A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs

Anonymous submission

Abstract

The constantly evolving Web exerts a chronic pressure on the development and maintenance of the Content Security Policy (CSP), which stands as one of the primary security policies to mitigate attacks such as cross-site scripting. Indeed, to attain comprehensiveness, the policy must account for virtually every newly introduced browser feature, and every existing browser feature must be scrutinized upon extension of CSP functionality. Unfortunately, this undertaking's complexity has already led to critical implementational shortcomings, resulting in the security subversion of all CSP-employing websites.

In this paper, we present the first systematic analysis of CSP bug lifecycles, shedding new light on bug root causes. As such, we leverage our automated framework, BUGHOG, to evaluate the reproducibility of publicly disclosed bug proofs of concept in over $100,000$ browser revisions. By considering the entire source code revision history since the introduction of CSP for Chromium and Firefox, we identified 123 unique introducing and fixing revisions for 75 CSP bugs. Our analysis shows that inconsistent handling of bugs led to the early public disclosure of three, and that the lifetime of several others could have been considerably decreased through adequate bug sharing between vendors. Finally, we propose solutions to improve current bug prevention and response practices.

1 Introduction

Since their inception, web browsers have grown to become immense applications comprising tens of millions of code lines, introducing new features with virtually every major release. To keep up this pace, over 100 code revisions are applied to their code base every single day [55], ranging from bug fixes to new feature introductions. Although this pushes the Web forward in many great ways, meanwhile, browser vendors need to make a continuous effort to guard against newly discovered attacks and bypasses for both established and new security policies.

Unfortunately, numerous CVE reports and an extensive body of research have previously exposed countless vulnerabilities facilitated by flawed browser security policy implementations. More specifically, various shortcomings of essential security policies such as the Content Security Policy (CSP) [35], Same-Origin Policy [56], SameSite cookie policy [32] and access control policies [58] have been discovered and exhibited. In several cases this makes the security policy, which websites often rely on to safeguard their users, obsolete until a mitigation is in place. Furthermore, vulnerabilities are often caused by inconsistent implementations among browsers as well [14, 39, 57, 69]. However, the granularity of these studies halts at the level of browser release versions, disregarding information related to the individual revisions that cause or fix a bug.

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

Our analysis shows that half of the CSP bugs were already present at the time of the policy's introduction, among which a severe Chromium vulnerability with subsequent bug bounty of \$5000 (CVE-2021-30531) [18]. After undermining the effectiveness of CSP for a period of more than eight years, the issue was ultimately fixed in 2021, highlighting how even severe bugs can stay under the radar for extensive periods of time. Besides these so-called foundational bugs, a large part was introduced by revisions intended to fix other CSP bugs or redesigns of the underlying code structures, demonstrating the fragile nature of CSP related source code.

In our analysis we employed a dynamic evaluation, in contrast to static evaluation based on bug reporting information in prior work [4, 9, 15, 22, 30]. This allowed us to to perform a cross-browser evaluation of all reported bugs, reproducing bugs reported for one browser throughout the revision history of the other. This way we found 14 shared bugs, among which seven could be completely avoided or reduced in lifetime if bugs were more effectively shared between vendors. Furthermore, we could reproduce four of the collected bugs in Safari's most recent version. Additionally, we identify several other bug handling flaws such as the inconsistent use of bug report labels. More severely, our evaluation detected three bugs that were labeled as fixed and eventually publicly disclosed while the fixing revision was not effective, leaving the browser vulnerable unbeknownst to the developers. Two of these bugs remained public and unfixed for at least a year, and one was only fixed after we reported the matter.

We make the following contributions:

- We developed BugHog, a framework for pinpointing introductions and fixes of browser security policy bugs at the level of individual code revisions. This framework will be open-sourced upon publication of this work, and can be extended to facilitate the evaluation of other security policy implementations as well.

- To the best of our knowledge, we performed the first systematic lifecycle analysis based on dynamic evaluations over the full history of a browser security policy. As such, we analyzed 75 reported CSP bugs for Chromium and Firefox, covering 123 unique code revisions that caused an introduction or fix.

- Based on our thorough analysis, we diagnosed several flaws regarding security policy implementations and bug handling practices, causing a needless escalation of security implications.

- Finally, we propose several remedies to these issues, such as more rigorous bug sharing between vendors and more stringent bug handling practices.

2 Background

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

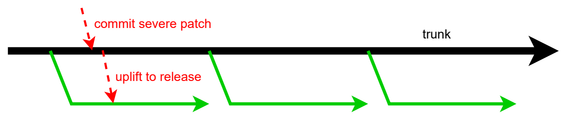

Figure 1: Chromium's development practice where all applied revisions are periodically forked into a release branch.

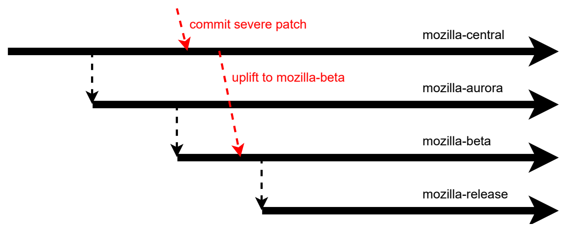

Figure 2: Firefox's development practice where all applied revisions are periodically imported to a more stable repository.

2.1 Web Browser Development

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

2.1.1 Version Control

Web browsers, being code development projects consisting of millions of code lines, are developed, managed and maintained by leveraging a version control system (VCS). Although Chromium and Firefox employ different VCSs (i.e. Git[1] and Mercurial[2], respectively), their underlying functionality is very similar. However, the employed version control strategies of the two projects slightly differ.

Chromium developers apply a trunk-based development pattern to a single repository (Figure 1), where each developer directly commits to a so-called trunk branch (e.g. instead of using feature branches) [20, 21]. Source code is prepared for building the release binary (e.g. disabling experimental features) on a so-called release branch forked at regular intervals from the trunk. Only in special cases like urgent security fixes, a revision on the trunk is cherry-picked and merged onto such a release branch [1].

Firefox manages four separate repositories, each associated with a different release channel (Figure 2). All code revisions are by default applied to the `mozilla-central` repository (i.e. nightly), where all revisions are periodically imported into `mozilla-aurora`. This process is repeated for each repository, such that all revisions will be consecutively imported to the `mozilla-beta` and eventually `mozilla-release` repositories. In particular cases, developers might decide to uplift a feature or a patch to mitigate a severe vulnerability on a more stable channel [41, 42].

[1]https://git-scm.com
[2]https://www.mercurial-scm.org/

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

2.1.2 Regression Testing

In addition to their own specific test suites, Chromium [24] and Firefox [26] share a common cross-browser test suite called *Web Platform Tests* (WPT) since September 2014 for identifying potential regressions (i.e. previously fixed issues that have been inadvertently reintroduced) [11, 17, 64]. Both browser vendors depend on and contribute to the project, which serves as a comprehensive set of checks to confirm compliance to established web standards, including security prerequisites. According to both vendors' contribution policies, each revision or patch should successfully complete all regression tests before it can be landed [25, 51].

Contributors to the Chromium project are advised to use one of two procedures to track down the introduction or fix of a regression. One option is utilizing their `bisect-builds.py` script which automates a binary search over a bounded revision range of publicly hosted revision binaries, though the user is required to manually check for each binary whether the bug is reproduced [49]. The other recommended practice utilizes `git bisect`, which is able to discern whether a revision reproduces the targeted issue in an automated manner if provided with the appropriate script and test files [36]. Nonetheless, this evaluation requires checking out and building each revision that is to be evaluated, making it resource-intensive.

Firefox provides a tool for bisecting regressions as well; similar to Chromium's script, their so-called `autobisect` tool relies on publicly available revision binaries [44]. However, this bisection tool is fully automated since the script can autonomously distinguish between test case results.

In Section 3.2.5, we discuss the advantages and limitations of each approach, compared to BugHog.

2.2 Content Security Policy

CSP version 1 was originally proposed as an in-depth defense mechanism against content injection attacks such as cross-site scripting (XSS) [46, 60, 61]. Websites can deploy a policy by providing the `Content-Security-Policy` header or `<meta>` tag in their response, and consequently the browser will enforce the defined policy client-side.[3]

CSP allows developers to distinguish between several directives, enabling different rules over different resource types. For instance, the policy defined below will only permit the browser to load external scripts from `third-party.com`, while all other resources can only originate from the current website (indicated by `self`).

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

In this study, we make a distinction between two subclasses of the content control use case: active content control (i.e. script blocking) and non-active content control (i.e. frames).

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

Subsequent versions of CSP, specifically CSP 2 and CSP 3, introduced additional functionality to the policy specification, among which new directives and keywords [67, 68]. For instance, the `nonce` keyword provides developers with the ability to allow inline script inclusion while simultaneously safeguarding against script injections through the use of a secret nonce. Moreover, new use cases have been introduced, such as the `upgrade-insecure-requests` directive, which facilitates the automatic upgrading of all requests made over unencrypted channels (e.g. `HTTP`) to secure channels (e.g. `HTTPS`), thereby enforcing TLS encryption. Another important aspect is framing control, which is facilitated by the `frame-ancestors` directive. This directive allows developers to specify which origins are permitted to embed the website within an `iframe` to prevent clickjacking attacks. Lastly, the `referrer` directive grants developers control over the `Referer` request header when users navigate from the current website. Previous studies have already highlighted a similar differentiation between CSP use cases [54, 65].

The inheritance of a policy between browsing contexts presents significant challenges for CSP designers and implementers. When creating a new browsing context, such as embedding an `iframe` or opening a new window, the enforced policy is inherited from the opener browsing context in specific situations. For example, new browsing contexts with a `blob:` or `data:` URL should inherit the employed policy from the opener context. However, exercising excessive caution in inheritance can create opportunities for CSP subversion, while overly strict inheritance may result in cross-site leaks (XS-Leaks) [38]. In XS-Leaks, malicious actors exploit CSP to extract user state information from cross-site services, leading to potential privacy breaches. For instance, a malicious website could employ a CSP policy that permits navigation to a benign website, but restricts access to the landing page to which logged-in users are redirected. This way, the adversary can infer the presence of an active session on the benign site through CSP's violation report, obtained through the `report-to` CSP directive.

Figure 3 depicts all CSP bugs related to the use cases that were discussed in this section. We refer to Mozilla Developer Network for a comprehensive overview of all CSP directives and keywords [46].

3 Methodology

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

3.1 Bug Collection and Reproduction

All bugs were collected from Chromium's[4] and Firefox's[5] public bug tracking platforms. For both browser projects, bug reports are by default confidential until a fix has been widely deployed [10, 50]. Unfortunately, we did not receive a response from the WebKit Security team regarding our inquiry for access to fixed WebKit bugs.

To ensure comprehensiveness and enable an evaluation of applied bug report labels, we adopted broad search criteria. For instance, we did not rely on any specific CSP labels, but instead used keywords that are matched against the content of bug reports. This approach ensures that any potential oversights in developer labeling do not impact the integrity of our dataset. Subsequently, we filtered out all bugs that were not related to CSP, by manually inspecting the included bug description. In total, we collected 86 bug reports; 58 for Chromium and 28 for Firefox, a ratio which is similar to that of prior studies [30]. In case two or more bug reports described the same bug, we considered it as one bug. As such, we collected 75 unique bugs in total. We refer to Appendix A for more details about the collection and filtering process. Figure 3 shows all classes that were identified in our dataset, along with the number of associated bugs.

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

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

3.2 Automated Lifecycle Identification

BUGHOG is designed to evaluate an extensive range of individual revisions of the Chromium and Firefox browsers, in order to identify a bug's complete lifecycle, from bug introduction until mitigation. In the following sections, we discuss the three main tasks of BUGHOG: selecting the appropriate revisions to evaluate, collecting the selected revision binaries and performing a dynamic evaluation on the collected binaries. Upon publication of this work, we will release BUGHOG as open-source.[6]

[4]https://bugs.chromium.org/

[5]https://bugzilla.mozilla.org/

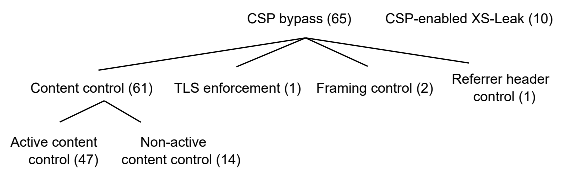

Figure 3: An overview of all bug classes with their respective frequencies in our dataset.

3.2.1 Overview

Figure 4 shows a high-level overview of BUGHOG covering three main components (indicated in bold), each of which runs inside its own Docker container. All Docker images are built on top of the Debian GNU/Linux 10 (buster) base image.

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

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

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

Apart from the three main components, BUGHOG utilizes a MongoDB database for storing the results of each revision evaluation, which can subsequently be queried and visualized. Finally, we use the publicly available revision binaries hosted by each vendor for our dynamic evaluation and scrape online repository information to traverse over revisions.[7][8] We only

[6]https://github.com/USENIX-SUBMISSION-23/bugs-life-framework

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

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

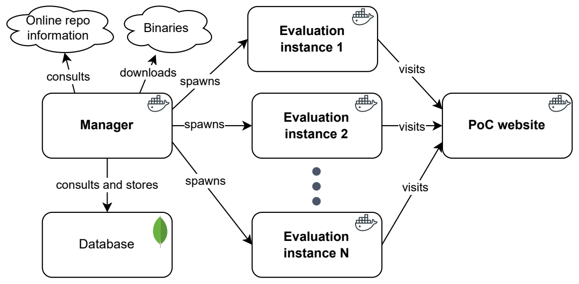

Figure 4: High-level overview of BugHog. The Docker logo indicates that a component is run inside its own Docker container.

require access to browser source code for building binaries when the provided online binaries are not sufficient.

3.2.2 Collecting Revision Binaries

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

Downloading. Besides hosting binaries of release versions, both Chromium[9] and Firefox[10] host additional binaries based on certain source code revisions as well. From these collections, it appears that Chromium builds a binary multiple times per hour, while Firefox seems to build a binary every 12 hours. If a to-be-evaluated revision binary is not available, BugHog will download the one closest available.

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

3.2.3 Revision Evaluation Selection and Order

The scope of this work covers CSP in the form of the currently employed `Content-Security-Policy` header or `<meta>` tag. We did not extend our analysis to experimental precursors such as `X-WebKit-CSP` and `X-Content-Security-Policy` [46], since these were not shipped as finished policy implementations, and as such, are not assumed to be bugless.

CSP 1.0 was introduced by revision 165317 [16] and revision 144546 [31] for Chromium and Firefox respectively. As such, to identify all plausible bug lifecycles we are required to evaluate the revision range between these revisions and the most recent browser version (version 109 for both Chromium and Firefox).

Our search strategy is composed of two phases:

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

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

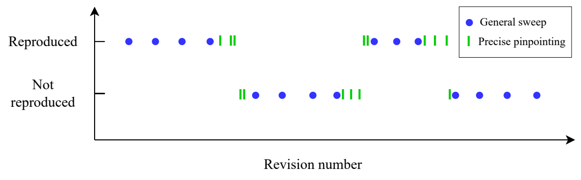

Figure 5: Example of the revision evaluation process.

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

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

Figure 5 depicts an example of evaluation output over a predefined range. Here, the general sweep (blue dots) reveals that the bug is reproducible in the first subrange of revisions, and that although eventually fixed, the bug is reintroduced at a later point. In the second phase, precise pinpointing (green rods) reveals more accurately where the introductions and fixes occurred.

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

3.2.4 Dynamic Evaluation

Although browser automation libraries like Selenium are prevalent, they often do not provide support for outdated, older browser versions. Fortunately, our use of the command

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

Table 1: Overview of bisection frameworks in terms of supported functionality.

line interface (CLI) for instructing browsers gives us the advantage of evaluating any browser binary.

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

3.2.5 Advantages and Limitations

In Section 2.1.2, we discussed various tools available for bisecting bugs through dynamic evaluation. An overview of the supported functionality for each existing bisection tool, including BUGHOG, is presented in Table 1.

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

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

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

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

3.3 Analysis

To conduct our analysis, we used an automated scraper to collect information from bug reports and code revisions obtained

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

Table 2: Overview of all revision intentions, where the Regression column indicates whether the associated introduction revisions would be a regression.

from the aforementioned public bug tracking platforms. Additionally, to enhance the depth of our analysis, we conducted manual inspections of relevant sections of the source code. The visualizations and statistics used in Section 4 were generated by automated scripts, which can be re-used for other bug studies.

To better understand the purpose of code changes, each revision was manually annotated with a label indicating its intended purpose. All labels are listed in Table 2, where the last column indicates whether a label is considered a regression if its intent is linked to a bug introducing revision. The labeling was done by two experts and evaluated with a Cohen's Kappa agreement score of 0.81, with any remaining disagreements resolved through discussion. We refer to Appendix B of a detailed description of each label, and further details on the labeling methodology.

4 Results

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

4.1 Bug Lifecycle

To shed light on the duration that CSP bugs stay undiscovered, we calculated the cumulative distribution function (CDF) of the time between the introduction and reporting of the collected CSP bugs for each browser (Figure 6). Interestingly, Chromium bugs seem to live longer before they are reported, compared to Firefox. For Chromium the median time between the introduction and report is 2.9 years, whereas this is 1.2 years for Firefox. Note that Chromium has enabled CSP support since November 2012, whereas Firefox only enabled it

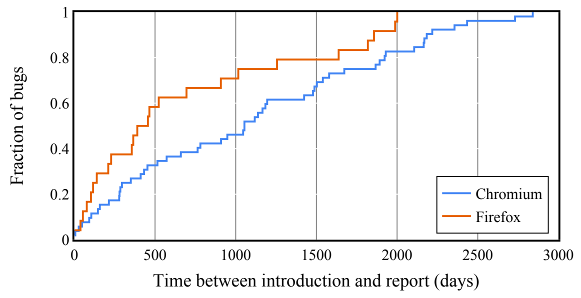

Figure 6: CDF of time between introduction and report of each bug.

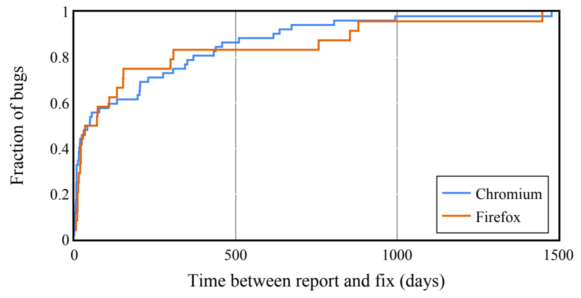

Figure 7: CDF of time between report and fix of each bug.

since May 2013.

Figure 7 shows the CDF of the duration between the first report and its subsequent effective fix.[11] For Chromium the median duration is 44 days, which is slightly lower than the 52 days it takes Firefox to land a fix.

When we look at the number of bug reports and fixing revisions for *foundational* bugs, i.e. bugs that were present since the introduction of CSP, over the ten years since CSP was introduced, we observe a downward trend (Figure 8a). Still, it should be noted that part of this downward trend in the last three years can be attributed to the fact that not all discovered bugs are publicly disclosed yet.

However, inspecting the same bar chart for *non-foundational* bugs, i.e. bugs introduced after the introduction of CSP, a far less apparent downward trend can be found (Figure 8b). Moreover, the last three years show a consistently higher number of both reports and fixing revisions in comparison to our dataset of foundational bugs. Indeed, when inspecting the number of introducing revisions for non-foundational bugs (Figure 9), it strengthens the conclusion that the number of non-foundational bugs does not necessarily decrease. Note that we did not find any introducing revisions for year eight and nine, since these reports are most likely not made public yet.

These findings suggest that CSP as a policy has not yet reached maturity; there is no indication of a decrease in the amount of new bug introductions. Upon closer examination, the major root causes for the introductions of non-foundational bugs appear to be the fixing of (security) bugs (27.2%) and adding new (security) functionality (48%). Interestingly, the fixing of CSP security bugs in particular caused 21% of the non-foundational bugs.

Presumably this conveys that web browser development follows trends similar to those of more general software development. As suggested by previous work, software matures regarding foundational bugs over time [47], while this is not necessarily the case regarding non-foundational bugs [5, 53].

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

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

The lifecycles of shared bugs are shown in the Gantt chart of Figure 10, where the presence of a bug identifier on the y-axis indicates whether a bug report was found for the associated browser. The vertical lines indicate at what time the associated bug report was filed, if present. In eight cases, a specific bug could be reproduced in both browsers whereas there was only a report made to a single browser vendor. In these cases, even the revision in which the bug was fixed did not refer to a report describing the bug, so presumably no issue was ever filed and the bug was introduced and fixed unbeknownst to the developers. We also find that in seven cases the bug was reported for one browser during or before the vulnerable period of the other browser. Although both browsers share the WPT as a common test suite, this result demonstrates a remaining lack of effective threat vector sharing between the two vendors.

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

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

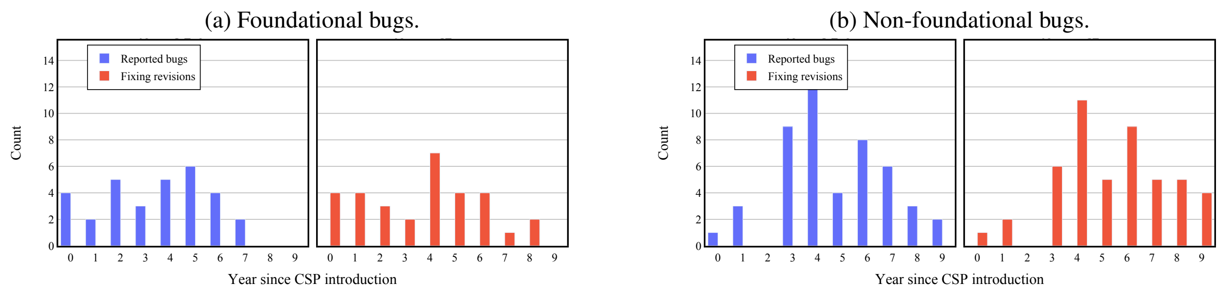

Figure 8: Number of bugs and associated fixing revisions for each year since the introduction of CSP.

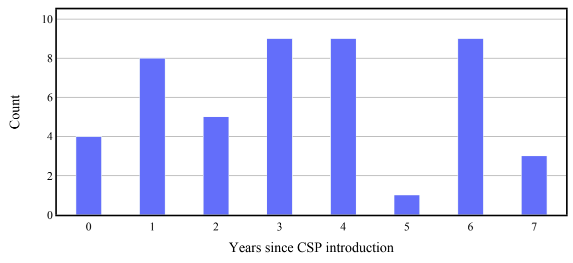

Figure 9: Number of non-foundational bug introductions for each year since the introduction of CSP.

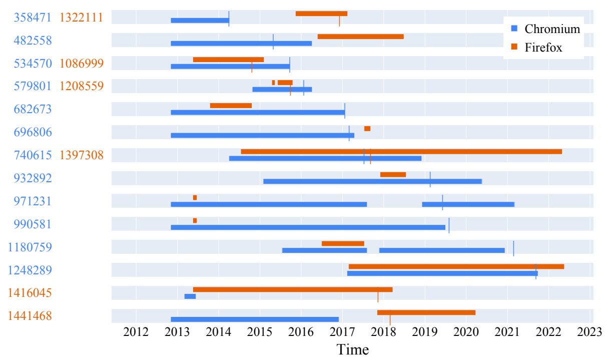

Figure 10: Gantt chart of the bugs that affected both Chromium and Firefox.

sibly disclosed, of which three have been fixed and one is not considered a bug by Safari developers. Although Chromium developers consider the latter issue to be of medium severity, it remains unresolved in their codebase as well.

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

We observe that of all shared bugs depicted in Figure 10, eight are foundational in Chromium, in contrast to only four in Firefox. This indicates that Firefox's introduction of CSP was more comprehensive and sound, however, five of Chromium's foundational bugs eventually appeared as regressions in Firefox. This highlights that even if foundational bugs are avoided through a comprehensive policy introduction, these particular bugs are still prone to being introduced as a future regression. Moreover, this underscores the importance of including *all* shared security tests from all other browser vendors, even if initially not affected by a bug.

4.2 Bug Introduction

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

Figure 11 shows the intention prevalence for revisions that introduced a CSP bug. All revisions introduced a bug a total of 97 times, as some revisions introduced multiple bugs and several bugs regressed after a fix. For both Chromium and Firefox most bugs (23 and 13 respectively) were introduced with the shipping of CSP, indicating that the implementation of CSP at the time was not sufficiently comprehensive. Similarly, enabling a new CSP feature (e.g. shipping a new directive) allowed for various bypasses as well (eight for Chromium, five for Firefox), reinforcing the idea that the introduction or extension of the policy is prone to lack of comprehensiveness.

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

4.2.1 Revision Intent

Of all non-foundational revisions, seven revisions introduced multiple bugs; six revisions introduced two bugs and one even introduced six. For all but one, modifications to CSP

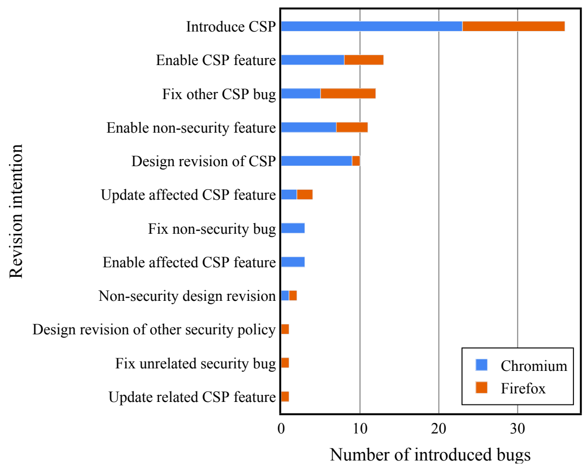

Figure 11: Intentions of revisions that introduced a CSP bug.

configuration logic (i.e. policy parsing, feature or directive introduction) were identified as the bug root cause, indicating that updating CSP configuration logic bears more risk to introducing multiple bugs in comparison to modifying enforcement logic. The revision causing six bugs was applied in an attempt to mitigate several bugs through a design revision of CSP. Here, an inadequate CSP inheritance upon navigating to a new context was reported, which could be abused by an attacker to inject a script.[12] Although the revision effectively mitigated the reported bugs, it introduced new bugs where an inadequate inheritance of CSP was again the root cause.

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

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

4.2.2 Revision Context

For every bug, we analyzed which CSP directive is bypassed (Figure 12a) and what web mechanism or feature facilitated the bypass (Figure 12b). Since we cannot assume that PoCs list all bypassed directives, we reproduced multiple versions of each unique bug to find out whether a single specific `src` directive or multiple `src` directives were affected. Interestingly, 24 bugs (32%) bypass only the `script-src` directive, and 23 (31%) bypass more than one `src` directive. Moreover, CSP's essential and critical responsibility of blocking inline scripts and `eval` were bypassed in five and four instances respectively. We argue that `script-src`'s complexity, given the various keywords that it supports (e.g. `nonce`, `strict-dynamic`, `sandbox`), contributes to its error-prone implementation.

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

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

4.2.3 Source Code

After examining the source code in detail, we have noticed that the enforcement of CSP for content control (Figure 3) is less centralized compared to other use cases. In this case, specific sections of code dedicated to different mechanisms are responsible for performing CSP checks. For example, mechanisms such as `<base>`, `<a>`'s `ping` attribute, favicon fetching, form submission, and Workers require conditional CSP checks in addition to the general resource fetching check. Conversely, the functionality scope for TLS enforcement, framing control, and referrer handling is much narrower, resulting in fewer detached CSP checks. For instance, we only found separate checks for form submission and WebSockets to ensure CSP compliance in both browsers regarding TLS enforcement.

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

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

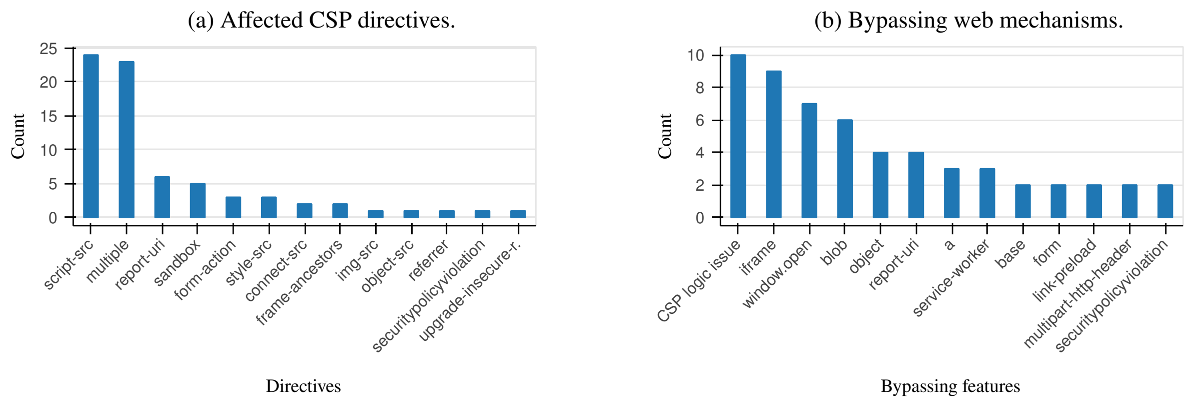

Figure 12: Distribution of affected CSP directives and most prevalent bypassing web mechanisms.

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

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

Remarkably, nearly all inheritance-related issues affected Chromium (20 out of 22), while only four affected Firefox.[13] This disparity implies that Firefox's inheritance logic has been considerably more robust compared to that of Chromium. However, Chromium developers undertook a considerable effort centralizing inheritance logic by incorporating CSP in the Policy Container [19], resolving seven inheritance-related issues simultaneously. As far as we can tell from our dataset, no new inheritance issues have been introduced since. Presumably, Firefox's inheritance logic was already more centralized at an earlier stage [43].

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

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

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

4.3 Bug Reporting

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

Bug tracking platforms are often integrated with their respective source code repositories; for instance, when a bug ID is mentioned in a revision message, this revision will be automatically linked within the bug report as well. Correspondingly, this aids developers in keeping track of all revisions relevant to a certain report. To this end, we investigated how thoroughly associated revisions are linked to a bug report, regarding introducing and fixing revisions. If more than one

[13]Two issues affected both browsers.

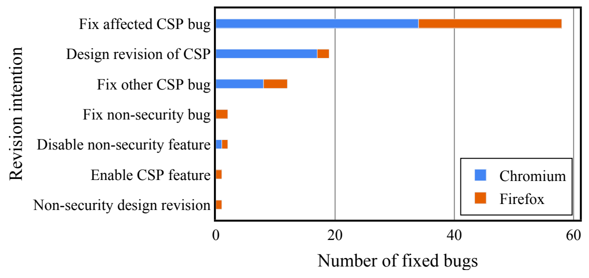

Figure 13: Intentions of revisions that fixed a CSP bug.

bug report is associated with a particular bug (e.g. duplicate reports), we consider a revision linked if it is mentioned by at least one report. We found very similar results for Chromium and Firefox; bug introductions were mentioned for only 4% and 7% of bugs, while fixing revisions were mentioned in 87% and 86% of the cases, respectively.

Since metadata provided in reports lies at the base of understanding the described bug for both developers and researchers, we argue that more effort should be directed at providing consistent and comprehensive information. Furthermore, prior work is known to rely on similar metadata, and as such, this could provide for more accurate evidence [6, 9, 70, 71, 73]. We believe that our framework could be a first step in this process to automatically identify the bug-introducing revision.

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

4.4 Bug Fixing

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

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

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

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

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

Firefox bug XXXXXXX. [14] The regression tests were run on non-packaged builds, on which the applied fix was successful. However, the issue still persisted in packaged builds, unbeknowst to the developers.

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

5 Discussion

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

5.1 CSP Implementation Flaws

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

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

[14]Redacted for anonymous submission

new browser features and CSP functionalities that lack robustness upon their initial implementation, as well as unintended side effects resulting from CSP bug fixes.

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

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

5.2 Improving Bug Handling

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

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

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

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

5.3 Future Work

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

Solutions for CSP soundness specifically could lie in the field of formalization, where CSP would be consolidated as a formal definition. Several aspects of the Web have already been explored in a formalized context, demonstrating its effectiveness by discovery of previously unknown bugs [3, 7, 28, 29, 37]. While this research direction would facilitate the sound introduction of new CSP functionality, formalizing the Web as a whole poses numerous significant challenges due to its dynamic nature and the complex interplay of its mechanisms and policies.

Another approach could leverage dynamic evaluation, similar to our methodology. However, the difficulty here lies in achieving true exhaustiveness, considering all combinations between supported mechanisms, policies and nested browsing contexts. While valuable efforts have been made to explore this approach [32, 35, 69], only limited comprehensiveness has been demonstrated.

6 Related Work

6.1 Dynamic Browser Policy Evaluation

As one of the first, Aggarwal et al. employ fuzzing to detect inconsistencies and flaws in private browsing mode, also demonstrating the potential negative impact of extensions and plugins [2]. Research by Schwenk et al. found inconsistencies among browsers for the Same-Origin Policy, which could lead to vulnerabilities [56]. In that same light, browser access control incoherencies were exposed by Singh et al., leveraging their automated evaluation framework WebAnalyzer [58].

Hothersall-Thomas et al. introduced BrowserAudit, a web application to validate multiple browser security policies [35].

Third-party cookie policies, SameSite cookie policies and various anti-tracking measurements implemented by both browsers and browser extensions were deemed inadequate by Franken et al., employing their framework for dynamic evaluation through browser automation [32]. Luo et al. found that mobile browsers are susceptible to UI attacks due to insufficient protection and even demonstrate a declining trend in security over time [40]. Luo et al. employed dynamic testing to construct a longitudinal overview of supported browser security policies in mobile browsers, uncovering that several widely-used browsers lack support for crucial policies, even several years after their introduction [39]. In recent work, Rautenstrauch et al. uncovered several new vulnerabilities through the first systematic analysis of XS-Leaks [52].

Finally, various frameworks have been developed to dynamically evaluate the security of JavaScript engines and web engines in different contexts as well [23, 33, 48].

6.2 Vulnerability Studies

By examining the rate at which vulnerabilities are reported, Rescorla was the first investigating whether software matures in terms of security [53]. Unfortunately, no conclusive evidence was found for this hypothesis, confirmed by later studies as well [4, 5, 47]. However, Ozment et al. presented statistically significant evidence that the rate of foundational vulnerability reports does decrease over time [47]. Indeed, complementing this research, Edwards et al. demonstrated that the adding of large amounts of new code can decrease software quality and Alexopoulos et al. highlight the need for maintaining stable branches longer in order to detect maturing behavior [5, 27]. Furthermore, Alexopoulos et al. suggest that more expressive security metrics can greatly help us understand the vulnerability lifecycles.

Regarding browser development, Braz et al. uncover several root causes of regression vulnerabilities such as the complexity of browser interactions required for certain regression tests [9]. Zaman et al. and Munauah et al. underline the considerable differences between non-security and security bugs, and consequently motivate the need for this distinction in research [45, 72]. Research of di Biase et al. demonstrated the importance of code review and argues that more security issues are found in case more than two reviewers are involved, as opposed to the two-reviewer policy of Chromium at the time [22]. In addition, further research indicates that security checklists do not significantly improve vulnerability detection and the relative order in which files are reviewed affects the probability for finding security issues [8, 34].

6.3 Content Security Policy

CSP has been the subject of various research projects, both with the focus on validating CSP implementations and on measuring CSP employment in the wild. To begin with, the aforementioned studies of Hothersall-Thomas et al. and Luo et al. investigated the CSP implementations of desktop and mobile browsers, respectively leveraging their automated frameworks [35, 39]. Van Acker et al. demonstrated how attackers could bypass strict CSP enforcement by abusing DNS and resource prefetching in major browsers [63]. Other bypasses were pointed out by Somè et al., where incompatibility issues between the Same-Origin Policy and CSP would allow attackers to execute otherwise blocked scripts [59].

The first study to identify and set out the challenges of CSP adoption was conducted by Weissbacher et al. [66]. Based on their longitudinal study, they uncover various reasons behind the slow adoption rate and ineffective deployments of CSP, proposing potential remedies as well [65]. Calzavara et al. identified various issues with CSP deployment in the wild, such as liberal src expressions, use of inline scripts and underutilization of CSP's monitoring facilities [12, 13]. Roth et al. brought to light the hurdles developers are facing when implementing a comprehensive policy [54]. Calzavara et al. exposed how inconsistencies among the enforced CSP policies in browsers can lead to various gaps in clickjacking defenses of websites [14]. More recently, Stolz et al. showed that the use of the `unsafe-hashes` directive does not necessarily lead to more secure event handles, and argue that although the introduction of the directive is a step in the right direction, web developers should be advised to avoid inline scripts [62]. Finally, Wi et al. uncovered 29 new CSP bypasses that lead to unauthorized script execution, by leveraging the first differential testing framework based on inconsistencies between browser implementations.

7 Conclusion

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

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

References

[1] Aaron Boodman. How Chromium Works. https://aboodman.medium.com/in-march-2011-i-drafted-an-article-explaining-how-the-team-responsible-for-google-chrome-ships-c479ba623a1b.

[2] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 6, USA, 2010. USENIX Association.

[3] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 290–304, 2010.

[4] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. How long do vulnerabilities live in the code? a Large-Scale empirical measurement study on FOSS vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 359–376, Boston, MA, August 2022. USENIX Association.

[5] Nikolaos Alexopoulos, Sheikh Mahbub Habib, Steffen Schulz, and Max Mühlhäuser. The tip of the iceberg: On the merits of finding security bugs. *ACM Trans. Priv. Secur.*, 24(1), sep 2020.

[6] Muhammad Asaduzzaman, Michael C. Bullock, Chanchal K. Roy, and Kevin A. Schneider. Bug introducing changes: A case study with android. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 116–119, 2012.

[7] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[8] L. Braz, C. Aeberhard, G. Calikli, and A. Bacchelli. Less is more: Supporting developers in vulnerability detection during code review. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1317–1329, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[9] Larissa Braz, Enrico Fregnan, Vivek Arora, and Alberto Bacchelli. An exploratory study on regression vulnerabilities. https://arxiv.org/abs/2207.01942, 2022.

[10] Handling Mozilla Security Bugs. Mozilla. https://www.mozilla.org/en-US/about/governance/policies/security-group/bugs/.

[11] Bugzilla. 945222 - web-platform-tests: Create a test runner for web-platform-tests suite. https://bugzilla.mozilla.org/show_bug.cgi?id=945222, dec 2013.

[12] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems? evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1365–1375, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-based analysis of content security policy deployment. *ACM Trans. Web*, 12(2), jan 2018.

[14] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: A formal analysis of inconsistent Click-Jacking protection on the web. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 683–697. USENIX Association, August 2020.

[15] Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 269–279. IEEE Press, 2015.

[16] Chromium. Revision 165317: introduction of csp 1.0. https://chromium.googlesource.com/chromium/src/+/46dd3610caa75097ba521f7f74e5f5c0d7c23b79, nov 2012.

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

[18] Chromium. Issue 1115628: Security: Full csp bypass through blob: Uris. https://bugs.chromium.org/p/chromium/issues/detail?id=1115628, aug 2020.

[19] Chromium. Issue 1149272: Add content security policies to the policy container. https://bugs.chromium.org/p/chromium/issues/detail?id=1149272, nov 2020.

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

[21] Trunk Based Development. Introducion. https://trunkbaseddevelopment.com/.

[22] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. A security perspective on code review: The case of chromium. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30, 2016.

[23] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS*, 2021.

[24] Chromium Docs. Web Tests (formerly known as "Layout Tests" or "LayoutTests"). https://chromium.googlesource.com/chromium/src/+/HEAD/docs/testing/web_tests.md.

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

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

[27] Nigel Edwards and Liqun Chen. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 183–194, New York, NY, USA, 2012. Association for Computing Machinery.

[28] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1204–1215, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202, 2017.

[30] Matthew Finifter, Devdatta Akhawe, and David Wagner. An empirical study of vulnerability rewards programs. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 273–288, Washington, D.C., August 2013. USENIX Association.

[31] Firefox. Revision 144546: introduction of csp 1.0. https://hg.mozilla.org/releases/mozilla-release/rev/6b181afc9fadbd4bb9d04648aa24a34bd9731e82, sep 2013.

[32] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Who left open the cookie jar? a comprehensive evaluation of Third-Party cookie policies. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 151–168, Baltimore, MD, August 2018. USENIX Association.

[33] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Reading between the lines: An extensive evaluation of the security and privacy implications of epub reading systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1730–1747, 2021.

[34] Enrico Fregnan, Larissa Braz, Marco D'Ambros, Gül Çalikli, and Alberto Bacchelli. First come first served: The impact of file position on code review. *arXiv preprint arXiv:2208.04259*, 2022.

[35] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. Browseraudit: Automated testing of browser security features. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 37–47, New York, NY, USA, 2015. Association for Computing Machinery.

[36] Testing in Chromium. Web tests (formerly known as "layout tests" or "layouttests"). https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/testing/web_tests.md#bisecting-regressions.

[37] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 113–128, Bellevue, WA, August 2012. USENIX Association.

[38] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1771–1788, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*, 2019.

[40] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer*

and Communications Security, CCS '17, page 149–162, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Mozilla. Patch uplifting rules. https://wiki.mozilla.org/Release_Management/Uplift_rules.

[42] Mozilla. Release Management/Feature Uplift. https://wiki.mozilla.org/Release_Management/Feature_Uplift.

[43] Mozilla. Understanding web security checks in firefox (part 1). https://blog.mozilla.org/attack-and-defense/2020/06/10/understanding-web-security-checks-in-firefox-part-1/.

[44] MozillaSecurity. autobisect. https://github.com/MozillaSecurity/autobisect.

[45] Nuthan Munaiah, Felivel Camilo, Wesley Wigham, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. *Empirical Softw. Engg.*, 22(3):1305–1347, jun 2017.

[46] Mozilla Developer Network. Content security policy (csp). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[47] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *15th USENIX Security Symposium (USENIX Security 06)*, Vancouver, B.C. Canada, July 2006. USENIX Association.

[48] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642, 2020.

[49] The Chromium Projects. bisect-builds.py. https://www.chromium.org/developers/bisect-builds-py/.

[50] The Chromium Projects. Reporting security bugs. https://www.chromium.org/Home/chromium-security/reporting-security-bugs/.

[51] The Chromium Projects. Testing and infrastructure. https://www.chromium.org/developers/testing/.

[52] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. The leaky web: Automated discovery of cross-site information leaks in browsers and the web. In *44th IEEE Symposium on Security and Privacy*, May 2023.

[53] Eric Rescorla. Is finding security holes a good idea? *IEEE Security and Privacy*, 3(1):14–19, jan 2005.

[54] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*.

[55] Kenneth Russell, Zhenyao Mo, and Brandon Jones. Continuous testing of chrome's webgl implementation. In Patrick Cozzi, editor, *WebGL Insights*, pages 31–46. CRC Press, July 2015. http://www.webglinsights.com/.

[56] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 713–727, Vancouver, BC, 2017. USENIX Association.

[57] Hendrik Siewert, Martin Kretschmer, Marcus Niemietz, and Juraj Somorovsky. On the security of parsing security-relevant http headers in modern browsers. In *2022 IEEE Security and Privacy Workshops (SPW)*, pages 342–352, 2022.

[58] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 463–478, Washington, DC, USA, 2010. IEEE Computer Society.

[59] Dolière Francis Somè, Nataliia Bielova, and Tamara Rezk. On the content security policy violations due to the same-origin policy. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 877–886, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[60] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 921–930, New York, NY, USA, 2010. Association for Computing Machinery.

[61] Brandon Sterne and Adam Barth. Content security policy 1.0. W3c candidate recommendation, W3C, November 2012. https://www.w3.org/TR/2012/CR-CSP-20121115/.

[62] Peter Stolz, Sebastian Roth, and Ben Stock. To hash or not to hash: A security assessment of csp's unsafe-hashes expression. In *2022 IEEE Security and Privacy Workshops (SPW)*, pages 1–12, 2022.

[63] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16,

page 853–864, New York, NY, USA, 2016. Association for Computing Machinery.

[64] web-platform-tests. web-platform-tests documentation. `https://web-platform-tests.org`.

[65] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[66] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is csp failing? trends and challenges in csp adoption. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 212–233, Cham, 2014. Springer International Publishing.

[67] Mike West. Content security policy level 3. W3c working draft, W3C, June 2021. https://www.w3.org/TR/CSP3/.

[68] Mike West, Adam Barth, and Dan Veditz. Content security policy level 2. W3c recommendation, W3C, December 2016. https://www.w3.org/TR/CSP2/.

[69] Seongil Wi, Trung Tin Nguyen, Jiwhan Kim, Ben Stock, and Sooel Son. Diffcsp: Finding browser bugs in content security policy enforcement through differential testing. In *NDSS*, February 2023.

[70] Guanping Xiao, Zheng Zheng, Bo Jiang, and Yulei Sui. An empirical study of regression bug chains in linux. *IEEE Transactions on Reliability*, 69(2):558–570, 2020.

[71] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 26–36, New York, NY, USA, 2011. Association for Computing Machinery.

[72] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: A case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 93–102, New York, NY, USA, 2011. Association for Computing Machinery.

[73] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery.

Appendix

A Bug Report Collecting

In this Appendix, we describe how we collected the relevant CSP bug reports. Section A.1 provides the search criteria used for finding relevant CSP security bug reports. These search criteria are intentionally overly broad as not to miss potentially relevant bug reports. False positives were removed manually.

In light of transparency, we show how many report CSP security bugs we were able to reproduce in Section A.2.

A.1 Search Criteria

The search criteria described in the following sections were utilized to locate vulnerabilities related to CSP or caused by CSP. Additionally, whenever a discovered report was linked to another relevant report that was not originally part of our search results, it was included in our dataset as well. Note that all keywords used in the search criteria will be checked against the whole bug report, including the title, description, and comments.

A.1.1 Chromium

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

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

A.1.2 Firefox

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

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

A.2 Bug Report Distribution

Table 3 shows the exact number of valid or available CSP bug reports, how many of those were reproducible by our framework (in regards to its technical limitations), and finally how many we were able to effectively reproduce. Note that this table only elaborates on the number of bug reports found exclusively through our used search criteria. The total number of reproduced bug reports amounts to 86 when also taking into account reports found through cross-report links.

Valid reports

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

In-scope reports

Not all relevant bugs were reproducible by our framework, due to technical limitations:

- The bug is reported for another OS-specific (other than Linux).[15]
- User interaction is essential for reproducing the bug.
- Console access is required to check exploit success.
- The bug is facilitated by an installed extension.

Reproduced reports

Finally, while all technical requirements where fulfilled, we were not able to effectively reproduce a few edge-case in-scope bugs. We believe that this may be due to inadequate proof-of-concept, unclear bug description or limited understanding of the bug.

B Revision Intention Labels

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

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

Table 3: Number of bug reports

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

B.1 Labeling Process

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

B.2 Label Interpretation

B.2.1 Policy introduction

Introduce CSP CSP is supported starting from this revision.

B.2.2 Fix a Bug

Fix affected CSP bug Intentionally fixes the reported CSP bug.

Fix other CSP bug Intentionally fixes another CSP bug, though it fixed the reported CSP bug as an unintentional side effect.

Fix unrelated security bug Intentionally fixes a non-CSP bug, though it fixed the reported CSP bug as an unintentional side effect.

Fix non-security bug Intentionally fixes a non-security bug, though it fixed the reported CSP bug as an unintentional side effect.

B.2.3 Enable a feature

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

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

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

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

B.2.4 Update a feature

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

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

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

B.2.5 Design choice

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

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

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