A Permissions Odyssey: A Systematic Study of Browser Permissions on Modern Websites

Alberto Fernandez-de-Retana
Researcher
Bilbao, Spain
albertofdr@gmail.com

Jannis Rautenstrauch
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
jannis.rautenstrauch@cispa.de

Igor Santos-Grueiro
The International University of La Rioja
Logroño, Spain
igor.santosgrueiro@unir.net

Ben Stock
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
stock@cispa.de

Abstract

Modern websites behave like OS-native applications and use powerful APIs, such as *camera* or *microphone*. To ensure that untrusted third-party components, such as ads, cannot abuse powerful features granted to web applications, these features are governed via a permission system: containing the `Permissions-Policy` header and `iframe allow` attribute.

Even though the first versions of the permission system were implemented when browsers first allowed access to powerful features more than ten years ago, it is unclear if and how websites are using the permission system. To answer these questions, we systematically measured the permission ecosystem across the top 1,000,000 websites.

Our results show that 48.52% of visited websites exhibit permission-related functionality, and 12.07% of websites delegate permissions to embedded iframes using the `allow` attribute. Out of these delegations, many appear overly broad and unused by the iframe, posing a threat in the context of supply chain attacks. Additionally, only 4.5% websites use the `Permissions-Policy` header, and the primary use case is to turn off powerful APIs such as a camera entirely.

Finally, we developed open-source tools to help developers deploy the correct `Permission-Policy` header and `iframe allow` attributes following the principle of least privilege.

CCS Concepts

• **Security and privacy → Privacy protections**; **Domain-specific security and privacy architectures**; • **Information systems → Web applications**.

Keywords

Browser Permissions; Permissions-Policy; Web Measurement

IMC '25, Madison, WI, USA

ACM ISBN 979-8-4007-1860-1/2025/10
https://doi.org/10.1145/3730567.3764489

ACM Reference Format:
Alberto Fernandez-de-Retana, Jannis Rautenstrauch, Igor Santos-Grueiro, and Ben Stock. 2025. A Permissions Odyssey: A Systematic Study of Browser Permissions on Modern Websites. In *Proceedings of the 2025 ACM Internet Measurement Conference (IMC '25), October 28–31, 2025, Madison, WI, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3730567.3764489

1 Introduction

Nowadays, the World Wide Web has evolved far beyond its original design as a simple network of static documents [3]. It has transformed into a dynamic and interactive platform, supporting a wide array of applications, multimedia content, and real-time communication, fundamentally altering how information is created and consumed across the globe.

To enable such dynamic web applications, browsers allowed websites access to more and more powerful features such as *camera* or *geolocation* that were formerly only available to OS-native applications. Recently, the World Wide Web Consortium (W3C) began the standardization of the permission ecosystem. They introduced the `Permissions` specification [43], that governs how powerful features should prompt for user's choice, and the `Permissions Policy` specification *"that allows developers to selectively enable and disable use of various browser features and APIs"* [42].

There are several studies on how to best display the permission prompts to users [4, 15, 16], browser differences and misleading text in the permission prompts [27, 20], and studies on individual features [28, 6]. Additionally, Kaleli et al. [20] measured the `Feature-Policy` header[1] on 100K websites in 2020, discovering that among the few websites using the header, most used it to turn off features. However, the state of the permission ecosystem at scale remains opaque.

We close this knowledge gap by performing a large-scale measurement on the top 1 million websites. We investigate how many websites use which permissions and how websites delegate and control permissions using the `Permissions-Policy` header and the `iframe allow` attribute. To perform the measurement, we built a pipeline based on Playwright, collected all headers of all documents on a page, and extracted all iframe attributes. Additionally, we instrumented the relevant APIs, dynamically recorded permission invocations, and collected all scripts to perform a static analysis of the relevant APIs.

[1]The predecessor of the `Permissions-Policy` header.

Our results show that 48.52% of websites exhibit permission-related behavior, either dynamically or through static functionality, while 40.65% called some permission-related API alone. Only on 7.98% of websites, embedded documents called permission-related APIs, on 39.41% of websites, the call came from the top-level document; however, in the majority of cases, it came from third-party scripts. 12.07% of websites delegated permissions using the `allow` attribute, mostly relying on the default value, and 4.5% of websites used the header, with the vast majority using it to turn off powerful features such as *geolocation*, *microphone*, or *camera*. Further, more than 6,000 websites had misconfigured headers with invalid syntax or semantics. Additionally, we discovered 36,307 iframes that are potentially over-permissioned as they were delegated a permission, but we did not observe any permission usage. Also, we discovered a bug in the specification, affecting browser vendors, that permits external third party contexts to access permissions, despite the website specifying a self-only policy, via a local-scheme document attack. This is the case when the Content-Security-Policy header of a website does not specify a *frame-src* directive. Lastly, we created tools for developers to learn about the permission ecosystem, supported permissions and deploy secure headers.

In summary, the contributions of our work are:

- We perform a systematic measurement of the browser permission ecosystem on the top 1,000,000 websites according to the July 2024 CrUX list [14] (see Section 4).
- We identify the risk of potentially over-permissioned iframes and present a case study on one Customer Support widgets that could compromise the security and privacy on at least 13,734 websites (see Section 5).
- We identify limitations of the current specifications and develop tools to allow web developers to deploy the correct `Permissions-Policy` header according to the observed permission usage (see Section 6).
- We open-source our framework and tools to support reproducibility, enable future research, and assist web developers [12].

2 Background

In this section, we introduce the key concepts relevant to our work. We first introduce how browser permissions generally work and then explain how the `Permissions Policy` allows websites to govern permissions.

2.1 Browser Permissions

Modern web browsers provide various features that enable websites to offer services with functionalities akin to those of system-wide applications, such as camera access. These features are governed by many individual specifications. For example, the *Media Capture and Streams* specification [47] defines how browsers handle camera access and how to expose these functionalities to web developers.

The W3C distinguishes between regular features and powerful features. Powerful features can pose major privacy, security, and performance concerns and usually require explicit user consent, often through a prompt. In addition to the states *granted* and *denied*, powerful features have a third state: *prompt*. When prompted, users must actively decide whether to grant or deny access. For instance, when a website requests access to camera, the browser may display a dialog asking: *example.org is asking you to: Use your camera.* The user must then choose to allow or block access.

To ensure a consistent framework for managing powerful features, the W3C introduced the `Permissions` specification [43]. This specification standardizes an API for querying the permission state of powerful features (i.e., `navigator.permissions.query`) and provides mechanisms to receive notifications when a permission state changes. For example, a video conferencing website can monitor camera and microphone permissions and immediately trigger once the user grants access.

2.2 Permissions Policy

Depending on whether a feature is *powerful*, the browser might prompt the user for access if it is first used. However, any top-level code on a website, including third-party libraries, could call an API that results in a prompt, and it is unclear what happens if embedded documents use such APIs. To standardize rules for all features, enhance security and privacy, and allow websites to govern permissions themselves, the W3C introduced the `Permissions Policy` specification [42] [2].

The specification requires other specifications to define default allow lists for each feature and introduces an HTTP response header called `Permissions-Policy` and a corresponding attribute for the `<iframe>` HTML element (called `allow`), allowing websites to govern permissions. From now on, we call all features in browsers *permissions*.

2.2.1 Default Allowlist. The default allowlist decides in which contexts a permission can be used by default. There are two possible default allowlists: *self*, which allows the permission only in the same-origin website context, and *, which permits it in all contexts, including arbitrarily nested third-party iframes. This allowlist can be further restricted or relaxed using the `Permissions-Policy` header or the `allow` attribute.

2.2.2 Iframe `allow` *Attribute.* The `allow` attribute on iframes can delegate or restrict permissions to the corresponding iframe. For example, a website can use `allow="gamepad 'none' "` to restrict an iframe from using the *gamepad* permission, or use `allow="camera"` to allow an iframe access to the *camera* permission. It is important to note that only permissions that a website has access to itself can be delegated, and that the permission is only about whether the corresponding APIs can be called; the browser still decides whether to prompt the user or not.

2.2.3 Permissions-Policy Header. Unlike the `allow` attribute, the `Permissions-Policy` header can only further restrict the available permissions. For example, a developer can configure the header as `camera=(), geolocation=(self "https://iframe.com")` to allow only the main website and an embedded iframe from https://iframe.com to request *geolocation*, while completely disabling *camera* access for all contexts. Note that *geolocation* has a default allowlist of *self* and thus, assuming the website is hosted at https://example.org, for an iframe to receive this permission, it requires

[2] Formerly known as Feature Policy

Table 1: Example of Camera Permission Possibility to Prompt and Delegation

	Visited Website (example.org)			*Embedded (iframe.com)*
	Top-Level Permissions-Policy Header		Iframe Delegation	
#	Description	*Header value*	Camera Prompt and Delegation Capability	HTML *allow* value	Camera Prompt and Delegation Capability
1	No header		✔		✘
2	No header		✔	*camera*	✔
3	deny	*camera=()*	✘	*camera*	✘
4	allow self	*camera=(self)*	✔	*camera*	✘
5	allow all	*camera=(*)*	✔		✘
6	allow all	*camera=(*)*	✔	*camera*	✔
7	allow necessary	*camera=(self "https://iframe.com")*	✔	*camera*	✔
8	allow iframe	*camera=("https://iframe.com")*	✘	*camera*	✘

✔ represents "allowed", ✘ represents "blocked".

Table 2: Example of Permissions Characteristics

Permission	Powerful	Policy controlled	Default Allowlist
`camera`	✔	✔	self
`geolocation`	✔	✔	self
`gamepad`	✘	✔	*
`notifications`	✔	✘	N/A
`push`	✔	✘	N/A

✔ represents "yes", ✘ represents "no".

an `allow` attribute delegating the permission and no `Permissions-Policy` header of the frame itself restricting use of the permission.

2.2.4 Permissions Policy Example. Whether some JavaScript code is allowed to call the related APIs and delegate the permission, depends on the default allowlist of the permission, whether the parent context has access to the permission, the `Permissions-Policy` header of the parent context and the own context, and the iframe `allow` attribute.

In Table 1 we provide several examples to explain the interplay of the variables. In these examples, we consider the *camera* permission, which has a default allowlist of *self*, i.e., by default it is only allowed in same-origin contexts. In the table, we show the header of the top-level website (example.org), whether the top-level site itself has access to the permission, the `allow` attribute of the embedded iframe (`iframe.com`) and whether the iframe has access to the permission. For simplicity, we assume the iframe always uses no header or a header that allows the permission.

For case #1, no `Permissions-Policy` header is declared, thus the browser applies the default allowlist of the permission, which is *self* for *camera*, restricting the permission to the top-level context and same-origin iframes. However, in case #2, the permission is explicitly delegated via the `allow` attribute, and the iframe can access the camera. Note that the browser prompt would display: *example.org is asking to use your camera* instead of listing the site of the embedded iframe.

Cases #3 and #4 illustrate how developers can restrict permission usage through the header. Cases #5 and #6 describe a scenario where the * policy is broader than the permission's default allowlist, resulting in the same results as #1 and #2. Case #7 represents the optimal header configuration for developers who want to allow camera access on this specific iframe. The final case #8 highlights a limitation of the specification: currently, it is impossible to delegate permissions without also allowing the *self* context [39].

2.2.5 Common Misconceptions. It is important to note that not all policy-controlled permissions are powerful features, and not all powerful features are governed by policies. Table 2 shows various common permissions with different characteristics, highlighting which are policy-controlled (and therefore have an allowlist) and categorized as powerful.

If policy-controlled and powerful permissions are delegated to an embedded context and prompted for use, they will usually only reference the visited website and not the embedded document requesting the permission. The only permission prompt explicitly mentioning the embedded document requesting usage is *storage-access*.

In addition, once a permission is delegated to an embedded document, the developer of the top-level website can no longer prevent nested delegations. That is in all our examples in Table 1 where the iframe has access to the permissions, e.g., #7, the iframe can further delegate this permission to any other nested iframe regardless of the top-level header and `allow` attribute.

2.2.6 Support of the Specification by Browser Vendors. The specification is inconsistently supported across browsers. All major browsers partly support the `allow` attribute, but only Chromium-based browsers support the `Permissions-Policy` header. Firefox plans to add support soon [25], and Safari also seems to favor implementing it [50]. Additionally, the `Feature-Policy` header, which uses a different syntax, is still supported and enforced in Chromium-based browsers if there is no `Permissions-Policy` header.

3 Methodology

This section describes the framework used to collect data on response headers, permission usage, and permission delegation. We also present the framework setup, including the website list and the measurement instantiation process.

3.1 Framework Overview

Our web crawling system, based on the Playwright automation library [24], collects permissions usage and delegation data from each frame encountered during website navigation as well as the response headers. Below, we provide a detailed description of the data collected.

3.1.1 Permission Usage: To collect permission usage, we use two approaches: a static method that performs string matching of permission-related Web APIs in the scripts loaded by the website, and a dynamic method that involves recording the invocation of Web APIs related to permissions while visiting the website. The static method identifies cases where a permission may be hidden behind a user interaction (e.g., *clicking a button*), while the dynamic method records actual calls to permission-related functionality while visiting the website. To assess the effectiveness of static analysis in overcoming lack of interaction, we performed manual testing on three sets of 25 websites (details in Appendix A.2). Obfuscated calls remain observable through the dynamic approach. Figure 1 provides an example of the dynamic technique for recording invocations, where the original function is overwritten to log the call, stacktrace and its arguments before executing the original function. It is important to note that the instrumented function continues to work as expected. The stacktrace enables us to determine the origin of a call, recording the origin of the script who invoked the corresponding API. Additionally, for certain functions that use permissions as arguments to check their status (e.g., *denied*), analyzing these arguments enables us to identify which specific permissions are being checked. The dynamic instrumentation code is injected before the website is able to execute any content. Both approaches capture inline or dynamically generated scripts. Refer to the Appendix A.4 for a comprehensive list of all permissions we instrumented. In addition to the permission-related Web APIs, we also recorded the general Web APIs defined on the Permissions, Permissions Policy and the deprecated Feature Policy specifications. In summary, the joint use of static and dynamic analysis offers the most comprehensive perspective, partially addressing the limitations of each in isolation (see Appendix A.2). Our instrumentation approach is consistent with practices in related studies in our area [10, 30].

3.1.2 Permission Delegation (Iframe): For documents embedded as iframes, we collect a predefined list of common attributes of the HTML `<iframe>` element in which they are included; `id`, `name`, `class`, `src`, `allow`, `sandbox`, `srcdoc` and `loading`. Particularly important for our work is the `allow` attribute that controls the permission delegation. This attribute can contain a single permission or a list of permissions accompanied by their respective directives.

3.1.3 Website Permission Control (Headers): We collect the `Permissions-Policy` and `Feature-Policy` response headers from each frame in a document, no matter the depth of the frame within the document's structure. We collect the `Feature-Policy` header because, despite being deprecated in favor of `Permissions-Policy` header, Chromium-based browsers continue to support it.

```
// Save original function
var origFunc = navigator.permissions.query;
// Instrument the function
navigator.permissions.query = function (...params) {
    // Get call stack
    let stacktrace = new Error().stack;
    // Save invocation, params and stacktrace
    save(params, stacktrace);
    // Call original function
    return origFunc.apply(this, [...params]);
}
```

```
// Simulate website call
navigator.permissions.query({name: 'camera'})
```

Figure 1: Example of Function Instrumentation.

3.2 Measurement Instantiation

For this work, we used July 2024 CRuX dataset [8, 14] visiting the top the top 1,000,000 origins. Regarding crawling options, we allow up to 60 seconds for the website to fully load and trigger the `load` event. Following this, the crawler pauses for 20 seconds without any interaction. During data collection, the only interaction with the website occurs when a lazy-loaded iframe is detected. To ensure the embedded document loads and maximize data collection, the crawler scrolls to the frame, triggering the browser to load it.

We performed the crawl between August 23, 2024, and September 1, 2024, utilizing 40 parallel crawlers. Each website was crawled once, and the entire process was executed from the same server and Autonomous System Number (ASN) in Germany (EU).

4 Measurement In The Wild

In this section, we present the results of the analysis of the landing pages of the top 1M websites. Out of these, our measurement succeeded on 817,800 websites. For websites that could not be successfully visited, the failures were primarily: 60,183 websites experienced error collecting ephemeral content information (e.g., *Execution context was destroyed*), 28,700 websites experienced timeouts waiting for the page loading, 27,733 websites were not reachable due to major errors (such as DNS errors: `ERR_NAME_NOT_RESOLVED`), 315 websites exhibited minor errors in the crawler (e.g., unexpected values from Playwright or website crashing the crawler) and 90 websites provoked a timeout on the last update made after the specified waiting time on the website. We also applied specific filtering to the collected websites by excluding incomplete iframes or those with errors. This filtering resulted in the exclusion of a total of 65,169 websites. These excluded websites include those that reached a timeout during data collection and ephemeral documents. The primary factor for excluding most of these websites was the timeout, which often occurred due to the presence of numerous included frames on the website. To ensure the completeness of the analyzed data, we opted to exclude these websites. Furthermore, excluded websites relative to the total volume of data represents 20% of the overall total, which is consistent with other published research [31, 32].

The data collection process took approximately nine days, with an average of 35 seconds per website. For a total of 817,800 websites, we collected 2,718,437 frames, consisting of both top-level documents and embedded frames. Specifically, 1,121,018 frames were top-level documents (e.g., the initial document load and redirects), while 1,597,419 were embedded within the website (e.g., iframes). From this point onward, all comparisons are made with respect to the documents, rather than the initial list of websites. From the 1,121,018 visited top-level documents, we identified 1,062,824 distinct origins. Among the websites visited, 545,858 contain at least one iframe. In this 545,858 websites, there is an average of 3.2 iframe elements included directly. Among the embedded frames, 54.1% are local documents, and 45.9% were loaded from external URLs. We refer to local document iframes as documents that do not initiate a network request or include HTTP headers. This includes Local-Scheme documents (*about:*, *data:* and *blob:*), as defined by the *Fetch* Standard [52], and iframes created using the *javascript:* scheme.

Table 3: Top 10 External Embedded Documents Site

Embedded Document Site	# Websites including
google.com	53,227
youtube.com	28,024
doubleclick.net	25,968
googlesyndication.com	25,299
facebook.com	20,919
yandex.com	18,868
twitter.com	17,844
livechatinc.com	13,776
criteo.com	13,491
cloudflare.com	13,395
Total (any site)	304,865

Table 3 presents the top ten external embedded document sites, showing how frequently each embedded document appears at least once on a visited website. The table highlights the dominance of Google, ads-related and social media services, while also revealing the popularity of others such as customer support services from *livechatinc.com*.

4.1 Permission Usage

This section focuses on the analysis of invocations related to permissions and the static analysis in permission-related functionality embedded in the scripts loaded by websites, including inline scripts. We report permission usage in three categories. Two categories arise from dynamic analysis, capturing invocations and status checks, while the third comes from static analysis. In addition to individual permissions functionality, we include functions defined in the Permissions/Feature Policy and Permissions specifications [42, 43], referred to as General Permissions APIs. For these metrics, and throughout the results, we count only the first occurrence for each permission in each frame. This ensures that outliers, which might repeatedly invoke, check, or present functionality for the same permission, do not artificially inflate the results. Throughout this section and the remainder of the paper, we define first-party scripts as those originating from the same site as the context/document under analysis, and third-party scripts as those from any other site.

4.1.1 Permissions-related invocations (Dynamic). Starting with the invocations observed during data collection, a total of 455,676 (40.65%) of the websites we visited show some form of invocation related to permissions, either by using them or checking the status of allowed permissions. Specifically, 39.41% of total websites have invocations in top-level documents, while 7.98% show this activity in embedded documents. However, although most websites show permission-related activity in the top-level document, the majority of this activity comes from third-party scripts. We refer to a third party when the site of the script differs from the site of the frame. In cases where the origin of a call is absent from the stack trace or is an inline script, we classify the call as first-party. Analyzing the stack trace reveals that 98.32% of unique context invocations in the top level originate from third-party scripts. Embedded documents present a different picture, with 74.86% of the activity coming from first-party scripts. We refer to first-party scripts in an embedded document as those that share the same site as the embedded document itself, rather than the top-level site.

Table 4 shows the 10 most frequently invoked permission functionalities. The table also indicates whether invocations are made by first-party or third-party scripts. If both occur in the same context, it is counted once overall but contributes one to both categories, which may cause the combined percentages to exceed 100%. Across 455,676 websites with any invocation, we observe 441,831 top-level documents and 143,863 iframes invoking permission-related APIs, resulting in a total of 585,694 execution contexts with such activity. As shown in the table, for top-level documents, nearly all calls (98.32%) originate from third-party (3p) scripts, whereas for embedded documents, the majority (74.86%) of calls come from first-party (1p) scripts. The most commonly used functionality, by a significant margin (482,309 unique contexts), involves general specification APIs [42, 43] that check the permissions allowed in particular context or the status of a specific permission. General Permission APIs refer to APIs listed in the Permissions Policy and Permissions Specification, including those previously referred to under the deprecated term 'Feature Policy'. Many of the observed General Permissions API calls, most of them from third-party scripts in top-level, may serve benign purposes such as anti-bot detection. However, they also enable fingerprinting by revealing differences in permission support across browsers and even across versions of the same browser. To the best of our knowledge, we are the first to suggest that permission lists could fingerprint browsers and versions, though our data does not confirm such use. Another scenario is that scripts access the entire permission list, performing checks as a prerequisite for execution or to save the state for subsequent user interactions. Another important observation, discussed in detail in Section 6, concerns the use of General Permission API invocations. Specifically, a significant proportion of these invocations originate from the deprecated Feature Policy API, likely due to the API's renaming not yet being implemented. Notably, 429,259 of the websites visited, whether in top-level or embedded contexts, still rely on the Feature Policy API. These data suggest that if browser vendors rename the function, they should consider how many websites still

Table 4: Top 10 Permissions Used At Least Once Across Top-Level and Embedded Contexts

Permission	Top-Level Contexts (Invoked by 1P / 3P Scripts %)	Embedded Contexts (Invoked by 1P / 3P Scripts %)	Total Contexts
General Permission APIs [42, 43]	432,795 (2.91%/98.77%)	49,514 (62.44%/38.32%)	482,309
Battery	38,217 (14.79%/87.94%)	68,815 (96.83%/3.21%)	107,032
Notifications	55,594 (13.88%/89.18%)	1,654 (24.12%/77.15%)	57,248
Browsing Topics	16,033 (1.98%/98.05%)	26,072 (94.96%/5.04%)	42,105
Storage Access	106 (26.42%/73.58%)	16,438 (2.31%/97.69%)	16,544
Public Key Credentials Get	5,774 (100%/96.92%)	579 (100%/100%)	6,353
Geolocation	4,501 (81.03%/19.64%)	123 (91.06%/8.94%)	4,624
Encrypted Media	1,274 (36.34%/63.89%)	996 (23.29%/76.71%)	2,270
Payment	571 (45.01%/54.99%)	668 (20.21%/79.79%)	1,239
keyboard-map	862 (62.41%/37.59%)	306 (99.02%/0.98%)	1,168
Total (any permission)	441,831 (6.54%/98.32%)	143,863 (74.86%/26.15%)	585,694

rely on the old name of the specification. Ranking second, with 107,032 unique contexts, is the *Battery* permission. As highlighted in previous research [28], this may indicate potential tracking purposes by accessing the user's battery level. Ranked third is the *Notification* permission, with 57,248 unique calls. This permission has been extensively discussed in the literature due to its potential for abuse through unwanted notifications [4, 16]. As a result, the Notification permission is not policy-controlled: only the top-level context can request it, and it cannot be delegated. This explains the high counts at the top level and the low counts in embedded contexts. The last permission we wish to highlight from the table is *Browser Topics*, with 42,105 unique calls. Proposed by Google but rejected by other vendors such as Mozilla [26] and Safari [49], it still ranks in fourth position. Overall, permission usage is dominated by third-party components, either scripts or embedded contexts, with an exponential decline in activity as rank increases.

Table 5: Top 10 Permission's Status Checked

Permission	% Checked From Embedded	# Top-Level Websites
All Permissions	4.34%	405,302
Attribution Reporting	1.11%	126,565
Browsing Topics	21.23%	40,732
Notifications	5.99%	20,548
Geolocation	11.93%	8,826
Microphone	10.83%	6,905
Run Ad Auction	3.27%	6,512
Camera	10.79%	6,199
MIDI	10.47%	6,066
Push	10.37%	6,064
Total (any permission)	43.1%	435,185

4.1.2 Invocations for Permission Status (Dynamic). Websites or embedded documents may need to determine the status of specific permissions. To accomplish this, they can utilize general permission APIs defined in the specifications [42, 43]. 435,185 websites utilize these functionalities, either at the top level or within embedded documents. Among these websites, 433,555 demonstrate this activity at least at the top level, while 187,555 exhibit it at least within embedded documents. For top-level documents that implement permission status checks for specific permission, the mean number of permissions checked is 1.74, reaching a maximum of 33 different permissions checked. Table 5 displays the most frequently checked permissions, ranked by the number of websites where each permission is checked, whether at the top level or within embedded documents. Additionally, we present the percentage of cases in which each specific permission was checked from within an embedded document. The table illustrates that websites tend to retrieve the complete list of allowed permissions, rather than querying individual permissions. Such usage may suggest anti-bot or tracking behavior, or it may just be retrieving the full permission list to run a few checks rather than invoking individual calls repeatedly. Checking ad-related permissions (e.g., *Attribution Reporting*) is also common, while the remaining permissions are checked much less frequently. The table also shows that powerful permissions are checked, with the *Microphone* permission being checked on up to 6,899 different websites without any user-interaction. According to our data, ad-related permissions are primarily checked by third-party scripts, while powerful permissions are more commonly checked by first-party scripts.

4.1.3 Detected Permissions Functionality (Static). 341,924 (30.5%) of websites visited present permission functionality at any level. 306,914 cases are found at the top level only, with 128,676 cases occurring solely in embedded contexts. Similar to the previous case, Table 6 presents the top 10 permissions for which websites have functionality in their scripts. The lower detection rate, compared to dynamic analysis, arises because string-based matching does not account for variable assignments, aliases, or other syntactic variations, thus failing to identify semantically equivalent calls expressed in alternative forms or through obfuscation techniques [53].

4.1.4 Summary. 48.52% of the visited websites showcase any permission-related functionality that is either dynamic invocations or permission-related functionality found in the static analysis.

Table 6: Top 10 Statically Detected Permissions

Permission	% Functionality in Embedded	# Top-Level Websites
Clipboard Write	38.31%	135,694
Storage Access	67.62%	106,495
Geolocation	33.19%	96,429
Notifications	9.43%	88,953
Battery	50.52%	63,243
Web Share	54.04%	54,995
Browsing Topics	32.49%	50,346
Encrypted Media	79.44%	44,867
Camera	26.87%	26,456
Microphone	26.87%	26,456
Total (any permission)	37.63%	341,924

Table 7: Top 10 External Embedded Documents with Delegated Permissions

Embedded Document Site	# Top-Level Websites
googlesyndication.com	20,279
youtube.com	18,044
facebook.com	17,720
doubleclick.net	17,634
livechatinc.com	13,734
cloudflare.com	13,244
criteo.com	4,834
stripe.com	3,582
google.com	2,634
vimeo.com	2,028
Total (any site)	121,043

Permission-related activity is predominantly contributed by third-party components, both scripts and embedded contexts. Our measurements indicate that most activity is concentrated on a small number of permissions. Furthermore, much of the observed activity involves retrieving the full set of available browser permissions. This may serve to avoid repeated calls for individual permissions, or as an inefficient way to check a single permission. The permission list might also be used for anti-bot purposes to confirm an authentic browser, or for tracking, allowing distinction between browsers and their versions. Other common permissions observed in our results are mainly associated with advertising, such as *browsing topics*, while others, like *battery*, *geolocation*, or *keyboard-map*, are possibly used for tracking. Our data also highlights permissions supporting typical website functionality, for example, *encrypted media* for video playback or *clipboard-write* for sharing links.

4.2 Policy-Controlled Permission Delegation

In the following, we present the results of explicit permission delegation through the `allow` attribute as defined by the Permissions Policy Specifications. For simplicity, we consider only directly inserted embedded documents and do not account for documents embedded within other embedded documents.

Table 8: Top 10 Delegated Permissions to External Embedded Documents

Permission	Delegations	# Top-Level Websites
autoplay	90,566	61,663
encrypted-media	65,513	38,833
picture-in-picture	58,688	36,375
clipboard-write	53,755	34,049
fullscreen	39,630	32,485
attribution-reporting	65,388	25,006
microphone	29,102	24,368
run-ad-auction	54,505	23,689
join-ad-interest-group	35,702	23,164
gyroscope	36,566	20,005
Total (any permission)	682,883	121,043

A total of 135,341 (12.07%) of visited websites delegate permissions to embedded documents in the landing page. When considering only external URL iframes included in the top level document, the delegation percentage decreases to 10.8%, amounting to a total of 121,043 of websites. From this group, a total of 119,778 websites include a third-party iframe with delegation, i.e., a top level document that loads an embedded document from a different site. Table 7 presents the most common external embedded documents with delegated permissions, along with the number of websites where they appear with delegation. The table shows that *Google* Ad services and *Youtube* media are the most popular, followed by other social media and multimedia platforms like *Facebook*. Close behind, the customer support widget *LiveChat* appears; however, unlike the others, it delegates powerful permissions such as *camera*, *microphone*, and *display-capture*. We explore *LiveChat* further in Section 5.1. We identify 34 distinct sites that are present in at least 100 of the most visited websites. However, this number decreases to just 13 sites that appear in at least 1,000 websites. This indicates that, although many sites are included through delegated permissions, the number of sites that are consistently present across a larger set of websites is significantly smaller. In contrast to the data presented in Table 3, we observe two distinct groups. On one hand, there are extreme cases where sites, such as *yandex.com* or *google.com*, appear with delegation in a very low percentage of instances. This shows that their functionality does not require a specific permission, or that the required permissions are granted because of a wildcard allowlist. In the case of *google.com*, only 4.95% of iframe occurrences include a delegated permission. On the other hand, certain sites, such as *youtube.com* and *livechatinc.com*, are included with delegation in nearly all cases. For *livechatinc.com* when included as an embedded iframe, permission delegation was present in 99.69% of cases. This may suggest that delegation is necessary for the intended functionality of these components.

4.2.1 Delegated Permissions. To explore the questions related to the types of delegated permissions, we refer to the data in Table 8. Among the most delegated permissions, we find powerful permissions, such as *microphone*, or permissions that do not require delegation because their default is *, like *picture-in-picture*. We can infer two things. First, websites that include the delegation of powerful permissions create a chain of trust with the associated risks. This associated risk means that if the permission was not previously granted, it would be requested on behalf of the top-level document. Second, either embedded document developers do not understand the standard, are unaware of the policy-controlled permission defaults, expect permission default to change in the future, or prefer to explicitly name the permissions used in their iframe.

We also observed that permission delegations often exhibit clear grouping patterns. Based on our data, the primary purposes of embedded documents with delegated permissions include:

- **Ads-Related** (e.g., Google Syndication, DoubleClick): *attribution-reporting*, *join-ad-interest-group*, and *run-ad-auction*.
- **Social Media and Multimedia** (e.g., Youtube, Facebook): *autoplay*, *clipboard-write*, *fullscreen*, *encrypted-media*, *picture-in-picture* and sensors such as *accelerometer*.
- **Customer Support** (e.g., LiveChat, LaDesk): *camera*, *microphone*, and *display-capture*.
- **Payment-Related** (e.g., Stripe, RazorPay): *payment*.
- **Session-Related** (e.g., Google Account): *identity-credentials-get* and *otp-credentials*.
- **Others** (e.g., Cloudflare, HCaptcha). *cross-origin-isolated* or *private-state-token-issuance*.

Apart from the specific categories, there are cases that may fall into more than one category. *WixApps*, included in 246 websites, is one such example. In this case, it always delegates *autoplay*, *camera*, *microphone*, *geolocation* and *vr*. Through manual investigation, we suspect that some of these cases, which fall into different categories, may represent widgets with multiple possible purposes. These cases, which potentially use a template with all possible used permissions, delegate permissions that may never be used, thereby creating an unnecessary security/privacy risk (see Section 5).

4.2.2 Directives of Delegated Permissions. From the directives used in the delegation, 82.12% of them do not specify an explicit directive in the delegation, defaulting to *src*, while 17.17% specify * as the directive. Defaulting to *src* means that only the origin specified in the iframe's *src* attribute is permitted. In addition, 0.40% of websites explicitly set *src* as the directive, while 0.15% opted out of delegation by specifying *none*. A small fraction (0.16%) explicitly restricted access by allowing only a single source, for example, by setting a specific *src* value. The high percentage of wildcard (*) usage indicates that many developers prioritize convenience over security, opting for a permissive configuration that may introduce significant privacy/security risks. The use of the *none* directive is low, with only 245 instances of embedded content incorporating the directive in their delegation. Despite some permissions, such as *camera* access, previously being on the * default allowlist, and others potentially changing in the future, this number remains low.

4.2.3 Summary. Our analysis indicates that permission-delegating widgets are frequent, and a limited set of them are widely deployed

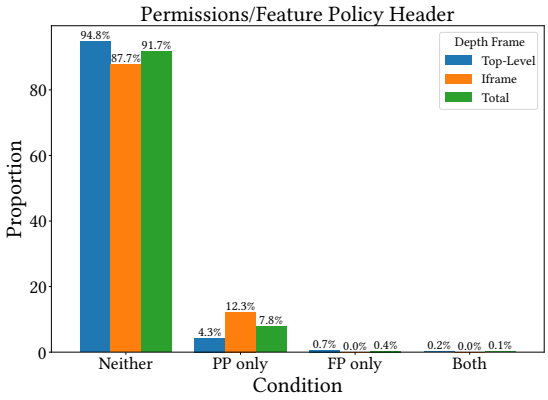

Figure 2: Permission Control headers adoption

across the web. The delegation of permissions, as observed from the data, depends on the functionality of the embedded document, spanning from ads to customer support. In some cases, such as general embedded documents with diverse functionalities, a fixed set of predefined permissions is always delegated, irrespective of the actual functionality. We find that powerful permissions are frequently delegated, raising potential concerns, given the risks associated with such delegation. Furthermore, 82.12% of the delegations, by not specifying a directive, default to *src*, allowing only the loaded source to receive the delegation. In contrast, 17.17% of delegations use the * directive, enabling any origin to access the delegation, even in the case of redirections.

4.3 Permission Control: Permissions-Policy Header and Feature-Policy Header

In this section, for analyzing the results, we excluded local document iframes (e.g., `data:`) due to the lack of headers. This approach excludes embedded documents without headers, preventing bias in the reported results.

Figure 2 illustrates that the adoption of the `Permissions-Policy` header is 7.90%, while the usage of `Feature-Policy` header is 0.51%. Among the two groups, websites that utilize the `Permissions-Policy` header and those that employ the `Feature-Policy` header, there exists a small overlap of 2,302 websites that declare both headers. Given the negligible usage of the `Feature-Policy` header and its deprecated status, we concentrate exclusively on the `Permissions-Policy` header for the remainder of this section. Our data indicates that 157,048 documents use the header, with 50,469 originating from top-level documents (4.5% of top-level total) and 106,579 coming from embedded documents (12.3% of embedded total).

The usage of the `Permissions-Policy` header in iframes is almost three times higher than in top-level documents. The main reason seems to be commonly embedded iframes for applications like advertisements (e.g., `doubleclick.com`) or video content (e.g., `youtube.com`) that use the header.

4.3.1 Common Usage in Top-Level Document. Out of 50,469 instances implementing the header, 47,681 are correctly parsed by the browser. The header, defined in websites declares an average of 10.01 permissions in the header. The most common number

Table 9: Top 10 Permissions-Policy header least restrictive directives for Top-Level Documents

Permission	Disable	Self	Same Origin	Same Site	Third-party	All *	# Websites
geolocation	23,559 (70.58%)	7,580 (22.71%)	234 (0.70%)	138 (0.41%)	232 (0.70%)	1,637 (4.90%)	33,380 (100%)
microphone	28,248 (89.55%)	2,546 (8.07%)	7 (0.02%)	27 (0.09%)	77 (0.24%)	638 (2.02%)	31,543 (100%)
camera	27,070 (87.74%)	2,884 (9.35%)	6 (0.02%)	16 (0.05%)	90 (0.29%)	786 (2.55%)	30,852 (100%)
gyroscope	23,745 (93.54%)	1,264 (4.98%)	0 (0.00%)	15 (0.06%)	24 (0.09%)	336 (1.32%)	25,384 (100%)
payment	21,933 (86.62%)	1,845 (7.29%)	4 (0.02%)	6 (0.02%)	65 (0.26%)	1,467 (5.79%)	25,320 (100%)
magnetometer	23,765 (94.33%)	1,126 (4.47%)	0 (0.00%)	7 (0.03%)	0 (0.00%)	296 (1.17%)	25,194 (100%)
accelerometer	21,208 (93.28%)	1,162 (5.11%)	0 (0.00%)	10 (0.04%)	31 (0.14%)	326 (1.43%)	22,737 (100%)
usb	21,014 (94.57%)	1,068 (4.81%)	0 (0.00%)	7 (0.03%)	2 (0.01%)	130 (0.59%)	22,221 (100%)
sync-xhr	16,258 (80.29%)	2,694 (13.30%)	211 (1.04%)	73 (0.36%)	73 (0.36%)	940 (4.64%)	20,249 (100%)
interest-cohort	18,904 (99.13%)	117 (0.61%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	49 (0.26%)	19,070 (100%)
Total (any permission)	398,356 (83.5%)	46,203 (9.68%)	508 (0.11%)	757 (0.16%)	2,533 (0.53%)	28,701 (6.02%)	47,681

of permissions defined in the header are 18 (26.62%), 1 (24.33%), and 9 (8.47%) permissions. These groups of defined permissions suggest that developers are copy-pasting directives from online sources. The maximum number of declared permissions is 64. Based on our tool [11], none of the websites implement a directive for all supported policy-controlled permissions. Table 9 presents the most commonly declared directives for top 10 permissions in the `Permissions-Policy` headers of top-level visited websites. In the table, we report the least restrictive configuration observed. Thus, if a website specifies that *display-capture* may only be used in its own context *self* and in third party iframe (e.g., *ads.com*), it would be counted as one directive for *third-party*, plus one in the number of websites declaring the permission. As the table shows, 83.5% of websites — which deploy the header — completely disable the use of features. Another 9.68% restrict their use to their own context, with these two cases accounting for a total of 93.19%. If we consider only powerful permissions, permissions that need user interaction, the percentage of websites declaring a disable or self directive increases to 97.08%. The table also shows that using specific directives to declare the origin of embedded documents is not very prevalent. Most websites do not utilize most of the permissions, and developers often use the header to disable a subset in order to mitigate security and privacy risks. Finally, 6.02% of websites declared directives with '*', which has no real effect, as the header is designed for restricting permissions, not for allowing them. This directive may suggest that developers are explicitly indicating the use of the feature.

4.3.2 Common Usage in Embedded Documents. In the case of embedded documents, contrary to what is shown in Table 9 for top-level headers, the nine most prevalent permission directives are for features related to User-Agent Client Hints [48]. In these features, the most common directive for these permissions is to allow all ('*'), which, as explained, effectively has no impact because the header can only enforce restrictions. In the case of the declared directives, disabling permissions is significantly less frequent compared to top-level documents, accounting for 51.05%. Directives for own context make up 16.89%, while those allowing all ('*') represent 30.73%. In contrast to top-level documents, where 56.29% of total directives pertain to powerful permissions, this figure drops to 26.30% in embedded documents. Although the picture may be influenced by some of the widely used social media and video entertainment iframes, it might still reflects the pattern that, in embedded documents, there is less concern about opting out of permissions that are not being used.

4.3.3 Misconfigurations. Of the 157,048 frames analyzed, comprising both top-level documents and iframes that declare the `Permissions-Policy` header, 3,244 (2%) contain syntax errors that results in the browser removing the complete header and not applying the policy. Although the number of cases is small, developers declaring this header, often to enhance the security and privacy of their site, create parsing errors that result in 2,788 of the websites and 456 of embedded documents lacking any permission restriction. These parsing errors in the header causes these websites to fall back to the default permission allowlists. An example of a common parsing error we found is declaring the header using the `Feature-Policy` header syntax. Another common case are misplaced commas, such as ending the header with a comma making the header invalid. Among the headers that parse correctly, we identified another 6,408 visited websites with misconfigurations, such as unrecognized tokens (e.g., *none* or *0*), missing double quotes around urls, contradictory directives (e.g., *self* and *), or url directives lacking *self*, which is not allowed [39]. In embedded documents, this number decreases to 653 visited websites that included an embedded document with a misconfigured header. This pattern suggests that, although less complex than other header solutions like Content Security Policy [51], developers still face challenges when implementing it.

4.3.4 Summary. Based on the presented data, we can conclude that website developers tend to use the header to opt-out of features, such as for mitigating potential misuse on their sites, while embedded document developers typically use the header to explicitly not opt them out without any effect or explicitly indicating the use of the permission. Moreover, developers struggle with deploying the header without making mistakes, declaring directives for all supported policy-controlled permissions, and in the vast majority of cases, only simple directives are implemented. More than 50% of top-level websites adopt one of three identical configurations in the Permissions-Policy header, likely reflecting the use of predefined templates.

Table 10: Top 10 Embedded Documents with Unused Delegated Permissions

Embedded Iframe	Potentially Unused Permissions	# Affected Websites
youtube.com	*accelerometer, gyroscope*	16,394
livechatinc.com	*camera, microphone, clipboard-read*	13,734
facebook.com	*clipboard-write, web-share, encrypted-media*	1,405
youtube-nocookie.com	*gyroscope, accelerometer*	982
razorpay.com	*payment, clipboard-write, camera*	389
ladesk.com	*microphone, camera*	303
driftt.com	*encrypted-media*	285
wixapps.net	*microphone, camera, geolocation*	246
qualified.com	*microphone, camera*	109
dailymotion.com	*accelerometer, gyroscope, clipboard-write, web-share, encrypted-media*	101
Total (any iframe)		36,307

5 Permission Delegation to Embedded Iframes

In this section, we analyze embedded documents running with delegated permissions. In particular, we examine whether the embedded documents with delegated permissions actually require those permissions for their functionality. To determine this, we compare the delegated permissions (Section 4.2) with the dynamic functionality recorded during navigation and the static functionality present on the loaded scripts (Section 4.1). In addition, we present a case study of one of the most common Customer Support widgets.

The threat models for permission delegation have been initially described by other researchers [21, 20]. In this section, we consider a threat model similar to supply chain attacks [2], where a widely embedded document is commonly deployed with delegated permissions. The data from the previous section confirms this is a realistic scenario. When such permissions are granted by default and remain unused, they introduce unnecessary risk, potentially enabling user permission hijacking via the embedded document. As an example scenario of the threat model, consider *support.example.org*, a *Customer Support* widget embedded in thousands of websites. This widget is always included in an embedded document with powerful permission (e.g., *clipboard read*) delegation. If a malicious entity is able to compromise this embedded document, they could hijack permissions across thousands of websites.

To establish an upper bound for potentially over-permissive embedded documents, we first collected, for each embedded origin, all delegated permissions that appeared in at least 5% of iframes of this origin. We selected the 5% threshold as a way to capture the most prevalent delegated permissions while minimizing noise. On the other hand, for each embedded document, we collected all permission-related activity, whether they directly used the permission, checked its status, or included permission-related functionality in the loaded scripts, including dynamically created scripts. With these two cases, we establish a method to verify that the appearance of a delegated permission is not a one-time event, as delegation occurs in at least 5% of cases, while the embedded document do not exhibits any form of functionality for the permission across the entirety of the recorded data.

Table 10 show the top ten embedded documents with delegated permissions that are not utilized nor contain any related API calls in their scripts. We also include the number of websites that delegate at least one of these permissions. The total number of affected websites which include any of the embedded documents delegating unused permission is 36,307 of the visited websites. Considering the delegated permissions, the most risky embedded documents are Customer Support widgets.

5.1 Customer Support Widgets

Customer Support widgets are embedded within top-level documents to offer support functionalities and improve communication with the associated businesses. A common example of these Customer Support widgets are the chat features with company representatives, often appearing in the corner of a website to facilitate question-asking. In most cases, these chat widgets provide basic functionality, which can be enhanced through premium subscriptions or by adding new plugins from their respective marketplaces [22]. We analyze one prevalent case of chat widget that have the potential to undermine the security and privacy of websites.

5.2 LiveChat Case Study

The first case is the *LiveChat* widget [23], which is integrated into a total of 13,753 different websites, with 27 of these appearing among the top 5,000 as reported by CrUX. Of this total, 13,734 websites exhibit overpermissioning. Their widgets do not utilize the `Permissions-Policy` header to mitigate potential risks and are consistently (99.70% of the times) included in the top-level document with the delegation of permissions: '*clipboard-read; clipboard-write; autoplay; microphone *; camera *; display-capture *; picture-in-picture *; fullscreen *;*'. In some of these delegations, they introduce potential risks by using wildcards instead of specifying the origin which uses the permissions. The wildcard directive indicates that redirecting the embedded document to a different origin would also delegate the associated permissions making the hijacking possible. Additionally, the fact that all LiveChat widgets utilize the same permission delegations may imply adherence to a template that systematically enforces these delegations, irrespective of the plugins used and the permissions that are essential. Thus, embedded documents are always included with the same delegation, unaware of the plugins installed from the marketplace [22] by developers.

During our investigation, we found no instances of these widgets performing any permission-related invocations, such as verifying the permitted features nor could we identify the APIs through static analysis of the included scripts. Through a manual analysis by setting up our own *LiveChat* chat widget, we confirmed that even without plugins installed, the permission delegation remains consistent, indicating that the same template is always used. Additionally, after installing free video-conferencing plugins, we observed that the permissions, such as *microphone* and *camera*, were not being utilized. In some cases, instead of requesting permission for these features, the plugin would send a message including a meeting url to an external service.

5.3 Recommendations

In chat widgets and other potential cases from Table 10, delegating permissions may be necessary to enable specific functionalities. However, our findings indicates that permissions are often delegated and then not utilized. In particular, customer support widgets seem to be running overpermissioned. When delegation is indeed required, widget developers should implement existing control mechanisms and avoid using wildcards in their permission delegations to ensure only the legitimate widget can use the permission. Furthermore, permissions should only be delegated when absolutely necessary. Failure to adhere to these best practices could pose significant risks to the top-level website, especially when a permission has already been granted previously. In such cases, the external URL could use the permission, even if the delegation occurred after the permission was granted. Additionally, for website developers, the solution is to explicitly declare the header to disable permissions that are not required for the website's functionality.

6 Discussion

This section outlines our work's limitations and the specification shortcomings revealed by our research, which highlight challenges for developers. We also present a specification issue affecting Chromium-based browsers and propose two solutions to help developers deploy the `Permissions-Policy` header and permission delegation effectively, even in complex cases.

6.1 Limitations

One limitation of our work is the lack of interaction with the website [19, 54], which is partly mitigated by our hybrid approach incorporating the static analysis of scripts for permissions APIs. This lack of interaction may limit the results presented in our study leading to conservative underreporting, such as websites utilizing permissions only after specific interactions (e.g., clicking a button, accepting cookie banner). While static analysis might compensate for the lack of interaction (see Appendix A.2), it may miss obfuscated code and may also report permission functionality contained in dead code that will never be triggered. Dynamic instrumentation instruments only static property accesses, method calls, and object instantiations, without tracking subsequent interactions. For instance, if a page retrieves all allowed features and later checks for a specific one, only the initial retrieval is observed, not the later check. To better understand the limitations of our no-interaction approach, we conducted a small-scale manual experiment, described in Appendix A.2. Another limitation is that our crawler is restricted to the landing page, which limits visibility into features and permission usage that may only appear after navigating through the website [1, 33]. In addition, our automated crawler could be detected by websites, potentially leading to different content being served compared to genuine user visits [18]. Finally, another limitation of this work is the potential bias in results caused by conducting the navigation exclusively from Germany (EU) [5, 19, 34].

6.2 Shortcomings in Specifications

The W3C's proposal of these specifications, along with the browser vendors' agreement on a unified permission model, is promising news for improving the security and privacy of modern websites. However, as other studies and our work have shown, there is still much to be done in the development and adoption of these specifications by both browser vendors and websites.

A key limitation of the current Permissions-Policy specification is the absence of an up-to-date list of existing permissions [40, 46], combined with the lack of a default disallow all directive [38]. The lack of an up-to-date permission list creates confusion for developers [37], while requiring each permission to be explicitly disallowed increases the risk of omissions. Our measurements show that no websites using the header specify a directive for all supported permissions, highlighting the need for a developer-friendly solution. To help address this gap, our open-source tool maintains a curated list of known permissions along with their browser support status (see Figure 3 in Appendix A.6) [11].

A second limitation that must be considered when advancing the specification is the prevalent use of deprecated Feature Policy Web APIs. Browser vendors currently lack support for the Permissions Policy specification (e.g., Permissions Policy Web API), and many functionalities within the web ecosystem, as demonstrated in Section 4, continue to depend on old specification functions (e.g., Feature Policy Web API).

Considering these issues, along with the fact that the header is currently only implemented in Chromium-based browsers [36], it is evident that the development and adoption of the specification are progressing slowly. Furthermore, the complexities in the specification or the renaming of the header have led to numerous questions from developers [41, 45], who do not fully understand its implementation, contributing to this slow adoption. In our data, one of the most common header misconfigurations was the use of `Feature-Policy` syntax within the `Permissions-Policy` header.

Specification Issue/Local-Scheme document attack. We detail a specification issue manually identified in the Permissions Policy specification during our research, which consequently affects all Chromium-based browsers. In the code associated with this research [12], we provide a proof of concept (PoC) that enables testing the specification issue in your browser [13]. This issue has been acknowledged but remains unresolved [44]. Details of the responsible disclosure are provided in the Appendix A.1. The identified specification issue causes local-scheme documents (e.g., data URIs) not to inherit the policies of their parent documents. Table 11 presents an example. If a website (*example.org*) declares a Permissions Policy directive of 'self' for a specific permission (e.g., *camera=(self)*), creating a local-scheme document (such as a *data URI* or *about:srcdoc*) can

Table 11: Example of Behavior and the Specification Issue Found

	example.org	*Local-Scheme document (data: URI)*		*third-party.com/attacker.com*
	Permissions-Policy Header *camera* directive value	Camera Access/Prompt and Delegation Capability	Iframe Delegation HTML *allow value*	Camera Access/Prompt and Delegation Capability
Expected	**camera self**: *self*	✔	delegate: *camera*	✘
Actual Specification	**camera self**: *self*	✔	delegate: *camera*	✔🐞 [44]

✔ represents "allowed", ✘ represents "blocked".

bypass the declared policy, allowing the delegation of that permission to an external URL. As presented in Section 4, this policy is the second most common. This discovered specification issue can lead to unintended behavior, potentially allowing third-party actors—or worse, malicious entities—to exploit it and bypass the restriction efforts implemented by developers. It is worth mentioning that, at present, the 'self' directive is required even when only delegating permissions to embedded documents [39]. This bypass could be effective in scenarios where a victim website uses the *self* directive alongside a strict Content Security Policy (CSP) that mitigates cross-site scripting. In such cases, an attacker could exploit HTML injection to perform permission hijacking if the CSP does not enforce frame restrictions. At worst, an existing permission could be silently hijacked, bypassing the need for user consent.

6.3 Facilitating Defense Deployment

The Permissions Policy header for controlling the use of powerful features still has a low adoption rate. Additionally, the inclusion of embedded documents with delegated permissions may further increase the risk of undermining the security and privacy of websites. One of the goals of our work is to facilitate the adoption of this protection and safeguard the vast majority of websites, regardless of whether they do not intend to use most of the features. To achieve this, we present two solutions.

First, we introduce a website (see Figure 3 in Appendix A.6) [11] that, to the best of our knowledge, provides the most comprehensive list of permissions. The website, similar to *caniuse* [35], also details which permissions are supported and whether they are classified as policy-controlled or powerful by different browser vendors. All these results are generated by an automated tool that tests permissions across major browser releases. Our tool also tracks historical changes across browser versions and includes references to the W3C specifications that define each permission. Consistent with the rest of our contributions, the tool and website are made available as open-source [11].

Additionally, the website contains a `Permissions-Policy` header generator (see Figure 4 in Appendix A.7). Based on the supported permissions across browser vendors, it allows developers to create their own custom header. This supported-permissions list is generated using our previously described tool, keeping it up-to-date with browser changes and ensuring reliable header generation. Additionally, it provides predefined options, such as disabling all permissions or, more commonly, disabling only powerful permissions. Finally, this generator is integrated into the same website that displays the supported-permissions list and is also released as open-source [11].

As a second solution, more suited for complex scenarios, we introduce a tool, similar to our crawler, that interacts with websites by crawling them and allowing developer interaction (e.g., click). After the developer interacts with the site, the tool suggests an appropriate `Permissions-Policy` header and delegation based on observed functionality. Its recommendations rely on the same data used in our study, including header configurations, permission usage, and delegation behavior. Additionally, the tool highlights instances where the actual configuration is broader than the ideal configuration.

7 Related Work

Our work builds on a long tradition of research into web headers for enhancing security and privacy such as like Content Security Policy (CSP) [51, 29], X-Frame-Options (XFO) [29], and Feature-Policy [20]. In addition, previous studies have examined the user experience with browser permission prompts [16, 15], the rationale behind permissions displayed by websites [9], attack models for permission model [20], and the role of individual permissions in enabling tracking [28, 6]. Other research highlighted inconsistencies across browsers in implementing permission mechanisms [27]. Our study complements the existing research by offering the most comprehensive view of the permission ecosystem, focusing on the `Permissions-Policy` header, permission usage, and delegation.

8 Conclusion

In recent years, web browsers have introduced new features. To control these capabilities, the web platform provides a permission system through the Permissions-Policy header and iframe allow attributes. This paper presents the first systematic study of the permission ecosystem across 1M websites. We find that while 48.5% of websites use permission-related functionality, only 4.5% deploy the header, primarily to disable powerful APIs. Moreover, 12% of websites delegate permissions to embedded documents, often granting overly broad access that is not used, posing security and privacy risks. In addition, our analysis identifies 36,307 websites embedding documents that may run with more permissions than necessary. To assist web developers, we introduce manual and automated solutions for applying permission controls effectively. We further uncover a bug in the specification that impacts Chromium-based browsers and publish our tool as open-source to support reproducibility and future work.

References

[1] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M. Maggs. 2020. On landing and internal web pages: the strange case of jekyll and hyde in web performance measurement. In *Proceedings of the ACM Internet Measurement Conference*. (2020). doi:10.1145/3419394.3423626.

[2] Zbigniew Banach. 2024. Polyfill supply chain attack: what to do when your cdn goes evil. https://www.invicti.com/blog/web-security/polyfill-supply-chain-attack-when-your-cdn-goes-evil/.

[3] Tim Berners-Lee. 1989. Information management: a proposal. https://www.w3.org/History/1989/proposal.html.

[4] Igor Bilogrevic, Balazs Engedy, Judson L Porter Iii, Nina Taft, Kamila Hasanbega, Andrew Paseltiner, Hwi Kyoung Lee, Edward Jung, Meggyn Watkins, PJ McLachlan, and Jason James. 2021. "shhh. be quiet!" reducing the unwanted interruptions of notification permission prompts on chrome. In 30th USENIX Security Symposium.

[5] Adrian Dabrowski, Georg Merzdovnik, Johanna Ullrich, Gerald Sendera, and Edgar Weippl. 2019. Measuring cookies and web privacy in a post-gdpr world. In *Passive and Active Measurement*. David Choffnes and Marinho Barcellos, (Eds.) doi:10.1007/978-3-030-15986-3_17.

[6] Anupam Das, Nikita Borisov, and Matthew Caesar. 2016. Tracking mobile web users through motion sensors: attacks and defenses. In Network and Distributed System Security Symposium. doi:10.14722/ndss.2016.23390.

[7] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. 2022. Reproducibility and replicability of web measurement studies. In *Proceedings of the ACM Web Conference 2022*. (2022). doi:10.1145/3485447.3512214.

[8] Zakir Durumeric and David Adrian. 2024. Chrome ux july 2024 snapshot. https://github.com/zakird/crux-top-lists/blob/main/data/global/202406.csv.gz.

[9] Yusra Elbitar, Soheil Khodayari, Marian Harbach, Gianluca De Stefano, Balazs Csaba Engedy, Giancarlo Pellegrino, and Sven Bugiel. 2025. Permission rationales in the web ecosystem: an exploration of rationale text and design patterns. Pre-published.

[10] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: a 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. (2016). doi:10.1145/2976749.2978313.

[11] Alberto Fernandez-de-Retana, Jannis Rautenstrauch, Igor Santos-Grueiro, and Ben Stock. 2025. Browser permissions compatibility and header generator website. https://albertofdr.github.io/browser-permissions-tool/.

[12] Alberto Fernandez-de-Retana, Jannis Rautenstrauch, Igor Santos-Grueiro, and Ben Stock. 2025. Permissions odyssey: source code repository. https://zenodo.org/records/16921477.

[13] Alberto Fernandez-de-Retana, Jannis Rautenstrauch, Igor Santos-Grueiro, and Ben Stock. 2025. Specification issue proof-of-concept repository. https://github.com/AlbertoFDR/browser-permissions-odyssey/tree/main/specification-issue-poc/.

[14] Google. 2024. Chrome ux report. https://developer.chrome.com/docs/crux/.

[15] Marian Harbach. 2024. Websites need your permission too – user sentiment and decision-making on web permission prompts in desktop chrome. In CHI Conference on Human Factors in Computing Systems. (2024). doi:10.1145/3613904.3642252.

[16] Marian Harbach, Igor Bilogrevic, Enrico Bacis, Serena Chen, Ravjit Uppal, Andy Paicu, Elias Klim, Meggyn Watkins, and Balazs Engedy. 2024. Don't interrupt me - a large-scale study of on-device permission prompt quieting in chrome. In Network and Distributed System Security Symposium. doi:10.14722/ndss.2024.24108.

[17] HTTP Archive Project. 2025. Http archive dataset. har.fyi. https://har.fyi/.

[18] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. 2019. Fingerprint surface-based detection of web bot detectors. In *Computer Security – ESORICS 2019*. Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, (Eds.) doi:10.1007/978-3-030-29962-0_28.

[19] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. 2021. Towards realistic and reproducibleweb crawl measurements. In *Proceedings of the Web Conference 2021*. (2021). doi:10.1145/3442381.3450050.

[20] Beliz Kaleli, Manuel Egele, and Gianluca Stringhini. 2020. Studying the privacy issues of the incorrect use of the feature policy. In Workshop on Measurements, Attacks, and Defenses for the Web. doi:10.14722/madweb.2020.23014.

[21] Jun Kokatsu. Permissions policy: self by default slides. (2023). https://docs.google.com/presentation/d/1r-IoO4zATUt4X2KyND16EoDiho5q56KdjqqfR5U7XaY.

[22] LiveChat. 2025. Livechat apps marketplace. https://www.livechat.com/marketplace/.

[23] LiveChat. 2025. Livechat customer service software. https://www.livechat.com/.

[24] Microsoft. 2025. Playwright. https://playwright.dev/.

[25] Mozilla. 2020. Permissions policy standard position. https://github.com/mozilla/standards-positions/issues/24.

[26] Mozilla. 2023. Topics api. https://github.com/mozilla/standards-positions/issues/622.

[27] Kazuki Nomoto, Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. 2023. Understanding the inconsistencies in the permissions mechanism of web browsers. *Journal of Information Processing*, 0. doi:10.2197/ipsjjip.31.620.

[28] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. 2016. The leaking battery. In *Data Privacy Management, and Security Assurance*. Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Alessandro Aldini, Fabio Martinelli, and Neeraj Suri, (Eds.) doi:10.1007/978-3-319-29883-2_18.

[29] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex security policy? a longitudinal analysis of deployed content security policies. In Network and Distributed System Security Symposium. doi:10.14722/ndss.2020.23046.

[30] Iskander Sanchez-Rola, Leyla Bilge, Davide Balzarotti, Armin Buescher, and Petros Efstathopoulos. 2023. Rods with laser beams: understanding browser fingerprinting on phishing pages. In 32nd USENIX Security Symposium (USENIX Security 23). doi:10.5555/3620237.3620470.

[31] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. 2021. Who's hosting the block party? studying third-party blockage of csp and sri. In Network and Distributed System Security Symposium. doi:10.14722/ndss.2021.24028.

[32] Junhua Su and Alexandros Kapravelos. 2023. Automatic discovery of emerging browser fingerprinting techniques. In The ACM Web Conference. (2023). doi:10.1145/3543507.3583333.

[33] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. 2020. Beyond the front page:measuring third party dynamics in the field. In *Proceedings of The Web Conference 2020*. (2020). doi:10.1145/3366423.3380203.

[34] Tobias Urban, Dennis Tatang, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. 2020. Measuring the impact of the gdpr on data sharing in ad networks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. (2020). doi:10.1145/3320269.3372194.

[35] Can I Use. 2025. Can i use... support tables for html5, css3, etc. https://caniuse.com/.

[36] Can I Use. 2024. Permissions policy | can i use... support tables for html5, css3, etc. https://caniuse.com/permissions-policy.

[37] W3C. 2024. Clarify "shipped in chrome" for picture-in-picture github issue. Github Issue. https://github.com/w3c/webappsec-permissions-policy/issues/502.

[38] W3C. 2022. Deny all like alias for the permission-policy: header discussion. Github Issue. https://github.com/w3c/webappsec-permissions-policy/issues/483.

[39] W3C. 2025. Denying self while still allowing subframes discussion. Github Issue. https://github.com/w3c/webappsec-permissions-policy/issues/480.

[40] W3C. 2020. Feature registry discussion. Github Pull. https://github.com/w3c/webappsec-permissions-policy/pull/366.

[41] W3C. 2024. Js playgrounds leak permissions. guidelines and examples needed. Github Issue. https://github.com/w3c/webappsec-permissions-policy/issues/547.

[42] W3C. 2024. Permissions policy standard. https://w3c.github.io/webappsec-permissions-policy/.

[43] W3C. 2024. Permissions standard. https://www.w3.org/TR/permissions/.

[44] W3C. 2024. Permissions-policy header inheritance for local schemes specification issue. https://github.com/w3c/webappsec-permissions-policy/issues/552.

[45] W3C. 2024. Query: can trusted subframe allocate permission to one of it's cross-domain subframe. Github Issue. https://github.com/w3c/webappsec-permissions-policy/issues/542.

[46] W3C. 2024. Update features.md (e.g., 'storage-access' is missing) github issue. Github Issue. https://github.com/w3c/webappsec-permissions-policy/issues/551.

[47] W3C. 2024. W3c media capture and streams standard. https://www.w3.org/TR/mediacapture-streams/.

[48] Web Platform Incubator Community Group. 2025. User-agent client hints. https://wicg.github.io/ua-client-hints/.

[49] WebKit. 2023. The topics api. https://github.com/WebKit/standards-positions/issues/111.

[50] WebKit Bugzilla. 2023. Implement permissions-policy http header. https://bugs.webkit.org/show_bug.cgi?id=253126.

[51] 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 ACM SIGSAC Conference on Computer and Communications Security. doi:10.1145/2976749.2978363.

[52] WHATWG. 2025. Local-scheme - fetch standard. https://fetch.spec.whatwg.org/#local-scheme.

[53] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious javascript code: a measurement study. In 7th International

Conference on Malicious and Unwanted Software. doi:10.1109/MALWARE.2012.6461002.

[54] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollsén, and Martin Lopatka. 2020. The representativeness of automated web crawls as a surrogate for human browsing. In *Proceedings of The Web Conference 2020.* (2020). doi:10.1145/3366423.3380104.

A Appendix

A.1 Ethics

We performed our crawl in an ethical and responsible manner to not disturb websites. The crawl was designed to be non-intrusive and low-impact. We conducted a single-pass crawl of one million publicly accessible websites using a browser-based crawler. Each site was visited only once, and we did not interact with forms or other site elements beyond what is triggered during a standard page load.

Regarding the responsible disclosure of the specification issue, we followed established best practices. After identifying the problem, we reported our findings to the W3C specification group and to a major browser vendor, providing sufficient detail to reproduce the issue. Both acknowledged the report, however, as of one year later, no fix or further communication has been provided.

A.2 Experimental Setup and Reproducibility

To ensure the reproducibility of our study, we report the criteria described by Demir et al. [7].

A.2.1 Dataset. **C1** For our analysis, we use the CrUX July 2024 list [8], which we include in our public repository [12]. From this list, we select the top one million origins to construct our website dataset. **C2** As the CrUX list contains origins, we relied on it without modification and visited each origin a single time given the scope of the measurement. **C3** We make the list openly available [12]. **C4** We visit each origin only once, so multiple measurements do not apply.

A.2.2 Experimental Design. **C5** We relied on Playwright `v1.45.1`, packaged in Microsoft's Playwright Docker image (`v1.46.0-jammy`). **C6** We disabled the `AutomationControlled` Blink feature (`navigator.webdriver`) to reduce detection and used Playwright's built-in scrolling function to trigger lazy-loaded iframes for complete data collection. Instrumentation was injected via built-in functions before pages and documents loaded. Additionally, we implemented a wrapper to perform multiple crawlers concurrently and store the collected data in a database. **C7** Besides the specified adjustments, no additional modifications were made to the browser. **C8** The `AutomationControlled` Blink feature (`navigator.webdriver`) was disabled to reduce bot detection. **C9** We make the used framework publicly available [12]. **C10** Interaction was limited to scrolling lazy-loaded iframes to trigger content loading. **C11** For running the measurement a headful stateless browser was used. When visiting a page, the crawler waits up to 60 seconds for the load event. It then remains on the page for an additional 20 seconds. If any lazy-loaded iframes are detected, the crawler scrolls to them to trigger content loading. After this period, a final data collection is performed, the browser is closed, and the crawler proceeds to the next page. A 90-second timeout is set for each page visit, if exceeded, for example due to many iframes, the page is marked as timed out. **C12** We conducted the crawl from Germany (EU). **C13** The experiments were conducted using Chromium version `v127.0.6533.17`, and no further modifications were applied to the browser. **C14** Upon completion of a site crawl, all collected data for the site and its embedded documents (see Section 3.1) are immediately saved to the database.

A.2.3 Evaluation. **C15** We will make our results publicly available. **C16** We elaborate on this point in Section 4. **C17** We discuss limitations in Section 6.1. **C18** Ethical considerations are discussed in Appendix A.1.

A.3 Experimental Validation of Static Method via Manual Testing

This section aims to evaluate how well the static analysis approach captures permission-related behavior compared to actual activated permissions with user interactions. First, we ran our automated tool to collect static and dynamic reports without any user interaction. Next, we performed manual navigation of the pages, running the tool in the background but adding interactions, including clicking through pages of same origin, visiting multiple paths within the same origin, and in some cases creating simple accounts. We extended manual navigation to multiple paths within the same origin to provide a broader comparison with our approach, as some permissions may only be triggered on specific paths. This also allows a limited evaluation of the known limitation of visiting only landing pages. For each site, we compare the permissions observed in the initial navigation without interaction against those collected in a second run with interaction. This approach provides a more complete view of the permissions a website may request, though it remains limited; some accounts could not be created, and some functionality (e.g., subscription-only features) remained inaccessible.

We conducted three experiments, each with 25 websites, to capture the problem from three perspectives. The first dataset comprises sites showing static permission activity but no dynamic activity randomly selected from the results of our measurement. The second and third datasets were drawn from the HTTP Archive, filtered by the categories "Ecommerce" and "Video players", respectively. We used these website categories under the assumption that they show higher levels of permission activity compared to others. We restricted the websites to the top 5000 for the first dataset, and to the top 1000 for the second and third.

Table 12 summarizes the results. It shows the average number of permissions detected automatically by static and dynamic analyses without user interaction, as done in the main measurement of this paper. During two working days, a researcher interacted with the sites while the tool was running, and the table reports the average number of permissions activated through these manual interactions. The final columns show how many of these permissions were already captured by static analysis and the total captured when including automated dynamic analysis, both of them without any interaction or navigation across the website. In other words, we compared the permissions observed at least once during the automatic crawl with those observed at least once during manual experiment. Looking at the first experiment, websites chosen for having no dynamic activity show a small average of 0.04%, caused

Table 12: Manual Testing of Average Permission Detection Across Experiments

Experiment Information				Avg. Permissions Reported			Detection Results	
				No Interaction		Interaction		
Experiment	Dataset	Details/Category	#	Static ($\pm\sigma$)	Dynamic ($\pm\sigma$)	Activated ($\pm\sigma$)	by Static	by $S \cup D$
1	Results (Section 4)	Static-Only	25	1.84 ±1.28	0.04* ±0.2	1.08 ±0.95	62.96%	62.96%
2	HTTP Archive [17]	Ecommerce	25	1.56 ±2.33	0.44 ±0.82	2.36 ±1.70	22.03%	35.59%
3	HTTP Archive [17]	Video Players	25	2.84 ±2.44	0.28±0.61	1.16 ±1.25	56.67%	73.33%
Avg				2.08	0.25	1.53	40.52%	51.72%

* **One site showed permission use, possibly due to content changes since the experiment.**

by a single website that may have updated its content since the experiment. No-interaction results show that across all three experiments, the static approach reports a higher average number of permissions than the dynamic approach. In two of the three experiments, static reporting even exceeds the permissions activated with manual interaction. This is likely due to several factors, such as functionality that could not be manually triggered (e.g., behind a login), dead code, a set of websites with unobfuscated code and broken features, as observed in one of our experiments, for example, a malfunctioning share button. Among the static permissions reported, certain permissions, such as *clipboard-write*, *geolocation*, and *battery*, appear consistently. Overall, the results indicate that static analysis partially mitigates the limitations of non-interactive crawling. Considering that the automated approach only visits landing pages, whereas manual exploration traverses multiple paths and loads additional content, static analysis proves to be a valuable method to reduce the lack-of-interaction limitation.

A.4 Permission Usage

The complete list of instrumented permissions includes: *accelerometer*, *ambient-light-sensor*, *battery*, *bluetooth*, *browsing-topics*, *camera*, *clipboard-read*, *clipboard-write*, *compute-pressure*, *direct-sockets*, *display-capture*, *encrypted-media*, *gamepad*, *geolocation*, *gyroscope*, *hid*, *idle-detection*, *keyboard-lock*, *keyboard-map*, *local-fonts*, *magnenometer*, *microphone*, *midi*, *notifications*, *payment*, *pointer-lock*, *publickey-credentials-create*, *publickey-credentials-get*, *push*, *screen-wake-lock*, *serial*, *speaker-selection*, *storage-access*, *system-wake-lock*, *top-level-storage-access*, *usb*, *web-share*, *window-management* and *xr-spatial-tracking* many which contain several instrumented APIs.

In addition to permissions, our instrumentation targets the following general permission purpose APIs: *Permissions API (navigator.permissions)*, *Permissions Policy API (document.permissionsPolicy.*)* and *Feature Policy API (document.featurePolicy.*)* .

A.5 Embedded Documents with Potentially Unused Delegated Permissions

Table 13 is an extended version of Table 10 and shows the top 30 embedded documents with unused delegated permissions showing that there is a long tail of such iframes that are used on multiple but not many websites. In connection with the delegation risks described in our contribution, if any embedded company widget is compromised, an attacker could exploit the delegated permissions. The severity varies. Some permissions, such as sensors, may pose low risk, while others, including camera, microphone, display-capture, or payment, are highly sensitive. In particular, similar to supply chain attacks, the scale of potentially affected websites is concerning, especially since some widgets appear to receive permissions they do not actually require.

A.6 Permission Support Across Browsers

Figure 3 presents a screenshot of our tool's results, which evaluate permission support across browser vendors and their versions [11]. We periodically update the results with newer browser versions and monitor source code to expand the list with new permissions. The website also reports changes in browser permission support across versions and tracks default allowlists for each permission.

A.7 Permissions-Policy Header Configuration tools

Figure 4 presents a screenshot of the manual website tool that helps developers configure the `Permissions-Policy` header based on supported permissions [11]. The tool generates a header from the supported permissions list obtained with our previously described method. Additionally, it provides predefined options, such as disabling all permissions or, more commonly, disabling only powerful permissions. Compared to other online tools, our approach guarantees that permissions and generated headers are always up to date, including the latest permissions.

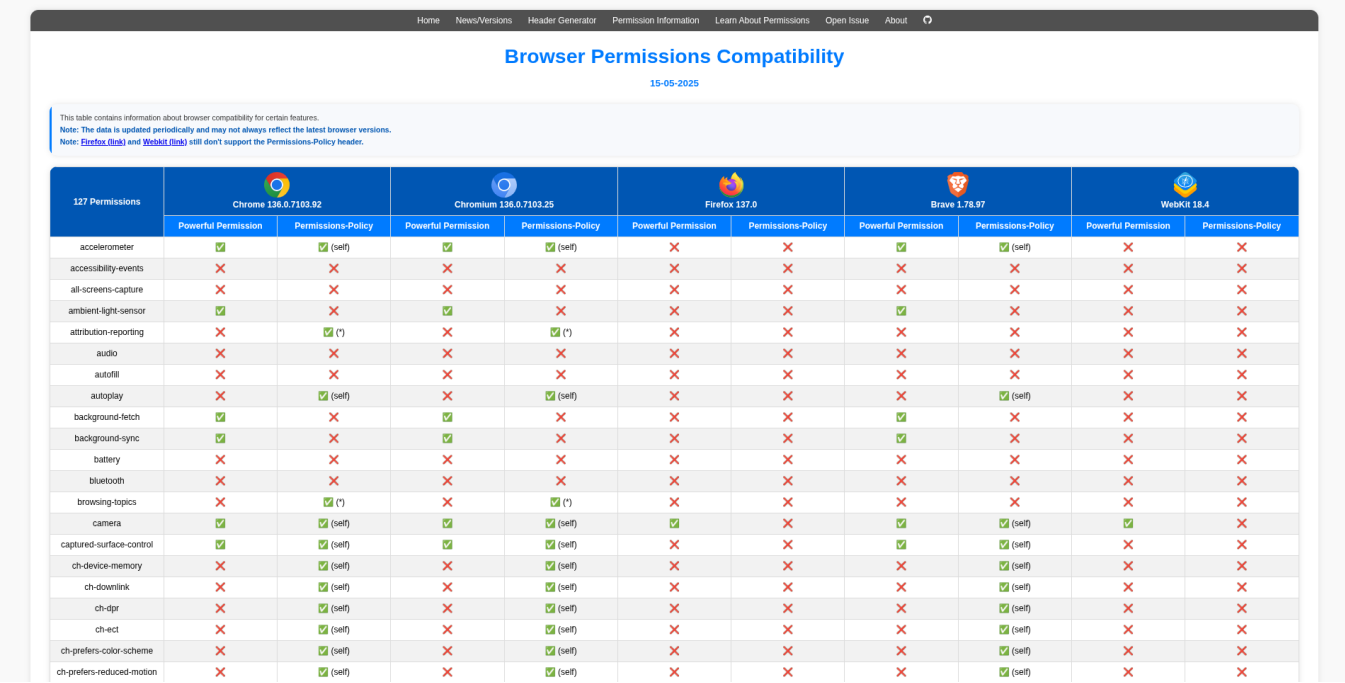

Figure 3: Website showing permission support results.

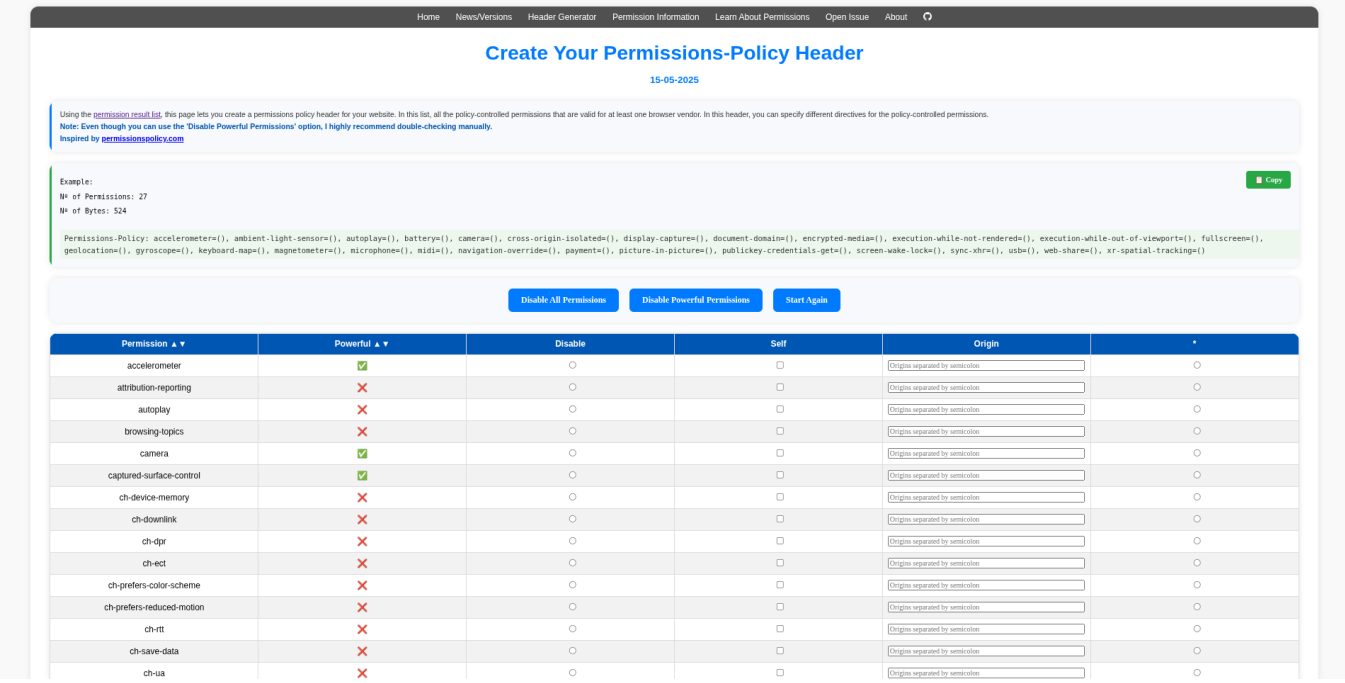

Figure 4: Website for header configuration tool for supported permissions.

Table 13: Top 30 Embedded Documents with Potentially Unused Delegated Permissions

Embedded Iframe	Potentially Unused Permissions	# Affected Websites
youtube.com	*accelerometer, gyroscope*	16,394
livechatinc.com	*microphone, clipboard-read, camera*	13,734
facebook.com	*clipboard-write, web-share, encrypted-media*	1,405
youtube-nocookie.com	*gyroscope, accelerometer*	982
razorpay.com	*payment, clipboard-write, camera*	389
ladesk.com	*microphone, camera*	303
driftt.com	*encrypted-media*	285
wixapps.net	*microphone, camera, geolocation*	246
qualified.com	*microphone, camera*	109
dailymotion.com	*accelerometer, gyroscope, clipboard-write, web-share, encrypted-media*	101
tinypass.com	*payment*	99
imbox.io	*camera, microphone*	93
piano.io	*payment*	92
appspot.com	*camera, microphone, geolocation*	91
facebook.net	*encrypted-media*	81
visitor-analytics.io	*camera, microphone, geolocation*	78
glassix.com	*camera, microphone, display-capture*	76
giosg.com	*camera, microphone, screen-wake-lock, display-capture*	56
cloudflarestream.com	*accelerometer, gyroscope*	55
mediadelivery.net	*accelerometer, gyroscope*	55
socialminer.com	*clipboard-read*	54
infobip.com	*camera, microphone*	46
kenyt.ai	*camera, microphone*	45
vidyard.com	*camera, microphone, clipboard-write, display-capture*	44
jotform.com	*camera, geolocation, microphone*	33
wolkvox.com	*encrypted-media, camera, microphone, geolocation, display-capture, midi*	33
typeform.com	*camera, microphone*	31
mitel.io	*camera, geolocation, microphone*	30
videodelivery.net	*accelerometer, gyroscope*	30
channels.app	*encrypted-media, midi*	30
Total (any iframe)		36,307