated security bug. To verify exploit success, the PoC communicates with the BugHog server, typically by making a network request to a specific control URL. For instance, if a SOP bypass allows a cross-origin read, the PoC would attempt to exfiltrate the stolen data, and if successful, signal bug reproduction.

BugHog was originally introduced to analyze 75 CSP bug lifecycles [34]. Here, the authors demonstrated that CSP vulnerabilities often persist for extended periods before discovery and that CSP implementations are prone to regression even from seemingly minor code changes. Furthermore, they showed that inconsistent bug handling impede secure development and remediation.

Limitations. While BugHog proved sufficient for a comprehensive evaluation of CSP bug lifecycles, it exhibits specific limitations that render a comprehensive analysis of SOP bugs difficult. These constraints primarily stem from limited support for the complex interactions required by various SOP PoCs. In this section, we detail the key shortcomings that motivated our extensions to BugHog. We successfully addressed Limitations 1 through 3 via our architectural enhancements described in Section 3.3. However, we did not mitigate Limitation 4 due to substantial engineering complexity; we discuss its impact on our results in Section 6.3.

(1) PoCs support limited user simulation. Although BugHog allows a PoC to visit multiple webpages via a URL queue, user simulation remains limited. Currently, the only way to simulate page interaction is through JavaScript API calls (e.g., clicking a button with the `click()` function). However, this is not always feasible because script execution can be blocked. Additionally, security measures in modern browsers often prevent these API calls from opening popup windows or playing media without genuine user activation [50]. Chromium bug report `40076853` is an example of an unsupported PoC, requiring the simulated user to copy text from one browser tab and paste it into another [9].

(2) PoCs cannot open multiple tabs. This restricts how BugHog can evaluate context isolation, a crucial aspect of SOP. For instance, BugHog cannot evaluate the PoC described in Chromium bug report `40089653`, where two SharedWorkers must be opened simultaneously on two cross-origin tabs [16].

(3) Limited support for dynamic PoC webpages. While BugHog supports basic dynamic serving (e.g., responses dependent on status codes or specific headers), many SOP PoCs require more sophisticated server behavior. For example, the PoC for Chromium bug `40084396` requires an endpoint that returns different responses based on incoming request headers [15]. Similarly, bug `40051486` considers the browser vulnerable only if the request method is POST and the `Content-Type` header contains a specific value [20].

(4) Support limited to browser versions 20 and later. BugHog supports historical executables starting from version 20 (released in 2012-2013 [52, 65]) through the most recent releases—136 for Chromium and 135 for Firefox at the time of our evaluation. Although the SOP has been a core feature since both browsers' inceptions, extending support to earlier versions would require substantial engineering effort due to outdated dependencies and the frequent absence of older executables in official vendor repositories. For these reasons, we did not extend BugHog to support releases prior to version 20.

3 Methodology

In this section, we describe how we collected and analyzed 97 unique SOP security bugs, and detail our extensions to the open-source BugHog framework to accommodate complex PoCs.

3.1 SOP Bug Collection

We collected bugs fitting the definition provided in Section 2.1.1 for Chromium[1] and Firefox[2] by leveraging their publicly available bug tracking platforms. Although these platforms provide extensive querying options (e.g., filtering by affected software components or security policy), we found certain report attributes to be inconsistently assigned, which was also indicated in the original BugHog study [34]. For example, while the Chromium bug tracking platform allows triagers to label reports with affected components relevant to our study, such as `component:SOP` or `component:CORS`, this was only effectively done for a small subset of related bugs. Similarly, Firefox's platform allows reports to be labeled as `DOM: Security`, but this practice is also inconsistent.

To address these shortcomings, we constructed platform-specific queries focusing on keyword searches for `sop`, `cors`, `cross-origin`, or `same-origin` in the title. Since Chromium reports appear to be labeled as security issues much more consistently, we used the `type:vulnerability` and `status:fixed`[3] label to filter out the many false positives. Conversely, for Firefox, we restricted the results to bug reports associated with the `Core` product (i.e., the Firefox browser) to exclude browser extensions, plugins, or other

[1] https://issues.chromium.org
[2] https://bugzilla.mozilla.org
[3] According to Chromium's guidelines, only *fixed* security issues are eventually made public [24].

Mozilla software. Both queries are listed in Section A.1. Additionally, we included relevant cross-referenced reports (e.g., a Firefox report linked within a Chromium SOP bug report) that were not captured by our queries.

This yielded 242 reports across both platforms, including many false positives, i.e., bug reports unrelated to SOP. After manually filtering out these false positives, we identified 104 confirmed unique SOP bugs. Ultimately, we were able to reproduce 102 of these reported bugs, and we could complete lifecycle analyses on 97 of them. A detailed overview of the reasons for exclusion is provided in Section A.2.

All PoC scripts were transpiled to ECMAScript 5 to ensure compatibility with older browser executables, consistent with the methodology of the original BugHog study [34]. Due to BugHog's version support floor of version 20 for both browsers, the precise introduction date remains indeterminate for 15 security bugs (11 in Chromium, 4 in Firefox) that originated in earlier versions. These bugs are excluded from the analysis in Section 4.1.1, and their introducing revisions are excluded from the analysis in Section 4.3.

3.2 Bug Lifecycle Analysis

BugHog's bisection typically reduces each introduction and fix window to a small candidate range, but identifying the exact causal revision can still be challenging (e.g., when executables for specific revisions are unavailable). In Chromium, these windows usually comprised only a few revisions; in Firefox, however, they sometimes spanned 50–300 revisions due to missing builds. We therefore manually inspected these ranges: in most cases, the causal revision could be identified quickly by searching revision titles for relevant keywords such as `origin` and `cors`, or terms related to the affected browser feature. This revision identification was performed by a single expert, as were the classifications of all bugs and bypassing features (both discussed in Section 4.2), given the objective nature of these labels.

To enable a direct comparison with the prior BugHog study on CSP, we adopted the same developer revision intention taxonomy (see Section D.3). Two experts independently labeled the intention of each introducing and fixing revision using this taxonomy, after which disagreements were discussed and resolved. Inter-annotator agreement, measured using Cohen's kappa, was 0.66, indicating substantial agreement. The follow-up discussion further suggested that remaining discrepancies were minor and largely attributable to slightly different interpretations of closely related intention labels (e.g., `Update a feature` versus `Enable a feature`).

3.3 BugHog Extensions

In this section, we detail how we extended BugHog to address Limitations 1 through 3, as outlined in Section 2.2. All extensions have been merged into the public BugHog repository [25].

3.3.1 Complex User Simulation. Although we initially considered using standard remote browser control protocols such as WebDriver BiDi or the Chrome DevTools Protocol (CDP) [8, 38], their support varies substantially across different browser versions (e.g., Firefox only adopted WebDriver BiDi in version 129). This inconsistency makes them unsuitable for evaluating bugs across a wide range of historical releases. We therefore adopted a browser-independent approach based on `PyAutoGUI`, a Python module that controls mouse and keyboard input at the operating-system level.[4] Because it operates outside the browser, this approach remains compatible across vendors and versions. To integrate user interaction into PoCs, we allow each PoC to include an interaction script (a `script.cmd` file). BugHog parses this script and translates it into a sequence of `PyAutoGUI` actions. Supported commands cover common tasks such as navigating to a URL, clicking predefined UI elements on PoC webpages, and issuing keystroke sequences. We document the full command set in Appendix B.

Within this command set, we implemented a `NEWTAB` instruction to enable multi-tab workflows within a single browser session. Unlike the standard `NAVIGATE` command, which launches a fresh browser instance for each URL, `NEWTAB` opens a target URL in a new tab of the existing window.

This extension proved critical for our study, as 27 of the 102 reproduced SOP PoCs relied on simulated user interaction.

3.3.2 Dynamic Server Responses. While BugHog previously supported static response header configuration, many SOP PoCs require the server to react dynamically to request attributes such as methods or headers. To address this, we extended BugHog's Flask-based backend to support custom server-side logic.

PoCs can now include a Python function that accepts a Flask `Request` object and constructs a tailored response, including status codes, headers, and body content. The manager container dynamically imports and executes this function upon receiving a request to the configured endpoint, enabling scenarios such as returning a `307 Redirect` only when a specific `Content-Type` is present, or reflecting input headers in the response body.

This feature was essential for reproducing 21 of the SOP bugs. We note that since this mechanism permits arbitrary code execution, any public deployment would require strict sandboxing to mitigate security risks.

3.3.3 Other Enhancements. Beyond the essential extensions discussed above, we implemented various usability and performance improvements, and fixed several bugs. These include fixing syntax highlighting in the PoC editor, populating new experiment files with sensible default values, and improving the framework's runtime performance.

4 Results

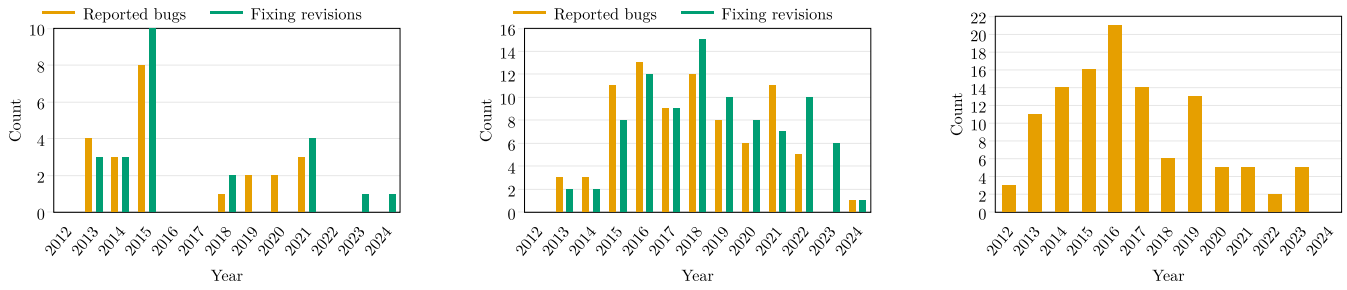
In this section, we present the findings from our lifecycle analysis using the enhanced BugHog framework.

4.1 Lifecycle Characteristics

In this section, we analyze bug lifecycle interval lengths and whether bugs are potentially foundational.

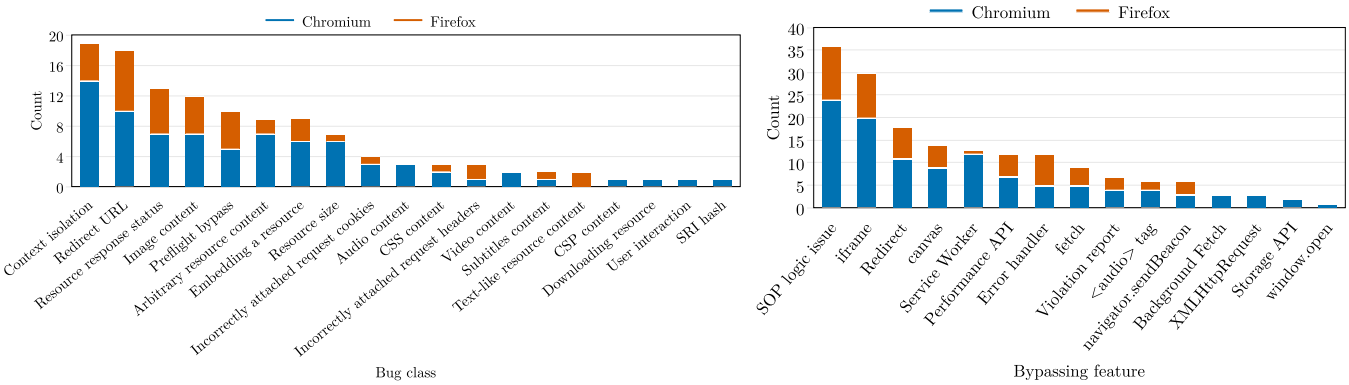
4.1.1 Lifecycle Interval Lengths. As for the latency between bug introduction and reporting, we observe medians of 346 days (0.9 years) for Chromium and 451 days (1.2 years) for Firefox. While the Firefox figure aligns closely with prior findings for CSP bugs (1.2 years), Chromium deviates substantially from its CSP counterpart

[4] https://github.com/asweigart/pyautogui



(a) Potentially foundational bugs. **(b) Non-foundational bugs.** **(c) Introduced non-foundational bugs.**

Figure 1: Distribution of reported and fixed bugs per year.



(a) Bug classes. **(b) Bypassing web mechanisms.**

Figure 2: The distribution of bug classes and bypassing web mechanisms.

(2.9 years) [34]. This difference is noteworthy given that the SOP is a much older policy than CSP;[5] yet, SOP bugs remain latent for significantly shorter periods.

Regarding fix latency, the median time between report and fix is 19 days for Chromium and 52 days for Firefox. In comparison, the CSP study reported medians of 44 days and 52 days, respectively [34]. While Firefox exhibits consistent fix times across both policies, Chromium demonstrates a 2.3 times faster response for SOP bugs.

A plausible explanation for both Chromium discrepancies (shorter introduction-to-report and report-to-fix times) is that SOP vulnerabilities are generally perceived as more important than CSP vulnerabilities, because the CSP is considered a defense-in-depth mechanism. This hypothesis is supported by the assigned severity labels and bounty rewards discussed in Section 4.4.3.

We refer to Appendix C for cumulative distribution functions of both lifecycle intervals.

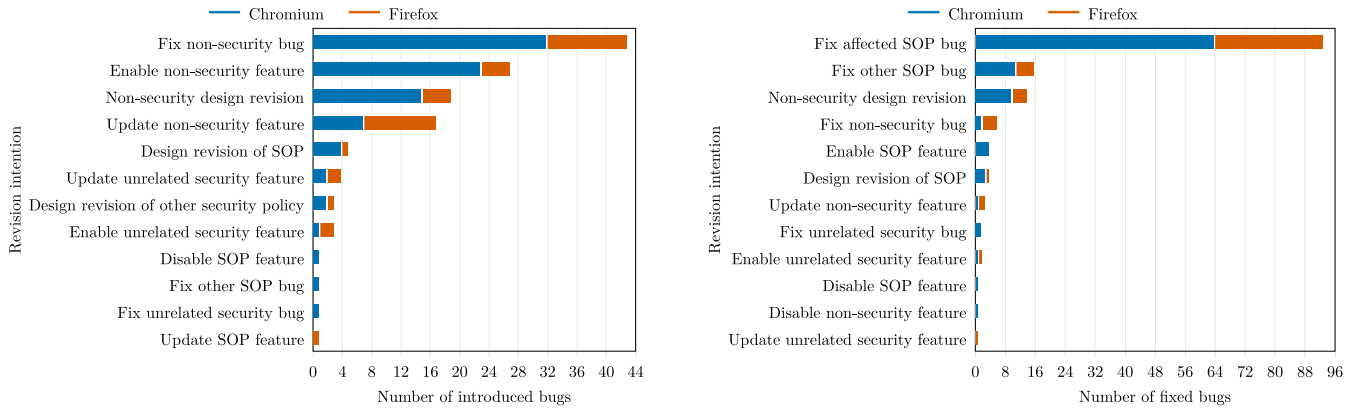
4.1.2 Potentially Foundational vs. Non-foundational Bugs. In their study of CSP lifecycles, the authors defined *foundational bugs* as those present since the policy's inception [34]. As detailed in Section 3, BugHog's version limitations prevent us from identifying the introducing revision for 15 SOP bugs that predate Chromium and Firefox version 20. Consequently, we classify these early cases as *potentially foundational bugs*.

Figure 1a indicates that while the majority of potentially foundational bugs were reported and fixed before 2016, six persisted until at least 2020—implying a lifespan of over eight years. Notably, the proportion of potentially foundational bugs (15% for Chromium and 38% for Firefox) is significantly lower than that observed for CSP, where such bugs constituted half the dataset.

Figure 1b presents the timeline for non-foundational bugs. Despite the SOP's expanding attack surface, driven by the addition of new sub-policies and web features, we observe a discernible decline in reported bugs from 2016 onwards. This trend is corroborated by Figure 1c, which depicts the number of bugs introduced per year. While the drop in recent years is likely an artifact of disclosure lag (as security reports typically remain private for months after a fix), the overall downward trajectory stands in contrast to prior CSP findings. The decline in SOP bug introductions may indicate that SOP implementations are maturing. However, many bugs introduced in recent years likely remained undiscovered at the time of the evaluation.

> **Section takeaway:** SOP bugs in Chromium have substantially shorter lifecycles than their CSP counterparts, and foundational bugs are far less prevalent overall.

[5] Chromium supports CSP since 2012.



(a) Introducing revisions. **(b) Fixing revisions.**

Figure 3: Developer intentions of revisions that introduced or fixed a SOP bug.

4.2 Bug Classification

To enable a more fine-grained analysis of the observed bugs, we classify them along two dimensions: (i) the type of SOP violation (i.e., what is leaked or which isolation boundary is broken) and (ii) the web mechanism exploited in the PoC to trigger the bypass. Because a single PoC may leverage multiple web mechanisms, the total count of exploited mechanisms exceeds the total bug count. Full labeling details are provided in Sections D.1 and D.2. Note that, due to fundamental differences in enforcement models, we do not compare these classifications directly with the CSP study.

We assign each bug to exactly one bug class, capturing either the resource type whose content/metadata is leaked or the isolation boundary that is violated. Figure 2a shows the resulting distribution, and Section D.2 provides the corresponding definitions. The most frequent classes involve executing JavaScript in a cross-origin browsing context (i.e., `Context isolation`) and leaking redirect URLs. Overall, the distribution highlights the breadth of SOP protections: beyond classical data reads, violations include leaks of niche resource types (e.g., subtitles) and even whether user interaction occurred in a cross-origin context (i.e., `User interaction`).

In addition, we classify the specific web mechanism used in each PoC to bypass the SOP; Figure 2b summarizes this distribution and Section D.2 lists the definitions. The largest category, present in more than 30% of cases, consists of SOP logic issues, which reflect broad implementation flaws that manifest across multiple mechanisms. Conversely, nearly 70% of the bugs are tied to individual web features rather than general enforcement logic, suggesting that SOP checks are effectively distributed across a wide range of subsystems. This highlights the potential value of centralized, network-level enforcement mechanisms, such as Opaque Response Blocking (ORB), to mitigate these dispersed risks [67].

> **Section takeaway:** The most frequent SOP violation types are context isolation and redirect URL leaks, and the majority of bugs are tied to individual web features rather than general enforcement logic.

4.3 Revision Analysis

In this section, we investigate bug introducing and fixing revisions.

4.3.1 Revision Intent. To understand the reasons behind the revisions that introduce and resolve bugs, we analyze the developer intentions of the associated revisions. Figure 3a presents the distribution of all intentions of the 125 revisions (89 for Chromium and 36 for Firefox) that introduced a SOP bug. Although our dataset contains only 97 unique bugs, the number of introducing (and fixing) revisions is higher because some bugs were reintroduced (and re-fixed) multiple times. Conversely, bugs introduced prior to version 20 are excluded, as their introducing revisions predate our analysis window (see Section 3.1).

Interestingly, the intention behind 94% of the introducing revisions was unrelated to the SOP. The most common cause of SOP bugs was attempts to *fix non-security bugs*, typically in the mechanisms used for requesting resources. In Chromium, the second most frequent cause was the *enabling of new browser features*, such as the Service Worker or Performance API, without fully considering their impact on the SOP. In contrast, the second most common cause in Firefox was *updating existing non-security features*, for instance, adding a new type of filter for `canvas` elements.

These findings contrast sharply with those of the CSP study, where most bugs were introduced during the initial CSP rollout (i.e., foundational bugs), while adding specific CSP features, or while fixing CSP-related bugs [34]. Furthermore, the SOP appears to be less brittle: fixing an SOP bug introduced a new vulnerability in only a single instance. Conversely, bug fixes were the third most common source of new vulnerabilities in the CSP lifecycle analysis.

Figure 3b shows the intentions behind the 147 revisions (100 for Chromium and 47 for Firefox) that fixed a SOP bug. Of these, 63% aimed to *fix the affected SOP bug* intentionally, which closely aligns with the 61% observed during the CSP study. The proportion of revisions that aimed to fix a different bug affecting the same policy is also similar (11% for SOP vs. 13% for CSP). The remaining fixes generally resulted from design changes, the resolution of non-security bugs, or the introduction of new policy features.

4.3.2 Revisions Affecting Multiple Bugs. Our analysis identified several revisions responsible for introducing multiple distinct bugs. Two revisions each introduced three separate bugs: one revision

implemented the `navigator.sendBeacon` method and the other enabled resource preloading via the `<link>` element. Twelve additional revisions each introduced two bugs. In every case, the primary intention was to implement or modify non-security features. Common triggers included enabling new Web APIs (e.g., Service Workers, Background Fetch) or resolving functional defects by expanding error messaging or relaxing existing restrictions. This contrasts sharply with findings from the CSP study, where every revision introducing multiple bugs was explicitly intended to modify CSP logic itself, either by fixing another bug or by adding policy functionality [34].

Regarding bug fixing revisions, nine were found to fix two bugs each. In eight of these cases, the developer intended to fix a specific SOP vulnerability and incidentally resolved a second one. The exception was a Chromium revision designed to mitigate user tracking by strengthening cross-origin cookie protections.

> **Section takeaway:** Nearly all SOP bugs are introduced through non-security changes, and fixing an SOP bug rarely introduces a new SOP bug—making the SOP less brittle than the CSP.

4.4 Bug Reports

In this section, we evaluate the quality of all bug tracking reports: 73 reports for Chromium and 36 for Firefox.

4.4.1 Classification. Our analysis shows that current classification mechanisms in bug tracking platforms are insufficient for effectively identifying SOP bugs. Although Chromium allows reports to be assigned to multiple components, only 19% of SOP bugs were correctly categorized under relevant components,[6] a figure significantly lower than the 43% observed for CSP reports in prior work [34]. Firefox, meanwhile, lacks dedicated SOP or CSP components entirely, instead categorizing reports based on the specific web feature used to bypass the policy.

Title-based filtering proved similarly ineffective. Relevant SOP keywords[7] appeared in only 25% of Chromium and 19% of Firefox report titles. This again contrasts with the CSP study, where only four reports lacked both the CSP component assignment and CSP-related keywords in the title [34]. We attribute this disparity to the fact that the SOP is an implicit, foundational browser mechanism, whereas CSP is an explicit, opt-in feature tied to a server-supplied configuration (e.g., a response header).

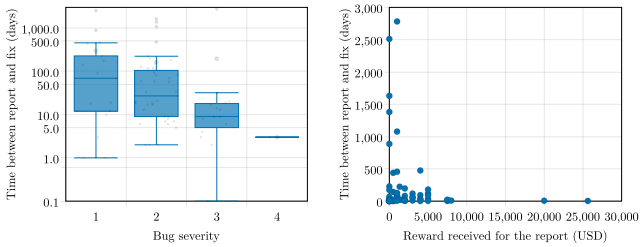
Furthermore, the platforms differ significantly in distinguishing security bugs from standard defects. Chromium correctly classified all but two reports using the `Vulnerability` type. In contrast, Firefox labels all issues as generic `Defects` and lacks a filtering option for security status; notably, only 25% of Firefox's SOP vulnerabilities were filed under the `DOM: Security` and `Security` components.

These classification shortcomings necessitate the broad keyword-based search strategy described in Section 3.1 and in general complicate efforts to identify duplicate reports.

4.4.2 Revision Linking. The bug tracking platforms are integrated with their browser's version control system, automatically listing revisions in bug report comments when the bug identifier is referenced in the revision's message. This linkage is critical for tracing bug lifecycles, yet our analysis shows it is inconsistently utilized.

Developers rarely document the revision that introduced a vulnerability, even when the cause is known. Only 7% of Chromium SOP reports contained a link to an introducing revision, while no Firefox reports did. These figures align with previous findings for CSP (4% for Chromium, 7% for Firefox), confirming that developers across both browsers consistently neglect to reference the code responsible for introducing security issues [34].

In contrast, fixing revisions are linked much more frequently. We found that 81% of reports for both browsers referenced a fix, though this is slightly lower than the rates observed for CSP (87% for Chromium, 86% for Firefox) [34]. Nonetheless, 19% of reports either lack any revision link or reference a revision that did not fully resolve the vulnerability. This gap complicates verification, increasing the risk that a vulnerability is marked as fixed prematurely—a scenario that can lead to accidental early public disclosure if the bug remains active.



(a) Time to fix - severity **(b) Time to fix - reward**

Figure 4: Correlation of the required time to implement an effective bug fix with the bug's severity and the bug's bounty reward for Chromium.

4.4.3 Bug Severity and Bounty. Figure 4 illustrates the relationship between the time required to implement an effective fix with the assigned severity of a bug and the bounty reward received, respectively. We focus exclusively on Chromium for this analysis, because Firefox does not publicly disclose reward amounts and assigned a generic *moderate* severity to 84% of its SOP bugs.

The data suggests a general trend where higher severity and larger rewards correspond to shorter fix times. However, the statistical correlation is weak; the distance correlation coefficient, which measures both linear and non-linear dependence, is only 0.2 for both metrics. This weak statistical link is largely due to the high density of bugs in the "low-severity", "low-reward", and "short fix-delay" cluster. Despite this, the system appears effective for critical issues: virtually no high-severity vulnerabilities were left unpatched for extended periods.

> **Section takeaway:** SOP bugs are systematically underclassified in bug tracking platforms, and while fixing revisions are generally linked in reports, introducing revisions almost never are.

[6] We considered `Blink>SecurityFeature>SameOriginPolicy` and `Blink>SecurityFeature>CORS` as SOP-related.
[7] The considered SOP-related keywords were `sop`, `same-origin policy`, `same origin policy`, and `cors`.

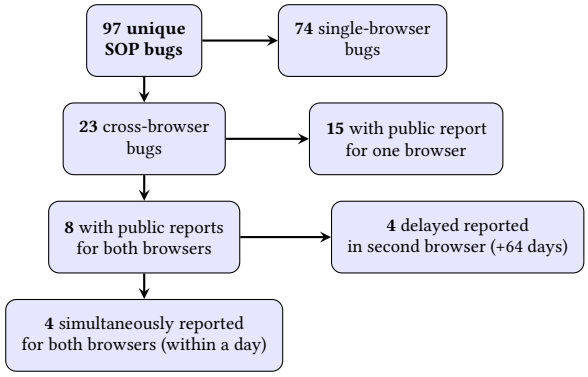


Figure 5: Breakdown of cross-browser SOP bugs.

4.5 Cross-Browser Bugs

Among all 97 analyzed bugs, 23 reproduced in both Chromium and Firefox. Figure 5 shows a summarized flow chart of these cross-browser bugs, while we provide a comprehensive Gantt chart in Appendix E. As shown, only eight of the 23 bugs had reports filed in both browsers; the remaining 15 either never had a bug report filed or the report was not made public.[8] For the eight bugs where both reports are available, four were filed simultaneously by the same reporter after testing in both browsers, with at most one day between submissions. The remaining four reports were filed between 64 days and 1.6 years apart, despite the bugs already being present in both browsers at the time of the earlier report. This suggests that testing exploits across multiple browsers could significantly reduce exposure time for undetected vulnerabilities.

The lifecycles of cross-browser SOP bugs differ substantially between browsers. The median time between a bug introduction in one browser and its introduction in the other was 295 days, while the median gap between fixes was even 2.1 years. Additionally, only 30% of bug introductions and 26% of fixes shared the same developer intention across both browsers, indicating that most vulnerabilities arose from browser-specific implementation issues rather than common factors such as specification flaws. On average, Firefox remained vulnerable to these bugs 68 days longer than Chromium.

Notably, four bugs were introduced in one browser only after being intentionally fixed in the other. Furthermore, three bug reports were published while the other browser remained vulnerable to the reported issue. These findings indicate that, despite browser vendors' shared efforts—such as using the Web Platform Tests (WPT) suite to prevent regressions and facilitate knowledge sharing [63]—they failed to detect or avoid several cross-browser vulnerabilities. This creates opportunities for attackers to exploit publicly available bug reports against users of browsers where the vulnerability remains unpatched.

> **Section takeaway:** Cross-browser SOP bugs exhibit large temporal gaps between fixes (a median of 2.1 years), creating windows where public reports can be exploited against still-vulnerable browsers.

[8] Among these 15 are two Firefox reports that were not classified as security bugs and are thus excluded from this analysis.

4.6 Unpatched Bugs

The lifecycles depicted in Figure 9 (Appendix E) also reveal that five Firefox bugs and one Chromium bug still reproduce in the latest browser versions, a finding we manually verified outside BugHog. However, not all indicate active neglect: Firefox bug `1741034` is public and currently being resolved; it was disclosed because Chromium had already published the corresponding report for the same vulnerability [28]. Conversely, Chromium bugs `40053936` and `40053780` persist in Firefox because the developers do not consider them security vulnerabilities, as Firefox is not supposed to prevent iframes from redirecting their embedding page [30].

The remaining three issues were previously unreported vulnerabilities, and the affected vendors were unaware of their existence before our analysis. Consequently, we responsibly disclosed one vulnerability to Chromium and two to Firefox. Additionally, we tested these three issues on the latest version of Safari and found that two also reproduce there, prompting disclosure to Safari as well. However, as our responsible disclosure process demonstrates, there is significant disparity among browser vendors regarding what constitutes a SOP vulnerability.

4.6.1 Firefox Bug 1731614 (Status Leak via Media Errors). This bug leaks whether a cross-origin same-site resource returns a `2xx` status by observing the error message thrown by a `<video>` element, enabling state-dependent leaks such as login detection [29]. When reported to Chromium, the Chromium team treated the behavior as intended and opted not to address it, citing limited security impact and the lack of a clearly preferred behavior [23]. Safari exhibited a more severe variant where the status could be inferred cross-site. Although Apple confirmed the vulnerability and initiated an investigation, the issue remains unresolved more than six months after our initial report.[9]

4.6.2 Chromium Bug 40093907 (Status Leak via Script Events). This bug allows inferring the status of a cross-origin resource because different statuses trigger either the `load` or `error` events on the `<script>` element, affecting cross-site resources as well [18]. After responsible disclosure, Firefox implemented a patch that landed in version 139 [32]. Safari reported that it does not consider this behavior a security vulnerability, despite its similar impact to the previous issue.

4.6.3 Chromium Bug 40081583 (Origin Header Not Cleared on Redirect). This bug prevents the browser from correctly clearing the `Origin` header when a form submission is redirected, potentially enabling Cross-Site Request Forgery (CSRF) by making the redirected request appear to originate from a trusted origin [13]. For example, if `example.com` submits a POST form to `attacker.com`, which then redirects the request to `example.com/delete-account`, the affected browser fails to clear the `Origin` header, leaving its value as `example.com`. At the time of writing, the Firefox team has confirmed the issue and is working toward a fix [31].

> **Section takeaway:** Six SOP bugs remained active in the latest browser versions, three of which were previously unknown, revealing significant vendor disagreement on what constitutes an SOP vulnerability.

[9] Because Apple does not publicly disclose security bug reports even after a fix has been deployed, we are unable to provide a public reference for this vulnerability.

4.7 Early Public Disclosures

We identified two cases of early public disclosure where the bug remained active after the report became public. In the first case, a Chromium bug remained reproducible for 3.5 years after its report was published [19]. The issue had been marked as fixed based on revisions that failed to address the root cause, and was only resolved incidentally years later by a new unrelated cross-origin cookie protection. In the second case, a report filed by a Chromium developer was public from its inception because it was not classified as a security vulnerability [14], despite having clear security implications and Firefox treating the identical issue as a vulnerability [27].

We also observed a significant "near miss" in Chromium, where a security fix was reverted due to automated test failures after the bug was marked as fixed [12]. The fix was successfully reintroduced shortly before the report was scheduled for automatic publication. However, any delay in re-applying the patch would have risked exposing an active vulnerability, given Chromium's policy of automatic disclosure 14 weeks after a fix assignment.

Overall, while the bug handling processes for the SOP are generally robust, these incidents indicate that stricter adherence to verification protocols and more consistent classification of internal reports could further reduce the window of exposure for users.

> **Section takeaway:** Early public disclosures and near-misses indicate that stricter fix verification and more consistent security classification could reduce user exposure windows.

5 Related Work

5.1 SOP Flaws

In 2011, Saiedian et al. argued that SOP contained fundamental security flaws and required a comprehensive redesign [57]. They highlighted the risks posed by common developer bypasses, such as *JSON with Padding*, and explained the policy's inability to prevent prevalent attacks like CSRF and Cross-Site Scripting (XSS). Instead of a fundamental redesign, however, the web ecosystem eventually addressed these vulnerabilities through auxiliary mechanisms, including the `Origin` header, `SameSite` cookies, and CSP [3, 48, 62].

Prior research has scrutinized the consistency of SOP's enforcement. Braun highlighted the complexity of implementing the SOP correctly, identifying several bypasses through systematic testing [5]. Similarly, Schwenk et al. analyzed cross-site DOM access, finding widespread inconsistencies across browser implementations and concluding that the absence of a formal SOP definition is the primary cause of these discrepancies [58].

Moving beyond implementation inconsistencies, recent works have exploited architectural gaps in SOP enforcement. Rogowski et al. demonstrated that SOP is merely a logical boundary within the scripting engine that does not extend to the underlying process memory [56]. By introducing memory cartography, they executed data-only attacks to locate and manipulate internal security flags, effectively disabling SOP. Chen et al. uncovered a protocol-level SOP bypass stemming from a mismatch between the browser's strict, URI-based origin definition and the looser, certificate-based authority model in HTTP/2 Server Push and Signed HTTP Exchange [7].

Similarly, DNS rebinding exploits a mismatch in SOP, which relies on domain names for security boundaries but technically enforces them based on IP addresses. To bypass defenses like DNS pinning, Karlof et al. introduced *dynamic pharming*, where attackers force a browser to re-resolve a domain to a legitimate server's IP after loading malicious code, allowing session hijacking even over SSL [40]. Furthermore, Johns et al. demonstrated that the HTML5 Application Cache could persist malicious scripts until DNS pins expire, rendering standard cache-based defenses ineffective [39].

5.2 Cross-Site Leaks

Cross-site leaks (XS-Leaks) are a class of web vulnerabilities that allow an attacker to infer sensitive information about a user on another origin, effectively acting as a side-channel attack against the SOP. Recent work has formalized these attacks to better understand their mechanics; Knittel et al. defined XS-Leaks as a combination of a state-dependent resource, an inclusion method, and a leak technique, using this model to discover 14 new attack classes [42]. Van Goethem et al. extended this model to classify leaks based on affected browser components, highlighting the need for defenses that combine resource isolation with strict state partitioning [66].

Other studies focused on automated leak discovery frameworks. Rautenstrauch et al. introduced *observation channels* to model information leakage, automatically identifying 280 distinct vectors across major browser engines and revealing that 77% of tested top sites remain vulnerable to login detection [55]. Noß et al. developed `AutoLeak`, which translates the DOM into directed graphs to detect structural differences between states, uncovering thousands of unique leak techniques including novel header-based vectors [53].

5.3 Other Security Policy Flaws

Prior studies have uncovered numerous flaws in browser security mechanisms beyond SOP and XS-Leaks. CSP implementations have proven to be inconsistent and prone to bypasses [34, 60, 70, 73]. Cookie security mechanisms have suffered from similar weaknesses, with gaps in SameSite enforcement and susceptibility to cross-site leakage [35, 61]. Broader analyses have further documented widespread disparities in security and privacy headers, including CORS misconfigurations, erratic header enforcement, and divergent implementations of web standards across browsers [4, 6, 37, 54, 59].

6 Discussion

In this section, we contextualize our SOP lifecycle analysis with the prior CSP study, and propose actionable steps to improve the development process in relation to the SOP. Finally, we discuss potential threats to the validity of our findings.

6.1 Comparison with CSP Lifecycle Analysis

Comparing the results of our SOP lifecycle analysis with those of the CSP study challenges the assumption that browser security policies can be treated uniformly: the primary drivers for bug introduction differ significantly between the two. While the prior CSP study found that vulnerabilities were predominantly foundational or introduced by revisions intended to modify CSP logic itself [34], our analysis reveals that 94% of SOP bugs stemmed from revisions unrelated to security. Consequently, heuristics that predict vulnerabilities based on security-related commit intentions which may be effective for CSP, are likely to fail for the SOP.

Furthermore, the significantly shorter median durations for introduction-to-report (0.9 vs. 2.9 years) and report-to-fix (19 vs. 44 days) in Chromium indicate that its developers prioritize identifying and resolving SOP bugs over CSP issues. Similarly, ethical hackers may be more incentivized to hunt for SOP bugs, as these typically command higher bug bounties. This prioritization is logical: the SOP is the web's fundamental security boundary, whereas CSP is generally regarded as a defense-in-depth mechanism.

However, our analysis reinforces several critical issues previously identified for CSP. For instance, bug report labeling remains inconsistent, with SOP reports showing even lower reliability than CSP reports. This complicates the proper handling of bugs, as exemplified by the discovered early disclosures and unpatched bugs.

6.2 Improving SOP Enforcement

The findings of this study suggest that the current approach to SOP enforcement is unsustainable given the policy's vast attack surface. Although the analysis in Section 4.1.2 implies a gradual maturing of SOP implementations, the fact that the majority of vulnerabilities stem from non-security feature updates and non-security bug fixes indicates that developers often fail to foresee the collateral impact of their changes. Because the SOP intersects with virtually every web mechanism, SOP compliance checks should be integrated more rigorously into any mechanism's design process and regression testing, ideally through collaborative efforts between browser vendors. Moreover, adopting a standardized bug labeling taxonomy would significantly aid developers and researchers in tracking issues affecting the SOP.

Furthermore, the existing ambiguity regarding the SOP's intended behavior, evidenced by the lack of vendor consensus on the security classification of identical behaviors, highlights the need for more standardization. While fully formalizing the SOP is impractical due to its complexity and historical baggage, a pragmatic approach remains viable. For example, formalization efforts could prioritize functionalities where browser implementations diverge, such as the conflicting responses to identical bug reports identified in our study. By resolving these specific inconsistencies, vendors can establish a unified security baseline without the burden of specifying the entire policy.

Finally, adopting centralized enforcement mechanisms can help mitigate both current and future vulnerabilities. Unlike feature-specific patches, centralized defenses do not require tailored SOP protections for every individual browser component. For instance, ORB acts as a network-level filter, blocking potentially dangerous cross-origin responses before they reach client code [67]. This effectively prevents side-channel leaks, providing a safety net even when specific browser components fail to fully adhere to SOP constraints.

6.3 Threats to Validity

Our focus on evaluating Chromium and Firefox limits the generalizability of our findings to the broader browser ecosystem, particularly regarding mobile and WebKit-based browsers. Similarly, we only gathered bug reports from their bug tracking platform, as most other vendors do not openly disclose security bug reports. For example, WebKit security issues are rarely made public even after they are fixed. Additionally, our analysis of developer intentions is potentially skewed by the practice of obfuscating security patches to prevent early exploitation; some revisions classified as generic bug fixes or refactoring may have been intended to silently resolve vulnerabilities [26]. Finally, because BugHog is restricted to evaluating browsers from version 20 onwards, we could not determine the exact introducing revision for 15 of the 97 unique SOP bugs. Although this limitation likely leads to an underestimation of bug lifetimes and excludes the introducing revision intentions for these cases, we remain confident that our findings are representative, as these instances constitute a minority of the dataset.

7 Conclusion

In this work, we present the first comprehensive bug lifecycle analysis of the SOP, evaluating 97 bugs across Chromium and Firefox by extending the open-source BugHog framework. Our findings reveal a fundamental divergence from previous research on the CSP. While most CSP bugs are foundational or stem from direct modifications to policy logic, SOP violations most often arise as side effects of unrelated, non-security code updates. This discrepancy challenges the generalizability of browser security bug models and highlights the fragility of the SOP: because the policy governs virtually every web mechanism, changes to seemingly unrelated components can inadvertently introduce violations.

This complexity is compounded by inconsistent bug handling practices and a lack of consensus on classification among browser vendors: a behavior considered a SOP violation in one browser may not be treated as such in another. These gaps are exemplified by our discovery of three publicly known vulnerabilities that persisted in the latest releases of major browsers, all of which we have reported. Given that SOP violations pose significant security risks and warrant bounties up to $25,633, more robust handling is required. Concretely, we urge browser vendors to adopt more rigorous, standardized bug report labeling, thereby facilitating analysis by both developers and researchers. More fundamentally, we advocate for formalizing informal aspects of the SOP into a shared specification among vendors, prioritizing the reconciliation of known behavioral discrepancies between browsers.

Future Work. Our extensions to BugHog not only enabled the analysis of SOP bug lifecycles, but also paved the way for evaluating other bug classes that require complex user simulation. Future work could leverage this capability to map the origins of broader security vulnerability categories. Furthermore, as research into LLM-aided vulnerability detection and management accelerates [44, 46, 64, 74], datasets of verified bug-introducing commits could provide a valuable ground-truth training set. Ultimately, integrating tools like BugHog directly into CI/CD pipelines, combined with deeper analysis of bug origins, holds significant promise for advancing secure software development practices.

Acknowledgements

We thank the reviewers for their valuable feedback. This research is partially funded by the Internal Funds KU Leuven, and by the Cybersecurity Research Program Flanders.

We utilized Gemini 3 Pro to refine the grammar, spelling, and readability of the text, while preserving the original meaning.

A Bug Report Filtering Details

A.1 Bug Report Queries

We used the following URL queries to find SOP-related bugs.

Chromium https://issues.chromium.org/issues?q=title:(cross-origin%20%7C%20same-origin%20%7C%20cors%20%7C%20SOP)%20type:vulnerability%20status:fixed

Firefox https://bugzilla.mozilla.org/buglist.cgi?classification=Client%20Software&classification=Developer%20Infrastructure&classification=Components&classification=Server%20Software&classification=Other&short_desc_type=anywordssubstr&resolution=FIXED&keywords=sec-critical%2C%20sec-high%2C%20sec-low%2C%20sec-moderate%2C%20sec-other&query_format=advanced&short_desc=cross-origin%2Csame-origin%2CCORS%2CSOP&bug_type=defect&keywords_type=anywords&product=Core

A.2 Overview of Excluded Bugs

Figure 6 provides an overview of all SOP bug reports returned by our query, and Table 2 defines the exclusion categories. Of the 104 unique SOP bugs identified, we were unable to reproduce two: Chromium bugs `323695987` and `40080211` [11, 22]. The former exploits the newly enabled `MessagePort.close` event to leak cross-origin popup window navigation. As the PoC relies on triggering garbage collection, the reproduction failure is likely due to subtle differences in memory management between BugHog's Docker container environment and a standard desktop operating system. The latter attempts to exploit JavaScript object prototypes to access a cross-origin document; however, we could not identify a clear cause for the reproduction failure.

Of the 102 reproducible bugs, we could not fully determine the lifecycle for five. Specifically, Chromium bugs `40091708` and `40057097` required enabling the download of insecure Python files and multiple concurrent files, respectively [17, 21]. While these options were configurable in the specific browser versions where the bugs were initially reported, enabling them consistently across the entire version history would have required complex modifications to BugHog's browser initialization process. Finally, the lifecycles of three Chromium bugs were excluded from further analysis as their execution was inherently unreliable. For instance, one such bug relied on a race condition involving Linux file descriptors to read local file content, making consistent reproduction impossible [10].

B BugHog Extension: Interaction Commands

The following list outlines the new PoC commands introduced to simulate user interaction:

- **`NAVIGATE url`** behaves similarly to an entry in the URL queue; namely, it launches a new browser process with the specified URL and waits for several seconds. Importantly, the process is not terminated until the experiment ends or another `NAVIGATE` command is executed.
- **`NEW_TAB url`** opens the given URL in a new tab of an existing browser process and waits for several seconds.
- **`CLICK_POSITION x y`** simulates a mouse click on the provided position, either specified by absolute coordinates or as a percentage of the screen dimensions.

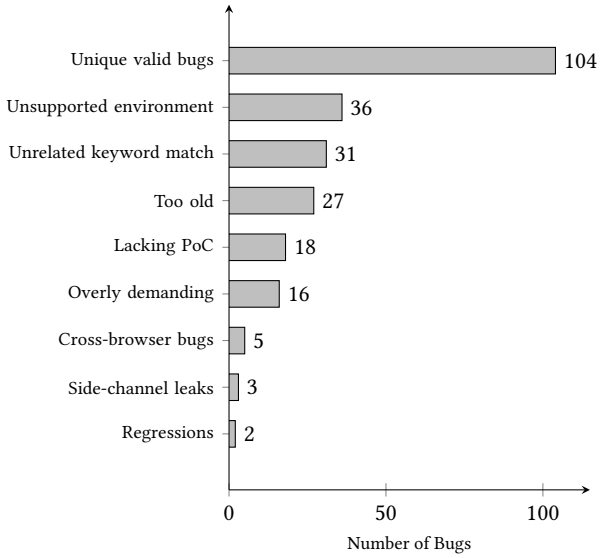


Figure 6: Overview of considered SOP bug reports.

- **`CLICK id`** imitates a mouse click on the element with the specified identifier. If the element is not found on the currently active page, an exception is thrown, and the experiment is marked as an error.
- **`WRITE text`** mimics the user typing the given string on a keyboard with a 0.1-second delay between individual keystrokes.
- **`PRESS key`, `HOLD key`, `RELEASE key`** simulates the user pressing, holding, and releasing the provided key, respectively.

Table 2: Classification of considered bug reports.

Report category	Description
Unique valid bugs	All unique and valid SOP bugs included in the final dataset.
Unsupported environment	Bugs requiring non-Linux OSs, browser extensions, or non-standard browser flags.
Unrelated keyword match	Reports with coincidental keyword matches where the bug does not affect the SOP.
Too old	Bugs resolved before browser version 20, predating the supported range of BugHog (see Section 3.1).
Lacking PoC	Bugs deemed unreproducible because the report lacked a PoC or working link, and the code could not be reconstructed from the description.
Overly demanding	Bugs where reproduction conditions were deemed too demanding (e.g., visual spoofing undetectable via JavaScript, or timing attacks requiring minutes per character).
Cross-browser bugs	Bugs reported by the same researcher to both browser vendors with the same PoC attached.
Side-channel leaks	Bugs that affected cross-site renderer process isolation, making them exploitable solely through side-channel attacks (e.g., Spectre [43]) rather than through the HTML of an attacker-controlled webpage.
Regressions	Bugs that were previously reported, fixed, and later reintroduced.

`HOTKEY key1 key2...` imitates the user performing the specified keyboard shortcut composed of one or multiple keystrokes. Specifically, the keys are pressed in the given order before being released in the reverse order.

`SLEEP seconds` pauses the script execution for the provided number of seconds, typically to wait for JavaScript execution to finish.

`ASSERT_FILE_CONTAINS file substring` continues the execution if the specified file in the Downloads folder includes the given substring; otherwise, the script execution is terminated and the bug is considered not reproduced.

`REPORT_LEAK` reports the bug as reproduced; typically used after an assertion command.

`SCREENSHOT file_name` captures the contents of the virtual display and saves it in the specified file, prefixed with the PoC name and suffixed with the evaluated browser identifier.

`OPEN_FILE file_name` navigates the browser to a file with the given name located in the Downloads folder.

`OPEN_CONSOLE` opens the developer tools window of the browser focused on the JavaScript console section.

C Bug Lifecycle Intervals

Figures 7 and 8 depict the cumulative distribution functions for the introduction-to-report and report-to-fix intervals, respectively.

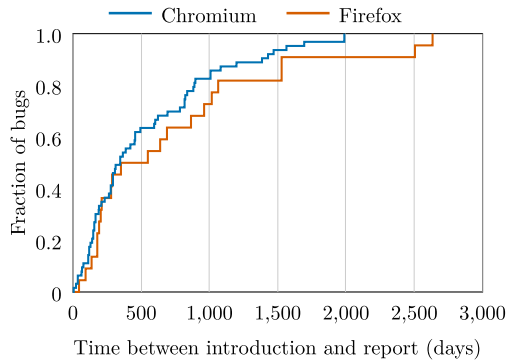


Figure 7: CDF of the introduction-to-report interval.

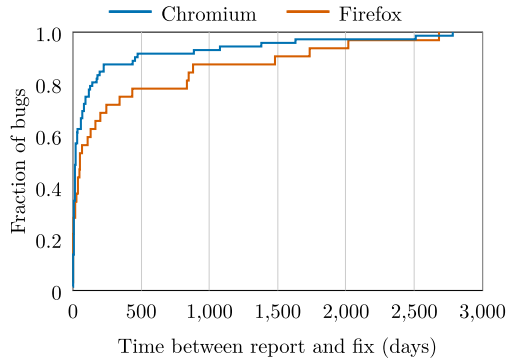


Figure 8: CDF of the report-to-fix interval.

D Analysis Labels

In this section, we list the various labels used in our analysis.

D.1 Bug Classes

Table 3 presents the classes used to classify vulnerabilities based on the type of resource that was leaked or whose integrity was violated due to the bug's presence. One option can be used per bug, and the broadest applicable category should be used. For instance, if a vulnerability allows the attacker to read an arbitrary cross-origin resource content, `Arbitrary resource content` should be used instead of `Image data`; although the bug also enables the leak of cross-origin images, the former category is broader.

D.2 Bypassing Features

Table 4 describes the mechanisms of the browser used in a PoC to exploit the SOP vulnerability. Multiple options per bug can apply. However, an option should be selected only if it is inherently tied to the bug. For instance, if both an `iframe` and `window.open` can be used similarly to exploit the vulnerability, neither of these options should be selected, as the behavior indicates an underlying `SOP logic issue`. On the other hand, if the PoC only works if an `iframe` is used and its source URL redirects to a different URL, both the `iframe` and `Redirect` options apply.

D.3 Revision Intention

Table 5 defines the classification categories used to describe the intent of revisions during our analysis. Crucially, a single revision may be assigned different intentions depending on which bug is being analyzed. For example, if a revision fixes *Bug A* but simultaneously introduces *Bug B*, its intent relative to *Bug A* is classified as `Fix affected SOP bug`, whereas for *Bug B*, it acts as `Fix other SOP bug`.

We strictly define "security-related" intents as those where the revision explicitly aims to harden the browser's security posture. Consequently, revisions that relax overly restrictive security policies are classified as `Fix non-security bug`, as their primary goal is functional correctness rather than security enforcement. Furthermore, when classifying revisions that revert commits or implement specifications, we focus on the developer's underlying motivation. For instance, if a feature is reverted due to performance issues, or a specification is modified to resolve inconsistencies, the intent is classified as `Fix bug`. Conversely, if a specification update is implemented to support new capabilities, the revision is classified as `Enable feature`.

E Cross-Browser Bug Lifecycles

Figure 9 presents an overview of the identified cross-browser SOP vulnerabilities. The x-axis displays the time in years, while the y-axis displays the unique bug report IDs, color-coded according to the browser vendor. The green markers indicate the specific date each vulnerability was reported.

Table 3: Classification of SOP bugs.

Bug category	Description
Arbitrary resource content	Leaking the content of cross-origin resources of all types, generally making them readable through JavaScript.
Audio content	Leaking the content of cross-origin audio files.
Context isolation	Executing JavaScript in a cross-origin browsing context, such as reading the context's DOM or accessing its APIs.
CSP content	Leaking the content of a CSP of a cross-origin document.
CSS content	Leaking the content of a cross-origin CSS file.
Downloading resource	Downloading the content of a cross-origin resource to the user's device in a situation where downloading only a same-origin resource should be permitted.
Embedding a resource	Embedding and applying a cross-origin resource in a situation where only a same-origin resource should be permitted, such as a font or subtitles without CORS enabled.
Image content	Leaking the content of a cross-origin image, as well as leaking only limited characteristics, such as the used colors or alpha channel value.
Incorrectly attached request cookies	Attaching a cookie to a cross-origin request in a context when it should not happen.
Incorrectly attached request headers	Attaching a header field other than a cookie or its incorrect value to a cross-origin request in a context when it should not happen.
Local file content	Leaking the content of a resource stored in the local filesystem.
Preflight bypass	Performing a complex cross-origin request when the cross-origin server did not enable the CORS protocol.
Redirect URL	Leaking the full or partial URL where a cross-origin browsing context redirected; if the only information leaked is whether a redirect was performed or not, the Resource response status category should be chosen instead.
Resource response status	Leaking the exact HTTP status of a cross-origin resource, as well as leaking partial status information, such as whether the status is in the 200–299 range or not.
Resource size	Leaking the exact or approximate size of a cross-origin resource.
SRI hash	Leaking the value of the SRI hash of a cross-origin resource.
Subtitles content	Leaking the content of a cross-origin subtitles file.
Text-like resource content	Leaking the content of a cross-origin text-like resource, such as a text or CSV file.
User interaction	Leaking user's interaction with a cross-origin browsing context, such as whether or where a click was performed.
Video content	Leaking the content of a cross-origin video file.

Table 4: Classification of exploitation mechanisms.

Exploit mechanism	Description
SOP logic issue	An underlying problem of the SOP implementation, generally manifesting by the existence of multiple mechanisms to exploit the same vulnerability.
`<audio>` tag	Using the `<audio>` tag.
Background Fetch	Using the Background Fetch API.
canvas	Using the `<canvas>` tag.
Error handler	Using an error handler or error message, generally through the `onerror` attribute.
fetch	Using the `fetch` method of the `Window` interface.
iframe	Using an `<iframe>` element.
Local file	Using the URL from the local file system, either by accessing it directly or by referencing it in the code.
`navigator.sendBeacon`	Using the `sendBeacon` method of the `Navigator` interface.
Performance API	Using methods of the `Performance` interface, generally by inspecting its entries.
Redirect	Using a URL redirecting to a different URL, generally redirecting from the same origin to a different origin.
Service Worker	Using the Service Worker API, generally to intercept requests, except for using the Background Fetch API.
Storage API	Using the Storage API, generally by calling the `estimate` method on the `navigator.storage` property.
Violation report	Using a violation report listener, such as for the CSP or the `X-XSS-Protection` header.
`window.open`	Using the `open` method of the `Window` interface.
`XMLHttpRequest`	Using the `XMLHttpRequest` API.

Table 5: Classification of revision intentions.

Revision intention category	Description
Fix affected SOP bug	Fixes the specific bug under analysis.
Fix other SOP bug	Fixes a different bug that affects the SOP.
Fix unrelated security bug	Fixes a bug in a security policy other than the SOP.
Fix non-security bug	Fixes a bug unrelated to security or aims to loosen a security policy.
Design revision of SOP	Refactoring or performance improvements to the SOP code base, largely unnoticeable to the user.
Design revision of other security policy	Design revision of a security policy other than the SOP.
Non-security design revision	Design revision unrelated to security.
Enable SOP feature	Enables a new SOP feature (e.g., CORS, preflight requests).
Enable unrelated security feature	Enables a new feature in a non-SOP security policy (e.g., enabling CSP).
Enable non-security feature	Enables a new feature unrelated to security (e.g., Service Worker API, new API methods).
Update SOP feature	Enables a relatively small change in the SOP, such as adding a new configuration option or new attribute.
Update unrelated security feature	Enables a relatively small change in a security policy other than the SOP.
Update non-security feature	Enables a relatively small change unrelated to the SOP.
Disable SOP feature	Removes a feature of the SOP due to being deprecated or the change of a launch plan.
Disable unrelated security feature	Removes a feature in a security policy other than the SOP.
Disable non-security feature	Removes a feature unrelated to security.

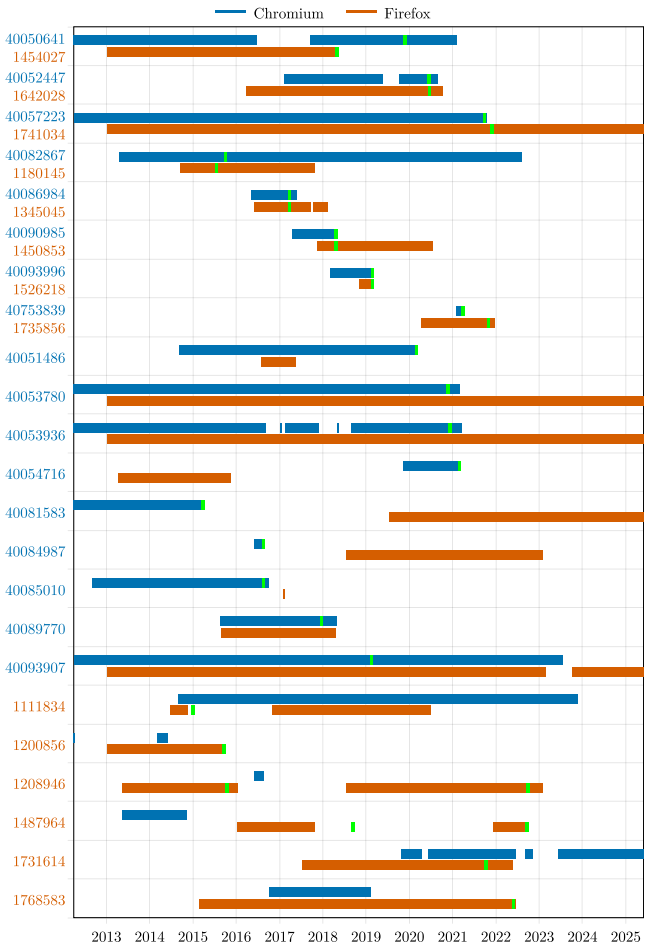


Figure 9: Developer intentions of revisions that fixed a SOP bug.

References

[1] Adam Barth. 2011. HTTP State Management Mechanism. https://datatracker.ietf.org/doc/html/rfc6265

[2] Adam Barth. 2011. The Web Origin Concept. https://datatracker.ietf.org/doc/html/rfc6454

[3] Adam Barth, Collin Jackson, and John C Mitchell. 2008. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*. 75–88.

[4] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. 2024. Web Platform Threats: Automated Detection of Web Security Issues With WPT. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 757–774. https://www.usenix.org/conference/usenixsecurity24/presentation/bernardo

[5] Frederik Braun. 2012. *Origin policy enforcement in modern browsers*. Ph. D. Dissertation. Ruhr-Universität Bochum.

[6] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. 2018. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1079–1093. https://www.usenix.org/conference/usenixsecurity18/presentation/chen-jianjun

[7] Pinji Chen, Jianjun Chen, Mingming Zhang, Qi Wang, Yiming Zhang, Mingwei Xu, and Haixin Duan. 2025. Cross-Origin Web Attacks via HTTP/2 Server Push and Signed HTTP Exchange. *Proceedings 2025 Network and Distributed System Security Symposium* (2025). https://api.semanticscholar.org/CorpusID:276862729

[8] Chrome DevTools Protocol. [n. d.]. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/

[9] Chromium contributors. 2013. Cross-Origin copy & paste / drag & drop allowing XSS (again, this time srcdoc). https://issues.chromium.org/issues/40076853

[10] Chromium contributors. 2014. Security: file-to-file SOP bypass on Linux via /proc/self/fd/. https://issues.chromium.org/issues/40080769

[11] Chromium contributors. 2014. Security: __lookupGetter__ and __lookupSetter__ can be used to leak all cross-origin data. https://issues.chromium.org/issues/40080211

[12] Chromium contributors. 2015. Insufficient fix for Cross-Origin (Partial) Status Code leak (XS-Leak). https://issues.chromium.org/issues/40062152

[13] Chromium contributors. 2015. Origin header preserved for cross-origin redirects with 307 status code, should be null. https://issues.chromium.org/issues/40081583

[14] Chromium contributors. 2015. [Resource Timing] should not include no-cors cross-origin stylesheet subresources. https://issues.chromium.org/issues/40082867

[15] Chromium contributors. 2016. Security: bypassing CORS by returning 308 for revalidating request for Resource previously without redirects from MemoryCache. https://issues.chromium.org/issues/40084396

[16] Chromium contributors. 2017. Cross-origin Shared Worker. https://issues.chromium.org/issues/40089653

[17] Chromium contributors. 2018. Cross-origin download bypasses SameSite cookie. https://issues.chromium.org/issues/40091708

[18] Chromium contributors. 2019. Security: CORB allows for cross-origin leaks on social networks. https://issues.chromium.org/issues/40093907

[19] Chromium contributors. 2020. navigator.sendBeacon doesn't make CORS preflight request. https://issues.chromium.org/issues/40051484

[20] Chromium contributors. 2020. Security: sendBeacon allows sending arbitrary POST requests with application/octet-stream content type without CORS. https://issues.chromium.org/issues/40051486

[21] Chromium contributors. 2021. Security: Cross-Origin Response Size Leak Via BackgroundFetch. https://issues.chromium.org/issues/40057097

[22] Chromium contributors. 2024. Cross-origin leak: GC activity and navigation info using the MessagePort.close event. https://issues.chromium.org/issues/323695987

[23] Chromium contributors. 2025. MediaError message property leaks cross-origin same-site response status. https://issues.chromium.org/issues/416685988.

[24] Chromium Docs. [n. d.]. Security Fields, Hotlists, and Issue Access / Visibility. https://chromium.googlesource.com/chromium/src/+/lkgr/docs/security/security-labels.md#drop-visibility-group-restrictions-from-old_fixed-bugs-for-disclosure

[25] DistriNet. [n. d.]. BugHog. https://github.com/DistriNet/BugHog.

[26] Firefox Contributors. [n. d.]. Fixing Security Bugs - Firefox Source Docs. https://firefox-source-docs.mozilla.org/bug-mgmt/processes/fixing-security-bugs.html

[27] Firefox contributors. 2015. Resource timing violate SOP for "no-cors" CSS. https://bugzilla.mozilla.org/show_bug.cgi?id=1180145

[28] Firefox contributors. 2021. Guessing the URL a cross-origin iframe was redirected to by listening and counting the number of load events. https://bugzilla.mozilla.org/show_bug.cgi?id=1741034

[29] Firefox contributors. 2021. MediaError message property leaks information on cross-origin same-site pages. https://bugzilla.mozilla.org/show_bug.cgi?id=1731614

[30] Firefox contributors. 2023. Top level Cross Origin Domain Redirection Bypass. https://bugzilla.mozilla.org/show_bug.cgi?id=1814879

[31] Firefox Contributors. 2025. https://bugzilla.mozilla.org/show_bug.cgi?id=1965430

[32] Firefox Contributors. 2025. Script element load vs error event leaks cross-origin resource status (ORB crbug/40093907). https://bugzilla.mozilla.org/show_bug.cgi?id=1965628

[33] David Flanagan. 1997. *JavaScript: The Definitive Guide* (2nd ed.). O'Reilly Media. 317 pages. https://archive.org/details/javascriptdefin000flan

[34] Gertjan Franken, Tom Van Goethem, Lieven Desmet, and Wouter Joosen. 2023. A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3673–3690. https://www.usenix.org/conference/usenixsecurity23/presentation/franken

[35] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. 2018. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. https://www.usenix.org/conference/usenixsecurity18/presentation/franken. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 151–168.

[36] Scott Helme. 2021. COEP COOP CORP CORS CORB — CRAP that's a lot of new stuff! https://scotthelme.co.uk/coop-and-coep/

[37] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. 2015. BrowserAudit: automated testing of browser security features. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 37–47. doi:10.1145/2771783.2771789

[38] Maksim Sadym James Graham, Alex Rudenko. 2026. WebDriver BiDi. https://www.w3.org/TR/webdriver-bidi/

[39] Martin Johns, Sebastian Lekies, and Ben Stock. 2013. Eradicating DNS Rebinding with the Extended Same-origin Policy. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 621–636. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/johns

[40] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner. 2007. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 58–71. doi:10.1145/1315245.1315254

[41] Eiji Kitamura. 2020. Making your website "cross-origin isolated" using COOP and COEP. https://web.dev/articles/coop-coep

[42] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. 2021. XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1771–1788. doi:10.1145/3460120.3484739

[43] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[44] Jie Lin and David Mohaisen. 2025. From Large to Mammoth: A Comparative Evaluation of Large Language Models in Zero-Shot Vulnerability Detection. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2025.241491

[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[46] Peiyu Liu, Junming Liu, Lirong Fu, Kangjie Lu, Yifan Xia, Xuhong Zhang, Wenzhi Chen, Haiqin Weng, Shouling Ji, and Wenhai Wang. 2024. Exploring ChatGPT's Capabilities on Vulnerability Management. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 811–828. https://www.usenix.org/conference/usenixsecurity24/presentation/liu-peiyu

[47] Antonio Sartori Mike West. 2025. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/

[48] Mozilla Developer Network. [n. d.]. Cross-site request forgery (CSRF). https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/CSRF

[49] Mozilla Developer Network. 2025. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/Same-origin_policy

[50] Mozilla Developer Network. 2026. User activation. https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/User_activation

[51] Netscape. 2002. Netscape 3.0 Handbook. https://web.archive.org/web/20020808153106/http://wp.netscape.com:80/eng/mozilla/3.0/handbook/javascript/advtopic.htm#1009533

[52] Mozilla Developer Network. 2025. Firefox 20 release notes for developers. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/20

[53] Dominik Trevor Noß, Lukas Knittel, Christian Mainka, Marcus Niemietz, and Jörg Schwenk. 2023. Finding All Cross-Site Needles in the DOM Stack: A Comprehensive Methodology for the Automatic XS-Leak Detection in Web Browsers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2456–2470. doi:10.1145/3576915.3616598

[54] Jannis Rautenstrauch, Trung Tin Nguyen, Karthik Ramakrishnan, and Ben Stock. 2025. Head(er)s Up! Detecting Security Header Inconsistencies in Browsers. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 3057–3070. doi:10.1145/3719027.3765119

[55] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. 2023. The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2744–2760. doi:10.1109/SP46215.2023.10179311

[56] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z Snow, and Michalis Polychronakis. 2017. Revisiting browser security in the modern era: New data-only attacks and defenses. In *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 366–381.

[57] Hossein Saiedian and Dan Broyle. 2011. Security Vulnerabilities in the Same-Origin Policy: Implications and Alternatives. *Computer* 44, 9 (2011), 29–36. doi:10.1109/MC.2011.226

[58] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-Origin Policy: Evaluation in Modern Browsers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 713–727. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk

[59] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. 2010. On the Incoherencies in Web Browser Access Control Policies. In *2010 IEEE Symposium on Security and Privacy*. 463–478. doi:10.1109/SP.2010.35

[60] Dolière Francis Some, Nataliia Bielova, and Tamara Rezk. 2017. On the Content Security Policy Violations due to the Same-Origin Policy. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 877–886. doi:10.1145/3038912.3052634

[61] Marco Squarcina, Pedro Adão, Lorenzo Veronese, and Matteo Maffei. 2023. Cookie Crumbles: Breaking and Fixing Web Session Integrity. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5539–5556. https://www.usenix.org/conference/usenixsecurity23/presentation/squarcina

[62] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*. 921–930.

[63] Web Platform Tests. [n. d.]. web-platform-tests documentation. https://web-platform-tests.org/.

[64] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 481–496. doi:10.1145/3564625.3567985

[65] The Chromium Projects. [n. d.]. Chromium Development Calendar and Release Info. https://www.chromium.org/developers/calendar/

[66] Tom Van Goethem, Gertjan Franken, Iskander Sanchez-Rola, David Dworken, and Wouter Joosen. 2022. SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 784–798. doi:10.1145/3488932.3517416

[67] Anne van Kesteren. 2022. Opaque Response Blocking (ORB, aka CORB++). https://github.com/annevk/orb

[68] W3C. 2010. Same Origin Policy. https://www.w3.org/Security/wiki/Same_Origin_Policy

[69] W3C. 2014. Cross-Origin Resource Sharing. https://www.w3.org/TR/2020/SPSD-cors-20200602/

[70] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1376–1387. doi:10.1145/2976749.2978363

[71] WHATWG. 2026. Fetch. https://fetch.spec.whatwg.org/

[72] WHATWG. 2026. HTML. https://html.spec.whatwg.org/

[73] Seongil Wi, Trung Tin Nguyen, Jiwhan Kim, Ben Stock, and Sooel Son. 2023. DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing, In NDSS. *NDSS*. https://publications.cispa.saarland/3891/

[74] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and Haoyu Wang. 2025. Large Language Models for Cyber Security: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2025). doi:10.1145/3769676 Just Accepted.

Received 2026-02-05; accepted 2026-03-19