Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies

Sebastian Roth*, Timothy Barron†, Stefano Calzavara‡, Nick Nikiforakis†, and Ben Stock*
*CISPA Helmholtz Center for Information Security: {sebastian.roth,stock}@cispa.saarland
† Stony Brook University: {tbarron,nick}@cs.stonybrook.edu
‡ Universita Ca' Foscari Venezia: calzavara@dais.unive.it

***Abstract*—The Content Security Policy (CSP) mechanism was developed as a mitigation against script injection attacks in 2010. In this paper, we leverage the unique vantage point of the Internet Archive to conduct a historical and longitudinal analysis of how CSP deployment has evolved for a set of 10,000 highly ranked domains. In doing so, we document the long-term struggle site operators face when trying to roll out CSP for content restriction and highlight that even seemingly secure whitelists can be bypassed through expired or typo domains. Next to these new insights, we also shed light on the usage of CSP for other use cases, in particular, TLS enforcement and framing control. Here, we find that CSP can be easily deployed to fit those security scenarios, but both lack wide-spread adoption. Specifically, while the underspecified and thus inconsistently implemented X-Frame-Options header is increasingly used on the Web, CSP's well-specified and secure alternative cannot keep up. To understand the reasons behind this, we run a notification campaign and subsequent survey, concluding that operators have often experienced the complexity of CSP (and given up), utterly unaware of the easy-to-deploy components of CSP. Hence, we find the complexity of secure, yet functional content restriction gives CSP a bad reputation, resulting in operators not leveraging its potential to secure a site against the non-original attack vectors.**

I. Introduction

Given the complexity of modern Web applications and the number of different attacks that Web sites and their users can be exposed to, browser vendors have long supported a wide range of additional opt-in security mechanisms (with more being adopted regularly). These mechanisms attempt to protect the confidentiality of cookies (e.g., through the `secure` and `httpOnly` flags), to stop malicious sites from framing benign ones (e.g., through `X-Frame-Options`), or to ensure users are protected against SSL stripping attacks (e.g., through HTTP Strict Transport Security). The Content Security Policy (CSP) is one such mechanism that was introduced in 2010 and was initially designed to grant Web developers more control over the content loaded by their Web sites [40]. Unlike other security mechanisms that have a limited number of configuration options, CSP allows Web developers to author complex, whitelisting-based policies to precisely specify the sources that are trusted for a wide range of content including JavaScript, images, plugins, and fonts.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

Though the (in)effectiveness of CSP has been analyzed and debated in several research papers [6, 8, 50, 51], CSP is still under active development and is routinely adopted by more and more Web sites: the most recent study [8] observed an increase of one order of magnitude in CSP deployment in the wild between 2014 and 2016. Notably though, virtually all papers have focused on CSP as a means to restrict content and have treated its newly added features (such as TLS enforcement and framing control) as side notes. To close this research gap and holistically analyze CSP it is important to take a critical look at how CSP deployment has evolved over time, so as to understand for which purposes developers use CSP and how it affects their sites' security.

To this end, we leverage the Internet Archive to obtain the historical CSP headers for 10,000 highly ranked domains from 2012 to 2018. Leveraging this unique vantage point for a long-term study, we first investigate CSP's evolution for content restriction and specifically highlight numerous case studies that document the struggle in rolling out a CSP. Among the insights, we find that 56% (251/449) of Web sites that test CSP for content restriction with report-only, presumably due to its complexity, refrain from ever enforcing *any* policy. Similarly, we find numerous examples of sites trying to keep on top of their whitelists for months, before eventually giving up. Moreover, we outline the unexpected interactions between CSP and DNS and discover that over 13% of sites attempting to control script resources whitelisted domains that have expired, contain obvious typos, or resolve to private IPs.

Notably though, our longitudinal lens over seven years allows us to show that CSP's primary use case is gradually shifting away from the original goal of content restriction, with 58% (720/1,233) deploying it for other purposes. In particular, we document the increase in adoption of TLS enforcement and how the security mechanism is used in practice for *utility* purposes. Moreover, we find that while CSP is well-equipped to rid the Web of the `X-Frame-Options` (XFO) header, many sites still rely on the deprecated header incapable of providing CSP's fine-grained framing control. To understand the reason behind this lack in adoption, we conduct a notification campaign of sites running XFO. Based on the over 100 responses we received, we find that CSP's full potential is scarcely documented and the perceived complexity of CSP shies operators away from even the easy-to-deploy parts. Furthermore, using an in-depth analysis of violated policies as well as sites that are either trivially insecure or are able to sustain a secure policy, we propose CSP extensions to ease adoption and improve security in practice.

The rest of our paper is structured as follows:

- We leverage the Internet Archive to conduct the longest study of the CSP mechanism to date (2012–2018), and show how our insights can generalize (Section III).
- We document the evolution of CSP and its use cases over time, showing its gradual move away from content restriction to other security goals (Section IV).
- Based on the collected CSPs for content restriction, we discuss an overlooked attack caused by the interplay of CSP and DNS, and document the long-term struggles site operators face in the deployment of CSP (Section V).
- We shed light on the previously dismissed use cases of TLS enforcement (Section VI) and framing control (Section VII), and highlight their growing importance.
- To untangle the reasons behind CSP's failed adoption, we conduct an in-depth analysis of sites that gave up on CSP, ran insecure CSPs, and those that managed to run a strict CSP. Based on this, we identify characteristics which are blocking sites from CSP deployment (Section VIII).
- Given that the adoption of CSP for framing controls lacks behind the underspecified XFO header, we conduct a notification of sites at risk, and report on the insights gathered from the field (Section IX).
- Based on the insights gained throughout our analysis, we highlight how CSP's evolution has affected its prevalence and usage, and propose three actionable steps to improve adoption of CSP (Section X).

II. Background on CSP

Content Security Policy (CSP) is a browser-enforced security mechanism first proposed in 2010 by Stamm et al. [40]. Its primary purpose was to mitigate the impact of Cross-Site Scripting (XSS), as well as of other types of content injection vulnerabilities. However, CSP has undergone several revisions over the years and evolved to support different use cases.

CSPs can be specified inside the `Content-Security-Policy` header or by including them in `<meta>` tags inside an HTML head section. An example CSP is shown in Figure 1 and used as a running example in this section. Policy violations can be logged to the URL specified in the `report-uri/report-to` directive (line 5) Alternatively, a SecurityPolicyViolationEvent can be caught by scripts [25]. CSPs sent using the `Content-Security-Policy-Report-Only` header are not enforced by the browser, but their violations are logged for telemetry. This way, sites can test policies before their deployment, without affecting their functionality.

A. CSP for Content Restriction

CSP provides directives to bind content types to sets of *source expressions*, regular-expression-like constructs used to represent the Web origins from which resources of a given type may be included. For example, the policy in Figure 1 whitelists scripts originating from the origin itself (line 1) and images served from `https://b.com` (line 2). Resources of all the other content types can be loaded from any HTTPS origin, as denoted by the `default-src` directive (line 3) whitelisting the entire HTTPS scheme. If a policy lacks both a t`-src` directive (for some content type t) and a `default-src` directive, content of type t can be loaded from anywhere.

```
1  script-src        'self';
2  img-src           https://b.com;
3  default-src       https:;
4  frame-ancestors   https://*.a.com;
5  report-uri        https://a.com/csp.php;
6  upgrade-insecure-requests;
```

Fig. 1: *Example of a Content Security Policy*

Our example policy also implicitly forbids the use of inline scripts, event handlers, and string-to-code transformations via `eval` and its derivatives. These security restrictions can be relaxed by adding `unsafe-inline` or `unsafe-eval` to the `script-src` directive (or to `default-src` when `script-src` is missing).

The original CSP design proved to be an inflexible security mechanism, since it required the removal of all the inline scripts and event handlers to actually provide any security benefit; hence, its adoption initially lagged behind other security headers [51]. To remedy this, the second version of CSP introduced *hashes* and *nonces* to whitelist selected inline script elements [46]. The former allow to whitelist inline scripts matching a given hash, whereas nonces enable execution of `<script>` tags carrying this nonce. However, note that neither hashes nor nonces allow developers to whitelist event handlers. For backward compatibility, when nonces or hashes are present, supporting browsers ignore `unsafe-inline`.

Since 2016, the third and latest version of CSP further aims to ease deployment through the `strict-dynamic` source expression, used to propagate trust to scripts loaded by other scripts whitelisted using a valid hash/nonce [47]. In other words, when `strict-dynamic` is used in combination with hashes/nonces, any script with a valid hash/nonce can arbitrarily include additional scripts by programmatically adding script nodes to the DOM, even when they lack a valid hash/nonce.

B. CSP for TLS Enforcement

A use case not originally intended for CSP is *TLS enforcement*. While one could already leverage the content restriction capabilities of CSP to force the use of HTTPS by whitelisting just HTTPS resources, CSP also introduced new explicit directives for additional expressiveness. In particular, CSP was extended to better integrate with the Mixed Content specification, a security policy implemented by all modern browsers which regulates the use of HTTP resources in HTTPS pages [53]. In particular, modern browsers will refuse to load active content like scripts over such downgraded connections, but, e.g., for images blocking is at the browser's discretion. A site can be explicit in forbidding such resources by specifying the `block-all-mixed-content` directive in its CSP. This effectively instructs browsers to block *all* mixed content. Instead of blocking all mixed content, the `upgrade-insecure-requests` directive (shown in Figure 1, line 6) forces the automatic rewriting of all HTTP URLs to HTTPS upon page loading. This is useful to gracefully implement a transition from HTTP to HTTPS, while preventing warnings and breakage due to the use of mixed content.

C. CSP for Framing Control

Moreover, CSP introduced the `frame-ancestors` directive for *framing control*, thus providing protection against

framing-based attacks like Click-Jacking [30]. It was intended to replace the traditional `X-Frame-Options` header (XFO) with a more flexible whitelisting mechanism, as well as to patch underspecified points of XFO [36]. In particular, XFO just supports three types of policies: `DENY` prevents all framing, `SAMEORIGIN` restricts framing to the same origin as the framed page, and `ALLOW-FROM` u restricts framing to a *single* URL u. Note that `ALLOW-FROM` is not even supported by all browsers (in particular Chrome and Safari), thus making XFO particularly problematic [22]. CSP, instead, may be used to allow framing from a list of arbitrarily many Web origins specified through source expressions. Moreover, the CSP specification mandates that the security enforcement must be done against the *full chain* of ancestors of the framed page. In contrast, XFO delegates the choice to browser vendors. This enables *double framing* attacks on browsers which only compare the origin against the top frame, such as IE and Edge [30]. As an example of the benefits of CSP over XFO in terms of framing control, the policy in Figure 1 allows framing by any subdomain of `a.com` over HTTPS (line 4), which cannot be expressed using XFO. Moreover, it ensures that in case of multiple nested frames, all frames must originate from subdomains of `a.com` in order for the content to be shown.

III. Analysis Methodology

Contrary to prior work that mostly quantified the insecurity of CSP with respect to XSS mitigation, our main focus is to investigate how CSP's changing role affected its deployment. To this end, the Internet Archive (IA) [17] provides a glimpse into the past since, starting from 1996, it archives both content and headers of popular Web sites and makes the information publicly available. Here, we describe how we arrive at a representative dataset of 10,000 popular sites, followed by how we crawl the IA for data related to CSP's evolution.

A. Dataset Construction

Since widespread support for CSP landed in browsers in 2012 [9], we focus on analyzing the deployment and usage of CSP from 2012 to the end of 2018. However, merely using a current list of highly-ranked domains does not suffice, as many domains may not have been registered (let alone be popular) during the entire period of our study. Moreover, the reliability of lists such as Alexa was recently challenged for exhibiting massive fluctuations on a day-to-day basis [37]. Therefore, we used the following selection strategy for sites: using Alexa lists collected over time [37], for each month between January 2012 and December 2018 we extracted the set of sites that were in the top 50,000 on at least a single day within that month. We then calculated the intersection of these monthly sets, sorting the sites by their average rank (over the entire time).

To limit the overall number of queries to the IA, we opted to study the 10,000 most prolific sites for the given timeframe. To arrive at this set [2], for each entry in the sorted list of sites, we queried the IA to determine if it contained any archived version of the site. This check is necessary since the IA removes any previously stored content when a site blocks the IA crawler through `robots.txt` [20, 41]. To understand potential biases towards any period within the seven years, for each month we computed the average top 10,000 sites, and compared them to the list we finally chose. This showed that at least 4,960 and at most 6,199 entries overlapped for any given month, averaging at 5,738 domains. Hence, we argue our dataset is well-balanced, allowing us to draw general conclusions on the evolution of deployed CSPs over time.

Given this set of sites, we queried the IA for their daily snapshots from January 1, 2012, until December 31, 2018. Selecting one snapshot per day (where available), we followed any HTTP redirection to reach the final landing page for users accessing the site at archival time. Note that we only used HEAD requests to reduce the load on the IA. Once all redirects were followed, we compared the originally visited site to the final URL. In total, we queried the IA for 10,414,975 snapshots, out of which 10,316,325 yielded final URLs still on the same site. For each of the same-site snapshots, we collected all variants of CSP headers historically used (i.e., `Content-Security-Policy`, `X-Content-Security-Policy`, `X-WebKit-CSP`, and `Content-Security-Policy-Report-Only`), as well as XFO and `Strict-Transport-Security`. The IA prefixes archived headers with `X- Archive-Orig-`, making them easy to differentiate from other response headers [41]. Note that the IA removes CSP directives in `<meta>` elements to avoid interference with the CSP of the IA itself. The CSP deployment occurred after previous research had identified the possibility of maliciously tampering with historical results through the use of externally-hosted scripts [20]. This filtering has a minor effect on our experiments since few sites deploy CSP through a `meta` tag (Section III-B).

Naturally, not all sites have daily snapshots in the IA. On average, a site had snapshots for 1,031 of the 2,557 days in our analysis timeframe. Hence, whenever there is no snapshot for a given day d_i, we use d_{i-1} instead as a basis for our analysis, in a recursive fashion. This means that for every gap in the snapshot data, we use the last entry before that gap as the data point. This approach suffers from a certain level of imprecision, as it might be unclear exactly when a change in a CSP has occurred. This loss of fine-grained information, however, does not significantly affect the class of observations pursued in our work. For each collected CSP policy, we normalized all randomized elements. In particular, nonces are meant to used just once, and the violation reporting URL may also contain random strings. For those, we removed the random parts, allowing us to properly analyze actual changes. All normalized policies are available at **https://archive-csp.github.io**.

B. Threats to Validity

Given that our analysis uses the IA to extract information, it is prone to the following threats to the validity of the results.

1) Incorrect Archival Data: It is not clear to what extent the data collection process in the IA might influence our results, since a specific browser version might yield a different server response. To determine this IA-specific influence, we chose a second archive service to corroborate the IA's data. In particular, Common Crawl (CC) [10] has been collecting snapshots of popular sites since 2013. For each date on which we found a CSP in the IA, we queried the CC API for a matching snapshot. Overall, we found 38,129 overlapping snapshots for 940 sites. Out of these, 729 (1.9%) on 127 sites were inconsistent between the two archives. For 96 cases the

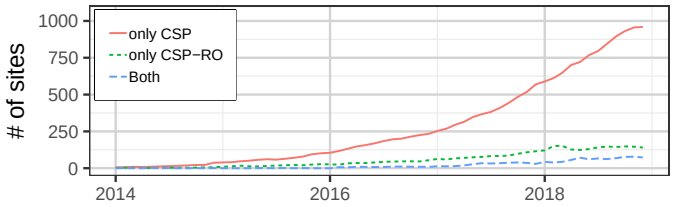

Fig. 2: *Usage of CSP only, CSP-RO only, or both*

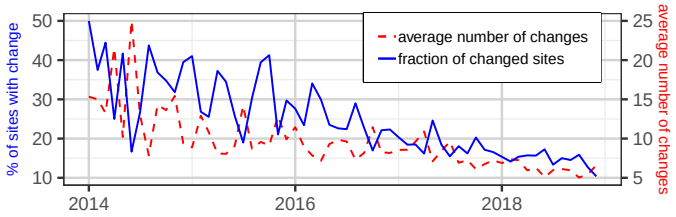

Fig. 3: *Maintenance of CSPs over time*

difference was the lack of `block-all-mixed-content` or `upgrade-insecure-requests` in the CC data. Further investigation showed that in the IA, these directives were separated from the remaining CSP with a comma instead of a semicolon. This likely relates to the IA joining headers with the same name with a comma. For those pages, we could always only find a *single* CSP header in the CC response. Moreover, starting from August 2018, these sites still used the aforementioned directives in the IA data, but CC returned two CSP headers (one including only those directives). Hence, we speculate this relates to a bug in CC, which was fixed around August 2018. 23 cases showed evidence for a difference in crawling time; e.g., taking the IA policy from the following day matched the CC. Additionally, 29 differences can be attributed to whitelisting different edge CDNs based on the crawler's IP. For the remaining 581 cases, the exact cause of the difference was not detectable. Notably though, in only 238/38,129 cases (0.6%) did those policies have a different use case (see Section II). Overall, this confirmation from a second source gives us high confidence in our utilized dataset.

2) CSP Through Meta Tags: As mentioned, CSP can also be deployed via HTML meta tags which are currently removed by the IA. To understand the potential impact of this drawback, we crawled the live main page of all 10,000 Web sites from our dataset on June 10, 2019. We collected CSP headers and also checked the content for CSP meta elements. In this experiment, a total of 1,206 sites deployed CSP, and, of those, 78 (6.4%) sites set their policy *only* through a meta element. Of the 1,147 sites that sent a CSP header, 19 *also* set the meta element. Notably though, only 3 with both meta and HTTP header CSPs had policies which differed in their use case (see Section IV-B). Hence, we are confident the archived headers provide a valid dataset for our large-scale historical analysis.

IV. Historical Evolution of CSP

In this section, we provide an overview of how CSP deployment evolved, studying the adoption and maintenance of CSPs, and the changing use cases we observed over time.

A. Adoption and Maintenance of CSP

Figure 2 shows the number of Web sites utilizing CSP in enforcement mode, in report-only mode (CSP-RO), and in both modes in parallel. The figure does not include the CSP adoption from 2012 to 2014, since only 8 different sites deployed CSP before 2014. In our dataset, `lastpass.com` and `adblockplus.org` were the first Web sites to adopt CSP in January 2012, while the other six sites joined in 2013. In total, we find that 1,233 out of the 10,000 sites in our dataset used CSP in enforcement mode for *at least a single day* in our analysis period. Notably though, in the last month of our analysis, only 1,032 domains enforced a CSP.

We draw two main observations based on the plot. First, even though CSP offers the report-only mode to enable developers to experiment with policies before deployment, this mode is not nearly as popular as the enforcement mode. This means that most developers are rolling out policies without having a chance to test them on their user base. We suspect that this is one of the main reasons why CSPs in the wild are so relaxed since they have not been appropriately evaluated and the developers eventually opted for utility over security (see Section V). Second, we observe even fewer Web sites utilizing CSP in enforcement and report-only mode at the same time, which is the preferred way of gradually testing and deploying more restrictive policies. These two observations together suggest that developers are likely confused about the role of report-only and therefore do not take advantage of it.

The plot also shows that the overall adoption of CSP is consistently increasing over time. Given that our list of Web sites remains stable, we can attribute the increased CSP adoption to Web developers deciding to use it, rather than CSP-capable Web sites suddenly climbing in Alexa popularity. Specifically, in 2017 and 2018 anywhere between 18 and 65 Web sites in our dataset were adopting CSP in enforcement mode for the first time every single month. Considering the ever-increasing complexity of Web sites and their deployed JavaScript code [41], the rising adoption of CSP seems to be a positive sign for security. Given the low adoption of report-only, the next sections focus primarily on enforcement mode.

Since CSP is one of many security mechanisms deployed by servers and enforced by browsers (like, e.g., HSTS [29]) one may think that similar to other mechanisms, once a policy is curated, that policy can be deployed and used for a prolonged period. Unfortunately, CSP — especially for its original use case of script content restriction — is way more complicated than other security mechanisms and requires constant maintenance to ensure that the appropriate sources are whitelisted so that the site remains operational and secure. Figure 3 quantifies the burden of keeping deployed CSPs up to date. The dashed red line shows that, in many cases, sites needed to modify their policies tens of times each month. Even though we see the average number of changes going down towards a steady state, we later show that this is rather caused by CSP being used for non-traditional reasons (such as for TLS enforcement) than by stabilized whitelists. The blue line shows the fraction of sites with changed policies. We still observe the need to regularly maintain CSPs since by the end of our analysis, still roughly 10% of sites changed their policy at least monthly.

B. Use Cases for CSP

Even though CSP was initially developed as a measure to mitigate the impact of script injection, the multitude of directives available nowadays allows site operators to control much

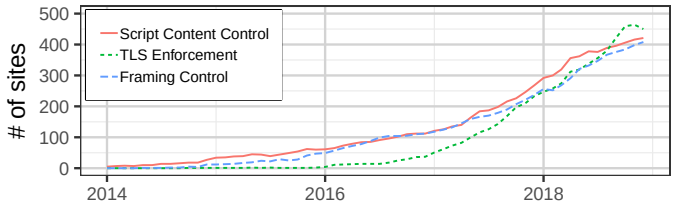

Fig. 4: *Classified policies over time*

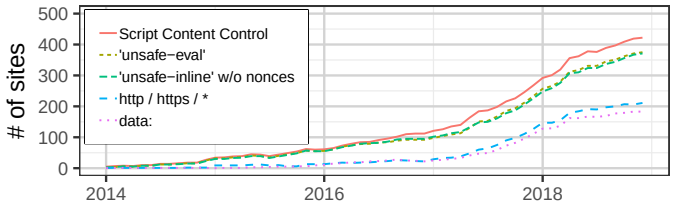

Fig. 5: *Overall adoption of content restriction and insecure practices*

more than included content. Specifically, we classify policies in the following (overlapping) categories: **Script Content Restriction** (policies using `script-src` or `default-src`); **TLS Enforcement** (policies using `upgrade-insecure-requests`, `block-all-mixed-content`, or whitelisting only HTTPS sources); and **Framing Control** (policies using `frame-ancestors`). Figure 4 shows the number of sites applying CSP for the identified use cases over time. When comparing the numbers to Figure 2, we find that the increase in the deployment of CSP starting from 2015 coincides with the increased usage of framing control. Similarly, the increase in the overall usage of CSP from 2017 onwards aligns with the increased enforcement of TLS connections, mostly through `upgrade-insecure-requests`. Moreover, the decrease in maintenance shown in Figure 3 is evidence of easily deployable mechanisms like TLS enforcement, rather than reduced effort to keep policies up to date. This clearly indicates that while CSP was meant as a tool to mitigate script injection, new additions to the set of the CSP directives have shifted CSP into new use cases. In the following, we analyze each category separately, discuss its evolution, and highlight key insights.

V. CSP for Script Content Restriction

In this section, we analyze how CSP has evolved with respect to its content restriction capabilities. This not only allows us to confirm findings of prior work through the longitudinal lense of the IA, but also to highlight unknown trends in the increasing trust of operators into lower-ranked domains, to investigate the success of newly introduced CSP features, and to identify previously unexplored attacks related to hijacking whitelisted domains. Finally, given the unique vantage point of an archival analysis, we conduct a number of case studies which document the long-lasting struggle of Web sites to deploy an effective policy.

The first observation we make is that, out of the 1,032 sites in the dataset that enforced a CSP by the end of our analysis period, only 421 sites shipped policies aimed at restricting script content. This clearly shows that although CSP was initially meant to mitigate script injections, this is only attempted by about 41% of the deployed policies. We now present insights gained from the deployed policies.

A. Insecure Practices Die Hard

Figure 5 shows the evolution in the number of sites using CSP for content restriction, and how many of them have been using various unsafe practices therein. The two most popular unsafe practices are the use of `unsafe-inline` (without the use of hashes/nonces) and the use of `unsafe-eval`. While `unsafe-eval` must be considered the lesser evil, given that its presence does not immediately nullify CSP's security, we observe almost all the policies deployed in 2014 and 2015 made use of `unsafe-inline`. We attribute this to the inflexibility of early versions of CSP. However, it is noteworthy that even at the end of our analysis period, this trivially insecure directive is contained in almost 90% of the policies. While we can only speculate about the exact reason for this trend, we opine that it is likely due to event handlers which cannot be whitelisted with hashes or nonces. Checking merely the start pages for the 378 sites which deployed `unsafe-inline` in December 2018, 180 (48%) of them carried event handlers. The actual number of sites making use of them is likely even higher, but we could not confirm this without adding a significant load on the IA by crawling sub-pages. The bottom two lines of Figure 5 refer to the whitelisting of entire schemes. In particular, the first line shows that developers are declaring that any HTTP/HTTPS origin is permitted, which obviously voids security. The second line represents the whitelisting of the `data` scheme, which can be used to add arbitrary code, e.g., through `data:;alert(1)` [23].

Our analysis also indicates that the numerous features added to CSP to ease its secure deployment are not successful. Table I reports on the adoption of hashes, nonces, and `strict-dynamic` on a yearly basis. Note that, although `strict-dynamic` has recently been shown to be bypassable through *Script Gadgets* [19], we still treat it as an improvement since it should ease CSP deployment. The table highlights that while the usage of CSP to control scripts has constantly grown, neither hashes nor nonces have gained significant adoption. We also find that in both 2017 and 2018, at most 8 sites made use of `strict-dynamic`. While we are only checking start pages and might therefore miss wider-spread deployment, this still highlights that the new directive is not widely used. Overall, we can conclude that insecure practices are present in 90% of policies, whereas secure practices like nonces or hashes, reach less than a 5% adoption rate.

B. Whitelisted Sources

We now complement the findings of Weichselbaum et al. [50] regarding the insecurity of whitelists by quantifying the evolution of the number of whitelisted script sources over time (Figure 6). We observe that even though the median remains relatively stable, the number and magnitude of outliers expand year after year with some Web sites whitelisting over 200

Year	Controls script	Hashes	Nonces	`strict-dynamic`
2014	27	0 (0%)	1 (4%)	0 (0%)
2015	75	1 (1%)	2 (3%)	0 (0%)
2016	135	1 (1%)	3 (2%)	0 (0%)
2017	296	4 (1%)	14 (5%)	7 (2%)
2018	478	6 (1%)	24 (5%)	8 (1%)

TABLE I: *Number of sites per year restricting script content, and using hashes, nonces, and* `strict-dynamic`*.*

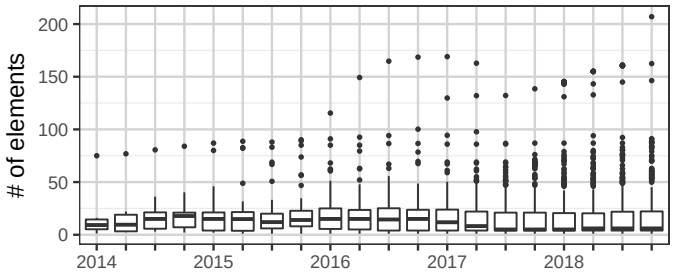

Fig. 6: *Boxplot showing the number of elements in script whitelist*

unique sources for their scripts. The low median must be interpreted in light of the unsafe practices described earlier. Whitelisting an entire scheme (such as `https`) and allowing `unsafe-inline` may result in short policies (in terms of the number of entries) which are, however, still more vulnerable than *explicitly* trusting hundreds of remote third parties.

The ranking of whitelisted sources is an important dimension of CSPs, since site popularity is often used as a proxy for security. This stems from the reasonable assumption that, on average, the developers of more popular Web sites have more know-how and resources to help secure their code. For example, Van Goethem et al. [45] discovered that more popular Web sites tend to utilize more security mechanisms than less popular ones. To analyze this, Figure 7 shows information about the ranking of Web sites that are whitelisted as script sources. In particular, this contains all hosts that were whitelisted; i.e., even in the presence of $\star$, we analyzed the remaining contained sites. We argue that this is useful, given that remote sources contained in the whitelist are *explicitly trusted* by the site, and the existence of $\star$ often is a byproduct of attempting to curate a limited whitelist (cf. Section V-D1). For this analysis, we used the publicly available historical dataset of Scheitle et al. [37], extracting the rank from each site on the day when it was whitelisted. To combat the fluctuation of these lists, we aggregate the results on a monthly basis. We find that starting from 2017, the average rank of trusted CSP sources increases, although the average number of elements in whitelists does not (cf. Figure 6). This means that developers are explicitly trusting less popular Web sites through their CSPs to host JavaScript code, thereby weakening their security.

C. Abusing Whitelisted Domains

Given the observed trend in trusting lower-ranked sites, we find that such domains are valuable targets for an attacker. Even though whitelisting domains can prevent script injections from arbitrary sources, this can be bypassed by an attacker who is able to serve content from a whitelisted source. Specifically, referring to trusted sources by domain names carries with it the typical security problems of domain names. In our dataset, the CSPs of 373/422 (89%) of sites trying to restrict content whitelisted at least one domain for script inclusion. We discuss three attacks to leverage trusted domains and determine how many sites could have had their policies bypassed.

1) Expired Domains: First, domains that expire while still appearing in a whitelist may be re-registered, giving the new owner the ability to serve malicious content through a source that is already trusted by the victim site. To find periods when whitelisted domains were expired, we use public TLD zone files which we have collected daily since 2016. We compute

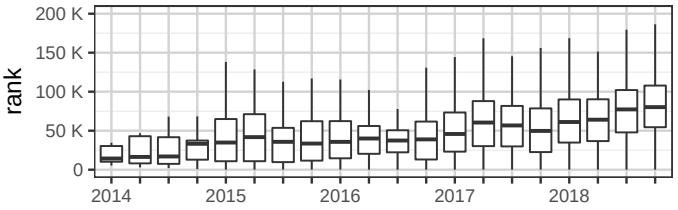

Fig. 7: *Average historic Alexa rank of whitelisted domains*

the difference between zone files on consecutive days to find domains that are newly registered or expired. These differences are indexed into a search engine allowing us to query for the registration history of a domain name. Despite the lack of zone files covering the full duration of our dataset, we were still able to find 9 cases of domains expiring after being added to a whitelist and 7 domains which were not registered at all during the periods they appeared in CSP whitelists.

2) Typos: CSP whitelists are also vulnerable to human error in defining trusted sources. A typo entered in the whitelist permits the loading of content from an unintended domain. The typo domain may not be registered, in which case it can be bought and abused as above. If the typo domain is already owned, there is still the potential for the current owner to discover the issue and abuse the misplaced trust. Compared to traditional typosquatting where each end-user needs to mistype the domain, typos in CSPs involve a one-time mistake which results in a persistent threat. To find typos we use the fat-finger typosquatting models of Wang et al. [48] to generate a list of domains that may have been the intended ones. We compare the Alexa ranks of these domains to the whitelisted domain to find cases where the ranks differ significantly, implying the site owner may have meant to type the more popular domain. We manually inspected each of these cases taking into account the length of the domain, whether the domains are whitelisted by other sites, the type of content served on both domains, and whether the domain is used elsewhere on the page. In doing so, we found 11 domains which appear to be typos on the CSP whitelists of 11 different Web sites in our dataset.

3) Local Addresses: Finally, even if the attacker cannot gain control of the whitelisted domain, in some cases they may still exploit the IP address it points to. If this address is in a private IP range, an attacker can serve malicious content to victims on the same LAN [28]. To find such cases, we queried Farsight's DNSDB to determine what the whitelisted domains resolved to in the past. We searched these records for private IP addresses and checked that the observations occurred during a period where the domain was present on a CSP whitelist. This revealed 15 whitelisted domains trusted by 26 sites.

Table II summarizes our findings from each condition as well as the overall impact on the sites using CSP. In addition, it shows examples which were vulnerable to abuse and the sites which could have been affected, notably including Dropbox, a site with an otherwise exemplary CSP record. As the table indicates, 50/373 (13%) sites using CSP whitelists trusted at least one abusable domain. Unlike other more obvious CSP mistakes, avoiding these requires carefully vetting the domains and regularly updating the list. This is not an easy task, as a surprising percentage of popular sites fail at some point and the maintenance burden is even higher for sites that whitelist hundreds of domains. Because of this non-obvious interaction

Category	Vulnerable domains	Duration	Impacted domains
Expired	16		15
Example	*sushissl.com*	39 days	*zomato.com*
Typo	11		11
Example	*optmster.com*	7 months	*experian.com*
Local address	15		26
Example	*marketo.net*	3 months	*dropbox.com*
Total	41		50

TABLE II: *Vulnerable whitelisted domains and the number of sites that allowed these domains in their whitelists. One example for each category with a high-profile site that included it and duration of attack opportunity.*

between CSP and DNS, even a perfectly secure CSP could become insecure without changes to the policy.

D. Longitudinal Case Studies

Contrary to other works which focused on analyzing CSP's adoption and security from a bird's eye viewpoint and at most over a period of a few months, our archival analysis enables us to investigate how sites struggled in their deployment of content-restricting CSPs over the years. In the following, we present our insights into sites that exemplify general behavior and issues we identified in our long-term study.

1) Curating Limited Whitelists: To understand how popular Web sites attempt to curate their whitelist, we use `airbnb.com` as a motivating example. The site first deployed a CSP report-only (CSP-RO) header on November 12, 2014. While initially 17 sites were whitelisted (and therefore allowed) to serve script resources, by March 27, 2015, the list of whitelisted sites had grown to 21. On the next day, however, a new policy was deployed (still in report-only mode), whitelisting the full HTTPS scheme; thereby essentially allowing any site to provide script content to Airbnb. In addition, we observed multiple changes to whitelisted hashes, indicating that due to changes of inline scripts, these had to be repeatedly modified. The policy was further modified until May 1, 2015, when the curated policy was deployed as an enforcement CSP. This policy was modified three more times until May 9, 2015, when CSP was fully disabled.

The policy was re-enabled on May 22, 2015, and from then up to December 8, 2017, we observed 222 changes to the deployed CSP, merely adding and removing hashes of whitelisted scripts, while still allowing any HTTPS host to serve scripts (i.e., not actually mitigating the risk of content injection at all). On December 8, 2017, `airbnb.com` started experimenting with changing CSP-RO headers, limiting the sites from which remote scripts could be loaded. The first variant of a blocking CSP with limited sites was deployed on January 13, 2018. However, on January 16, 2018, the policy regressed to trusting all HTTPS hosts for scripts. After another 29 modifications to CSP-RO headers, Airbnb on March 13, 2018, finally deployed a blocking CSP with limited hosts, and a different report-only policy until the end of our analysis timeframe. By then, while only two more sites were added to the enforced list, we observed almost daily changes to the policy, ensuring up-to-date hashes of the used inline scripts.

For both Airbnb and other domains exhibiting the same struggle, we attempted to attribute changes in the CSP to changes in the page itself, so as to understand if they had been caused through the site itself adding new scripts, or third-party code (such as ads) loading additional dependencies. However, apart from obvious changes in hashes due to changed inline scripts, we could only attribute a negligible fraction of changes in the whitelisted sites to modified content of the start page (10/106 for Airbnb). For Airbnb, we investigated further, crawling one level of links for each of the changes we could not attribute. Doing so, we were able to attribute an additional 15 changes, but at the cost of making 106,090 GET requests to the Internet Archive. Given this small increase in coverage and our goal of not flooding the IA with requests, we decided to not follow this path of attribution any further.

Even though Airbnb was an early adopter of CSP and made significant efforts to secure their site, they had to often effectively disable their policies for long periods of time. These gaps could have been abused by attackers to launch attacks that would not normally be possible with Airbnb's CSP. Moreover, while they finally managed to curate a whitelist of 33 sites, it took them multiple years from when they started experimenting with CSP to arrive at this final policy.

2) Eventually Giving Up on CSP: Next to those sites which deploy CSP in an insecure variant by allowing any origin to provide script code, our dataset also contains numerous sites which tried to deploy CSP but eventually gave up. Overall, 294 sites used CSP in either mode for at least one day but not within the last month of our analysis. Of those sites that used CSP for at least one month, we find that 227 eventually gave up. Lastly, considering only those sites that made a long-term attempt at CSP (at least one year), we find a total of 79 sites which ended their deployment before December 1, 2018.

`Researchgate.net` was a prominent example of a Web site which moved from CSP to CSP-RO on September 13, 2018, before completely abandoning it on November 27, 2018. Before moving to report-only mode, we observed 29 different attempts at enforcing a working policy. This indicates that `researchgate.net` struggled to build a functional, yet secure policy. In a similar case, `zaycev.net` had deployed CSP since September 2015 and stopped deployment on April 11, 2018. Checking snapshots from March and April, the site itself appeared to be unchanged. Notably though, in the week before CSP was disabled, we observed 3 changes, seemingly to get the non-functioning CSP to stop interfering with the Web site. These examples make it clear that even though sites spend significant effort to deploy CSP, many of them eventually yield to the complexity of keeping their policies secure *and* functional, and proceed to entirely abandon the mechanism.

3) Failing Gracefully Through Report-Only: While prior works have found evidence that CSP report-only is used to test and then deploy policies, our longitudinal allows us to investigate the usage of report-only in a more thorough fashion. In particular, we can identify all those sites attempted report-only within the five year period for which have extracted CSP data and understand how many of them actually made use of CSP report-only to test and then roll out policies. Even though report-only mode is meant to allow "developers to piece together their security policy in an iterative fashion" [47], it can also be used to determine that the deployment of CSP would cause too much interference with the operation of the application and should therefore not be rolled out; importantly before breaking the application due to blocked resources.

CSP-RO for Content	CSP for Anything	CSP for Content	CSP for TLS	CSP for Framing
449	216 (48%)	130 (29%)	78 (17%)	23 (5%)

TABLE III: *From report-only to enforced policies*

In total, our dataset shows that 449 sites experimented with report-only aimed at restricting content at some point throughout our analysis period. Of those, 233 never actually enforced any type of CSP afterward. This already indicates that CSP's complexity in restricting content deters developers to deploy it after seeing the results in report-only. Next, we analyze to what extent the remaining 216 sites leveraged the tested content-restricting policy in enforcement mode.

Table III shows the overview of our results. Of the 216 sites that enforced any type of CSP after testing a policy aimed at restricting content, only 130 end up deploying such a policy. The remaining 86 decided to not use CSP for its original purpose, but rather for TLS enforcement (78) and framing control (23), whereas 15 used it for both. Of those 86 sites, 79 had also tested report-only for framing or TLS control before, showing that they merely dropped the content restriction part. Two of the most prominent examples are `aliexpress.com` and `aarp.org`. AliExpress started experimenting with CSP report-only in September 2015, eventually allowing up to 35 sites as well as the usage of inline scripts and eval. In June 2016, they moved to a report-only policy permitting any origin to provide content, but later only enforced an `upgrade-insecure-requests` policy. `aarp.org` went further than this, trying to remove `unsafe-inline` and `unsafe-eval` from their policy altogether. They experimented with this for a month in November 2017 but re-enabled the insecure directives in December 2017. While this policy was still active for report-only in December 2018, the site instead adopted an enforced CSP merely used for framing control.

Overall, our results indicate that more than half of the sites which experiment with any sort of report-only policy never actually deploy one in enforcement mode. This provides clear evidence that the complexity of the mechanism is too high for many site operators. Notably though, we find that even when sites first deploy report-only to restrict content and subsequently move to an enforced policy, only 60% (130/216) keep their content restriction in place, whereas the remaining policies use CSP for the additional use cases, i.e., TLS enforcement or framing control. This shows that such sites could *fail gracefully*, in the sense of understanding that CSP for content restriction would break their site without causing any outages to an enforced policy. Overall, only 29% of all sites that tried script content policies via report-only ended up enforcing such a policy, again highlighting CSP's complexity.

4) Lasting Success in CSP Deployment: We now turn to sites with a long-standing track record of improving their CSP deployment. Three prime examples are `pinterest.com`, `github.com`, and `flickr.com`. Pinterest first deployed a CSP-RO in February 2014 and tested it with an increasing number of whitelisted hosts until July 2014. At that point, they deployed a policy with 15 whitelisted sites but still had to resort to `unsafe-inline`. The number of whitelisted sites went up to 23 sites until May 2017, at which point they deployed nonces and `strict-dynamic` (making modern browsers ignore both `unsafe-inline` and whitelisted hosts). Notably, even though nonces and `strict-dynamic` remained in the policy until the end of our experiment, the list of whitelisted hosts still grew over time. This, however, is because legacy browsers would otherwise block content.

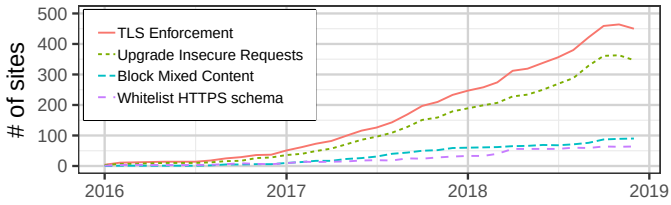

Fig. 8: *TLS Enforcement Strategies*

When Github first deployed CSP in November 2013, their whitelist contained 5 sites, including a CDN of their hosting provider Fastly and Google Analytics. In October 2014, however, Github stopped using Google Analytics and moved all their assets to two of their own subdomains. By October 2015, those two subdomains were explicitly whitelisted; but since that time, Github consolidated all scripting resources on `assets-cdn.github.com`, which has remained the lone entry in `script-src`. Notably, Github has never used `unsafe-inline`. This behavior is in line with Github's efforts for securing their service [14, 43].

Our last example, Flickr, first used CSP in enforcement mode in October 2015. While they had a limited host whitelist, they also had to employ `unsafe-inline`. The initial policy's whitelisted sites grew to 12 by May 2017 at which point Flickr started experimenting with nonces (in report-only mode). After enabling nonces in their enforcement policy in July 2017, they had to go back and forth with nonces three times until finally, since August 2017, they have always deployed nonces. In March 2018, Flickr first sent a report-only policy with `strict-dynamic`. This feature, however, while still active in CSP-RO had not been ported to their enforced CSP by the end of our analysis timeframe.

All three sites have shown a long-lasting commitment to improving security with CSP. These sites, however, are objectively major players on the Web with the ability to spend considerable time and resources on deploying appropriate policies. Even then, while Github is an excellent example of how to deploy CSP in a fully secure fashion, the other two generally positive examples have shown how complicated a deployment can be, especially with regard to removing the need for `unsafe-inline`. What's more, for legacy compatibility (as, e.g., Safari and Edge don't support `strict-dynamic` [27]), even sites with modern features like `strict-dynamic` need to constantly update their host-based whitelist, meaning CSP remains a long-term maintenance burden.

VI. CSP for TLS Enforcement

The second use case of CSP is to ensure that no content is loaded via HTTP on HTTPS sites. In this section, we first report on how the different means of achieving that goal have evolved over the course of time, and then report specifically on how they were used successfully for TLS migration.

A. Evolution of TLS Enforcement

Figure 8 reports on the trend of deployed CSPs for TLS enforcement. As we first saw policies specifically tailored towards TLS enforcement in 2016, we omit the data before 2016. We observe a steady increase in the usage of CSP for TLS enforcement from December 2016 onwards. Most notably, the enforcement is not pushed forward by exclusively whitelisting HTTPS sites, but rather by the `upgrade-insecure-requests` directive. Full support for this CSP feature first landed in browsers around March 2015 (with Chrome 43 [52]), meaning site operators did not deploy the directive until about 1.5 years after it became available. The related directive to block all mixed content has also increased in usage, but only sees adoption on a fraction of sites compared to `upgrade-insecure-requests`. Notably, we found that 51 sites make use of both directives in December 2018, even though in modern browsers `block-all-mixed-content` in the presence of `upgrade-insecure-requests` is effectively a no-op because the upgrade to HTTPS happens first. We find the decreasing number of sites deploying CSP for TLS enforcement in December 2018 is not a downward trend; a brief check for January 2019 shows that usage keeps increasing.

Figure 8 also compares the usage of a policy aimed at ensuring that no content can be loaded over HTTP with the usage of the HTTP Strict Transport Security (HSTS) header on those sites. The HSTS header is set to ensure that once a site has been visited over HTTPS, it cannot be loaded over HTTP until a specified timeout occured [16]. In contrast, the CSP directives ensure that any resources that are included by the original site will be loaded over HTTPS. We observe that alongside the increasing deployment of `upgrade-insecure-requests`, the fraction of sites using HSTS also rises. Notably, of the 450 sites that enforced transport security with CSP in December 2018, 267 made use of HSTS (59%). This uptake in HSTS adoption is a positive trend in line with the more widespread use of HTTPS on the Web [35].

Considering the 347 of those sites enforcing TLS explicitly via `upgrade-insecure-requests`, 194 made use of HSTS. This provides us with an interesting insight: the `upgrade-insecure-requests` directive is deployed to ensure that once the connection has been securely established, no resources or other URLs can be accidentally loaded via HTTP. HSTS, in contrast, is used to ensure that the site cannot be loaded via HTTP in the first place. While for a secure setup, both mechanisms are desirable, deploying HSTS comes with a loss of control for the operator. Once a client has observed the HSTS header, it will refuse to connect to the site via HTTP for a set amount of time controlled by the received HSTS header. Given that activating HSTS is no more complex than shipping `upgrade-insecure-requests` we posit that its absence by half of the sites using the directive has more to do with the developers being uncomfortable fully giving up HTTP than with issues regarding HSTS setup and deployment.

B. Leveraging CSP for TLS Migration

While one of the goals of CSP is to *enforce* TLS, it also provides a meaningful aid for developers when migrating to HTTPS. When mixed active content is detected by the browser while visiting an HTTPS site, it is automatically blocked. However, `upgrade-insecure-requests` instructs the browser to gracefully upgrade the connection, rather than blocking all HTTP resources, which is particularly useful if an HTTPS site still contains references to HTTP-based active content (like scripts). To understand which sites leveraged this added benefit of CSP, for each of the 347 sites that deployed `upgrade-insecure-requests` within the last month of our analysis, we determined when each started to use the directive. Subsequently, we downloaded the snapshots for those site's main pages from the IA for 31 days after the first `upgrade-insecure-requests` deployment.

Based on the snapshots we collected *before* `upgrade-insecure-requests` was deployed, we find that 251 (72.3%) sites deployed the directive as part of a transition to HTTPS. We base this observation on the archived URL, which indicates if the site had been loaded via HTTPS by the Archive's crawlers. For those sites, we parsed the HTML of all collected snapshots once `upgrade-insecure-requests` was deployed and extracted the URLs of external scripts, images, frames, and stylesheets. On 77 sites, we found that within a month from originally deploying `upgrade-insecure-requests`, resources were still linked via HTTP. Among these sites, we found high-profile pages such as `wired.com`, `airasia.com`, and `aol.com`. For those three sites, we further investigated how much longer they linked to HTTP resources, downloading all snapshots until December 31, 2018. For Air Asia and AOL, on the last day of our experiment, there was still one HTTP resource on their main page, even though both sites moved to HTTPS in 2017. Wired, which started deploying HTTPS in June 2016, removed the last HTTP resource from the start page only in September 2017.

Overall, these results indicate that `upgrade-insecure-requests` is a useful mechanism and therefore widely used when a site migrates to HTTPS, highlighted by the fact that more than 70% of sites deploying `upgrade-insecure-requests` for the first time did so as part of their move to HTTPS. In addition, 77 sites still made use of HTTP-linked resources after their move to HTTPS. `upgrade-insecure-requests` allowed these sites to function correctly due to the graceful upgrade to HTTPS.

VII. CSP for Framing Control

This section presents our historical findings for the third use case of CSP, i.e., framing control. In particular, we investigate to what extent CSP's `frame-ancestors` has achieved its goal of deprecating the underspecified XFO header, by analyzing sites that leverage its flexibility, sites that deploy both headers, and highlighting numerous cases in which `frame-ancestors` would be required to achieve proper protection.

A. Evolution of CSP for Framing Control

As discussed in Section II, XFO was not standardized before it was put into browsers, leading to inconsistent enforcement, which in turn enabled double-framing attacks. This is why CSP removes any ambiguity and checks all of a frame's ancestors. The high-level overview of our analysis on the use of CSP for framing control against XFO is shown in Figure 9. We observe a steady increase in the usage of XFO until the end of our analysis in December 2018. In that month, 3,253

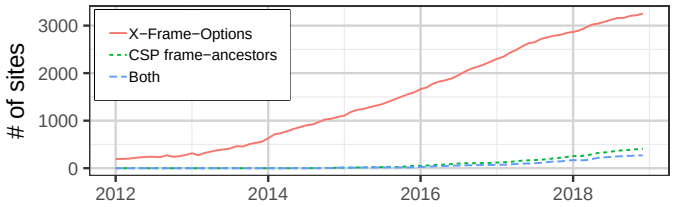

Fig. 9: *Evolution of* `X-Frame-Options` *and* `frame-ancestors`

sites made use of XFO, whereas only 409 used `frame-ancestors`. Moreover, out of the sites that use CSP for framing control, 270 do so in combination with XFO. In the following, we first analyze to what extent the flexibility of `frame-ancestors` is used by sites, i.e., we determine if the same whitelist could also be expressed with XFO. Second, for those sites that make use of both CSP and XFO for framing control, we analyze how the goals of the deployed headers differ. Finally, we report on how many sites would *have to* move to CSP to achieve universal protection.

1) Leveraging the Flexibility of CSP: The protection offered by XFO is coarse-grained both because of the small number of configuration options, but also because of Google Chrome's decision (shared by all Chrome-based browsers) to not support the `ALLOW-FROM` directive. As part of our analysis, we wanted to determine which sites needed the additional flexibility offered by CSP's `frame-ancestors` directive. We aggregate these by year, as this shows a clearer trend compared to monthly analyses. The result is shown in Table IV. We find that starting from 2015 when a meaningful number of sites adopted `frame-ancestors` (shown as CSP-FA), at least 50% of them required the flexibility of CSP. For those, we find that the most common pattern is whitelisting all origins from the same site (104 of 321 in 2018). Moreover, manual investigation of the remaining cases showed that CSP was often used to whitelist sites from the same company (e.g., `icloud.com` allows to be framed from `*.icloud.com` and `*.apple.com`). Hence, apart from delivering different XFO headers depending on the Referer or Origin header, these sites could not run uninhibited without `frame-ancestors`.

2) Combining XFO and CSP frame-ancestors: When an XFO header and a CSP `frame-ancestors` directive are both set, CSP-compatible browsers are supposed to ignore XFO. However, in older browser versions, XFO was the only mechanism to protect users from clickjacking. Based on this insight, we compared the semantics of XFO and `frame-ancestors` (CSP-FA) in cases where both were present, so as to understand how site operators dealt with different browsers. Over the whole period of time, 394 Web sites made use of both XFO and `frame-ancestors` at least once on the same day. Out of those, 290 did so inconsistently (at least once), i.e., were not semantically equivalent (e.g., did not combine SAMEORIGIN and `self`). On 70 sites, `frame-ancestors` was used to relax the security boundary from the same *origin* to the same *site*, e.g., `https://site.com` `https://*.site.com`, while the XFO header was set to SAMEORIGIN. 185 sites used CSP (at least once) to relax the security boundary even further than the same site, leveraging SAMEORIGIN for older browsers. Notably, these sites made the best of a bad situation, given that browsers without support for `frame-ancestors` are at least more secure than when

Year	Used CSP-FA	Required CSP-FA
2014	13	3 *(23%)*
2015	60	32 *(53%)*
2016	133	92 *(69%)*
2017	260	182 *(70%)*
2018	460	321 *(70%)*

TABLE IV: *Number of sites per year that set a frame-ancestors directive, as well as number and fraction of sites that set policies not expressible by X-Frame-Options*

XFO is absent. Only twice was CSP used to deploy a more restrictive policy; specifically going from the invalid value `ALLOWALL` (essentially disabling XFO) to whitelisting the site and all of its subdomains. Here, the site operators opted for an insecure solution for IE users, which is the only current browser that does not support CSP framing control. Finally, 65 sites deployed an invalid XFO header alongside CSP at least once, e.g., by using contradicting values like `SAMEORIGIN, DENY`, effectively disabling any protection for legacy browsers.

3) Replacing XFO with CSP-FA: It is worth noting that even though the double-framing attack is mitigated in most browsers since 2017 [26] (except for Edge and IE), using directives that are not supported by all browsers (e.g., `ALLOW-FROM` for Chrome and Safari) can have dangerous consequences. For those, setting an `ALLOW-FROM` directive in XFO results in the header being ignored completely (failing insecurely [18]). We found that in December 2018, 116 sites in our dataset made use of a non-universally supported directive, meaning the developers would have to deploy CSP's `frame-ancestors` to properly secure their sites. Notably, of the 3,253 sites that use XFO in December 2018, but not `frame-ancestors`, 362 already deploy a CSP. Hence, even though these sites would merely have to add a directive to make use of the more fine-grained and consistent framing control, they still *only* resort to the deprecated security header. While one could argue that this might originate from old, outdated CSP policies, we find that 120 of these sites made use of `upgrade-insecure-requests`: a directive that was only added after support for `frame-ancestors` was enabled.

VIII. In-Depth Analysis of Deployed CSPs

We now attempt to shed light on the main reasons which may affect a successful adoption of CSP. First, we focus on sites that gave up CSP and aim to attribute this choice to its cause. Second, we highlight characteristics of sites which kept running trivially insecure policies for content restriction, before finally investigating sites which deployed effective policies.

A. Reasons for Giving Up on CSP

We start by investigating whether we can attribute changes in the deployed CSPs to violations of said policies. Specifically, we focus on those sites which tried out CSP in enforcement mode for at least one month, but gave up. We define "giving up" as not having had a policy for a specific use case in all of December 2018. This way, we lower the risk of incorrectly flagging a site for having given up just because it did not have CSP for the last few days of 2018. For each domain, we investigate the final policy snapshot for each use case, namely content restriction, TLS enforcement,

and framing control. Out aim is to understand why a given domain stopped using CSP in a particular capacity.

1) Content Violations: CSP's initial purpose was to restrict the inclusion of content into a page. Hence, our first analysis focuses on sites that gave up this use case as per our earlier definition. For each site, we determined the exact timeframe for which the last policy was deployed. We then visited the start page on the last day of policy deployment, following all links to the same site that were archived in the timeframe of the last deployed CSP. While all of our previous analyses relied on downloading and statically parsing HTML documents, for this experiment we relied on an instrumented Chrome browser. This allowed us to not only determine if the resources statically linked in the document caused violations, but also observe if dynamically-added content interfered with CSP. We limited the analysis to the first level of links (in total, 3,347 URLs), as this already required us to make an additional 421,684 requests through loaded scripts, images, fonts, etc.

Overall, we found 63 domains which attempted content restriction, but eventually gave up. For 15, we found violations of the deployed CSP through dynamic analysis. When purely relying on the resources statically linked in the HTML document, only 7 sites indicated a violated CSP, which shows the benefits of the dynamic analysis. Unfortunately, the Archive does not always manage to correctly rewrite URLs to included resources, as shown by Lerner et al. [20]. Hence, it is likely that even the more accurate dynamic analysis missed at least a few violations, merely due to the fact that third-party code was not even executed. Of the 15 sites with violations, we analyzed which party included the violating resources. On 9 pages, the violation was caused by third-party code, whereas on 8 they were caused by first-party code, with an overlap of two domains having both first- and third-party-caused violations.

On the remaining 48 sites for which we could not find violations, we performed two analyses. First, since we cannot reason about violations that are deeply hidden in the application, we performed a live experiment on September 24, 2019, in which we crawled the live versions of all websites in our data set. For sites with a CSP on the start page, we randomly sampled 10 same-site subpages and checked their CSPs. In doing so, we found that of the 1,202 sites with CSP on the start page, 1,024 (85%) appear to have a site-wide deployment of the same policy. Hence, assuming a similar distribution of site-wide policy deployment for the archived versions, the abandoning of site-wide policies may very well be due to violations on pages other than the ones we crawled. Eventually, we resorted to manually checking the deployed policies to classify them, so as to provide an educated guess about the reason for dropping CSP. For 25 sites, we found that their policies showed increasing numbers of third-party entries (e.g., `milanoo.com` and `snai.it`). For those, it is likely that the overhead of keeping a whitelist of their dependencies was too burdensome for the operators. For another 9 sites, the policies were trivially insecure (e.g., `raspberrypi.org`) before they were dropped. Here, we argue that deploying insecure CSPs can eventually lead to their removal.

Summing up, though our archival analysis can only provide glimpses of the reasons why site operators gave up CSP, we find that more than half of the sites for which we could find violations had these caused by third parties. For the rest, even though we could not find specific violations, a significant fraction had large whitelists with tens of third parties, indicating that reliance on third parties could well be the major reason behind the sites' decision to abandon CSP.

2) TLS Violations: To understand whether a TLS enforcement policy was violated, we need to check multiple angles. Trivially, if a site includes a resource via HTTP and has `block-all-mixed-content` or `default-src https://*` (or equivalent for other resource types), the CSP is violated. Contrary to this straightforward case, understanding if `upgrade-insecure-requests` could have caused incompatibility is more involved, as the Archive crawler does not honor `upgrade-insecure-requests`, i.e., would not automatically archive the HTTPS variant of an upgradable resource. As a first step, we assume that if a site was delivered over HTTPS, all resources from the same host would also be available via HTTPS (e.g., the page was `https://foo.com` and the resource `http://foo.com/bar.png`). We do not look for violations in this case, to minimize the risk of false positives. We then leverage the observation that widely-used resources which are available over both protocols would have been loaded and archived by the crawler at least once over a secure connection, as at least *one* site would likely have included it via HTTPS. We exploit this by querying the CDX API for the HTTPS variant of the URL we want to check for upgradability, limiting the results to URLs archived within ± 30 days of the HTTP resource. If we can find an HTTPS variant, we mark the resource as upgradable. We then flag all remaining resources, i.e., those with only HTTP snapshots in the IA, as non-upgradable. Though this approach might not be a perfect solution, it is the best option considering the Archive limitations. Hence, whenever a resource is non-upgradable, but is included in a site with `upgrade-insecure-requests`, we say that the site's policy is violated.

In total, 46 domains were labeled as having given up TLS enforcement. Similar to content violations, we used Chrome to crawl the first level of links beginning from the last snapshot with CSP. For 28 of the sites, we detected a non-upgradable resource on the crawled pages. In addition, for 4 domains, TLS enforcement seemed to have been dropped along with all other CSP directives, i.e., was collateral damage. For the remaining 14 domains, we could not reach a definitive conclusion. However, given the insights about site-wide deployment we discussed in the previous section, we plausibly expect that many of these sites may have had at least one non-upgradable resource on subpages we did not crawl.

3) Framing Control Violations: To understand if a given CSP would have caused a framing violation, we would have to retroactively investigate which other sites framed a given page. As this is not feasible, we instead resort to a heuristic to determine if the removal of `frame-ancestors` was due to framing control issues. As we have seen before, `frame-ancestors` is often used in combination with XFO. Hence, if a given site has encountered an issue related to framing control, it would likely not only remove `frame-ancestors`, but also drop or adjust XFO. Therefore, once a site has stopped using `frame-ancestors`, we check its XFO status on that same day, as well as on the next snapshot. If a site has also stopped using XFO, it is extremely likely that framing control in general proved to be a problem.

In our dataset, 69 domains used CSP for framing control, but gave up on that use case. Given our above classification, we found that for 42 sites, restricting framing in general proved to be problematic, i.e., they dropped both `frame-ancestors` and XFO at the same time. In addition, we found 7 sites which moved from explicitly allowing a hostname through both CSP and XFO to only XFO `SAMEORIGIN`. These cases are interesting, as they indicate that developers determined this to be sufficient to constrain framing (as the flexibility of CSP was not necessary); notably showing the lack of awareness of the dangers of double-framing attacks. In another 7 cases, `frame-ancestors` was removed as collateral damage, i.e., XFO was used before and after CSP's removal. Surprisingly, we also observed two sites moving from exclusively using `frame-ancestors` to XFO, indicating those operators were also not aware of the drawbacks of XFO. Overall, we find that sites do not give up on `frame-ancestors` for reasons specific to CSP, but rather because they either find framing control too cumbersome, or altogether unnecessary.

B. Investigating Insecure Policies

As our results have indicated, around 90% of sites that tried to restrict content did so insecurely, e.g., by using `unsafe-inline` or whitelisting entire schemes. To understand the reasons behind this, we specifically looked at the content of all pages which deployed such insecure policies, and were never able to remove those unsafe keywords. We discovered 467 websites exhibiting this behavior. For each of the websites, we checked every snapshot from the Archive (totaling around 118K requests) for the presence of inline scripts, event handlers, and the number of third parties in the page.

Overall, 455 sites (97%) had inline scripts on the start page at least once while running an insecure policy. Moreover, 317 used event handlers (68%) and in the median, each site relied on 3 third parties (with a maximum of 26 third parties for a single site). It is worth noticing that all these numbers likely represent lower bounds, as we did not crawl the sites any further. Nevertheless, a staggering 68% of sites relied on event handlers, meaning they could not deploy a policy without `unsafe-inline` given the current CSP specification. The results also highlight the difficulty that operators face when trying to retrofit CSP; essentially, a policy that is tacked onto an existing application is virtually always trivially insecure.

C. Analyzing Secure Sites

To complement our previous analysis, we now focus on sites which managed to deploy a secure policy, i.e., one without whitelisting entire schemes or using `unsafe-inline`. While prior work [50] has indicated that additional risks may originate from whitelisting origins with JSONP endpoints or allowing Flash to be hosted locally, we do not consider these additional factors. In total, we found that 40 sites were able to deploy a meaningfully secure policy and still have that in operation at the end of our analysis timeframe. Notably, another 7 at some point deployed a strict policy; however, they either added the unsafe keywords again or entirely disabled CSP after mere days, indicating their policy caused functionality issues. In particular, for 3 sites we found event handlers on their start pages, even though their policy did not specify `unsafe-inline`, hence definitely causing a CSP violation.

Of the 40 sites which can be counted as successfully having deployed CSP for content restriction, 2 actually run policies which interfere with scripts on their start pages as of this writing. When looking at the other 38 cases, we discovered an interesting trend. First, we found 16 adult websites, most of which deployed a strict policy without attempting a more relaxed one before. Interestingly, they all had starting days of their first CSP about 1-2 weeks apart (each). Analyzing the CSPs, we found that they were all whitelisting the exact same sources. Looking at the start dates of CSP deployment, we found that the operators of these sites first experimented on one site with removing the event handlers on the page, exclusively used to track users through Google Analytics. Notably, this behavior of using inline event handlers was even advocated for by Google [3]. Once they had successfully rolled out CSP for one website, they proceeded with others. Of the remaining 22 sites, only 3 had any event handlers on the last snapshot before the deployment of the strict policy.

Overall, we find that of the few sites that were able to deploy a strict policy, virtually all either did not rely on event handlers (on their start page), or only used event handlers for a single, easy-to-change use case (such as registering event handlers programmatically for off-site links). This stands in stark contrast to the results for the sites which failed to deploy a secure CSP, where over two thirds used event handlers.

IX. Framing Control Notification

In general, we observed that sites have a clear preference for XFO over `frame-ancestors`. Moreover, we found cases where XFO was used even though the site deployed directives only introduced after `frame-ancestors`, indicating the CSPs were updated when framing control was already possible. To understand the reason behind these findings, we decided to notify site operators running XFO, informed them about the improved support that CSP's `frame-ancestors` offers, and tried to discover their reasons for preferring XFO. To this end, we checked all live versions of the sites in our dataset starting from May 31, 2019, for their deployed XFO and CSP directives. On June 4, we notified all 2,699 sites that used XFO headers which either had a syntactically incorrect header, used the non-universally-supported ALLOW-FROM directive, or deployed SAMEORIGIN, making them prone to double-framing attacks in Edge and IE. We did not notify the sites that also made use of CSP's framing control, since supporting browsers ignore any XFO headers when `frame-ancestors` is present. Given the insights from prior work on Web notifications [42], we chose to send emails to generic aliases on each domain (info, security, webmaster) as well as to the WHOIS contact (where available). The template of our email can be found in Appendix A. We sent this email from one of the researchers' regular email address, ensuring that recipients could verify our identity. As expected, in line with prior work's findings [42], most emails bounced, either due to non-existing addresses or lack of appropriate MX records.

A. Insights from Initial Responses

Notably though, we received responses from 117 sites which went beyond automated confirmation emails, such as out-of-office responders or confirmation of a created ticket. By categorizing these responses, we discovered that 62 operators

claimed that they would deploy `frame-ancestors` shortly. For a sample of anonymous answers we received, please consult Appendix B. Among the responses, we also found 24 answers which indicated that CSP was too complex to be deployed. In particular, they all claimed to have attempted to deploy CSP for content restriction, but either deferred it, or abandoned the attempt altogether. This is in line with the significant number of sites we discovered in our archival analysis, which either stopped deploying CSP or never moved from report-only to enforcement mode (see Section V-D).

With all respondents, we exchanged further emails, indicating that CSP's `frame-ancestors` could be used without any of the other CSP functionality. In doing so, we received emails from 16 operators stating they were not aware of any issues related to XFO, and 13 who explicitly noted they had not heard of CSP's `frame-ancestors` before. In contrast, 9 informed us in their initial response that they had already deployed the CSP directive. From the notification date and onwards, we continued our daily checks for both XFO and `frame-ancestors`. Overall, we observed an increase from 511 sites deploying `frame-ancestors` before our notifications to 554 sites by June 12, 2019. In particular, for the domains that answered to our initial message, 14 had taken action. Moreover, for the other sites, 4 belonged to a network of sites for which we had received one response. For the remaining 25 sites that rolled out `frame-ancestors` in the 8 days, we could only find two sites for which *all* our sent emails bounced. Hence, we believe that most of the sites deployed `frame-ancestors` as a result of our notification, demonstrating the ease of deployment within mere days.

Finally, in conversations with operators, several mentioned that they relied on external resources for security headers. In checking those resources, we found that they all list XFO as the only defense against framing-based attacks, whereas they advertised CSP as a means to mitigate XSS attacks [4, 5, 13, 38, 49]. Notably, even widely-used sites like `securityheaders.com` consider XFO the only viable option for framing control. Neither this service nor other resources like MDN [24] indicate that CSP can be used for this purpose.

B. Follow-Up Survey

Given the diverse responses we obtained from the notified site operators, we decided to run a more systematic survey, allowing us to ascertain the number of operators aware of issues with XFO, CSP, and the fact that `frame-ancestors` could be used in isolation. We made the survey as brief as possible and only sent it to operators who had previously answered our initial email, with the explicit goal of soliciting a high fraction of responses due to the limited effort necessary to answer the questions. In particular, as prior works have shown, unsolicited surveys have minuscule response rates [11, 42], which is why we decided to only reach out to operators to whom we had previously provided helpful information. The full questionnaire is available in Appendix C. For our survey, until June 12, 2019, we received a total of 39 answers. Out of those, two thirds (27) indicated they were not aware of the inconsistencies around XFO. When asked about why they had deployed XFO in the first place, the majority (20) said they had their own reasons to restrict framing, indicating the awareness of framing-based attacks. Moreover, 31 (79%) respondents indicated that they had been aware of CSP before our notification; yet only 12 of those claimed to have been aware of `frame-ancestors` beforehand. Of those 12, 9 claimed to know that `frame-ancestors` can be used in isolation. These reports suggest that while operators have a general understanding of CSP, they are not aware of all its directives and their security benefits.

For all respondents that indicated to have known about CSP beforehand, 23 said that their site would not work with a reasonably secure policy right away (2 claimed yes, 6 did not know). On the flip side, 29/31 operators believed that CSP could be a viable option to improve their site's resilience to XSS attacks. Additionally, when asked about the use case of TLS enforcement, 22 responded that they knew they could operate TLS enforcement in isolation before our notification. Hence, it appears that while content restriction is clearly known as a goal of CSP, most sites are unable to deploy it due to its complexity. Operators seemed to be more aware of the fact that TLS can be enforced through CSP, but were not as knowledgeable about framing control. This, combined with the insights from resources the respondents indicated (both in the survey and the email conversations) leads us to conclude that resources on CSP critically lack details about framing control.

In terms of tool support, 36/39 respondents answered that they had used the browser console to debug and analyze their site. Hence, if there had been warnings about inconsistencies (or even lack of support for certain directives), those operators would likely have taken action. With respect to required tools, the respondents named better tools to debug CSP errors (locally), improved collection and aggregation of warnings caused in users' browsers, and in general tools to suggest appropriate security headers. As a result of these insights and separate discussions with Google engineers, we filed a Chrome feature request to issue warnings about XFO; in particular to at least warn operators about the unsupported ALLOW-FROM directive and suggest to deploy `frame-ancestors` instead.

C. Limitations and Additional Survey

Our notification and subsequent survey cannot be considered an in-depth analysis due to its unstructured nature (especially of the emails we received). We specifically set up the survey to be brief, so as to achieve a high response rate. However, it is not clear whether security-aware operators filled our survey. Even then, our results are indicative of operators which did not use CSP for framing control, i.e., cannot be considered experts in CSP. While we cannot account for these facts in our initial survey, to partially alleviate the identified shortcomings, we ran our survey a second time after having presented a talk about the evolution of CSP and its different use cases at an OWASP conference. For this, we are confident that professionals with a Web security background answered the questions. We received a total of 20 responses with 10/20 claiming prior knowledge of XFO's shortcomings, and 19/20 being aware of CSP beforehand. 18 of those believed CSP to be a viable option, of which 9 argued their site would be able to run a secure policy. Regarding framing control, 13/19 said they already knew about `frame-ancestors`, of which only 2 did not know that it was feasible to deploy in isolation. In contrast, 9/19 were not aware that TLS enforcement could be deployed in isolation, indicating that this is a little known use-case even

among security experts. Naturally, as for our initial survey, we cannot assess incorrect reporting from operators. Nevertheless, as we stressed the anonymous nature of our survey, we expect the results to be characteristic of the participants.

X. Discussion

We now summarize the evolution of CSP's use cases, enabled by the unique vantage point of the IA, and highlight gathered insights. We then discuss if CSP is too complex, and outline how it can become more useful going forward.

A. Summary of CSP's Use Cases

1) CSP for Script Content Restriction: Our work has confirmed a previously investigated fact [6, 8, 50, 51]: CSP is largely failing as a defense mechanism for script content restriction. Through our longitudinal analysis, we could show that although nonces and hashes have been available since 2014, they have not gained significant popularity over time. On the flip side, most policies make use of `unsafe-inline`, which makes them trivially bypassable by XSS. We argue that this is due to the complexity of deploying CSP in a secure fashion. This is evidenced by the fact that more than half of the Web sites (251/449) which experimented with report-only never switched to enforcement mode. It is also confirmed by the notification responses, in which operators regularly stated they had experimented with CSP, but felt it was incompatible with their application. This anecdotal evidence is supported by our survey, in which only 2/30 respondents familiar with CSP claimed their site could deploy it right away without breakage.

In terms of deploying limited whitelists, we saw ample evidence in our case studies as well as the general uptick in whitelisted sites that curating such whitelists is challenging. Moreover, our analysis of the whitelisted sites has indicated that operators are prone to add typo domains, or leave unregistered domains in their whitelist, effectively undermining the provided security guarantees (on approx. 13% of the sites with content-restricting CSPs). Of the handful of sites which managed to actually deploy a restricted whitelist, both the case studies and the feedback from our notification indicated that curating such a list takes months or even years. Hence, the overall effort of setting up and maintaining a secure policy seems unbearable to all but the biggest players.

2) CSP for TLS Enforcement: Previous studies mainly focussed on CSP as a means to restrict script content, treating TLS enforcement and framing control as side notes. In particular, Weichselbaum et al. [50] reported that only 3% of policies were used to enforce TLS, while Calzavara et al. [8] reported that around 0.5% of the Top 1M used `upgrade-insecure-requests`, without providing further details. Our longitudinal analysis showed that CSP is a very valuable tool for TLS enforcement, being used by about one third of the Web sites that deployed CSP. Most prominently, we observe that 347 sites make use of the `upgrade-insecure-requests` directive to automatically upgrade HTTP resources to HTTPS. We find that this feature is not only used for security purposes but when investigating those sites that deployed `upgrade-insecure-requests` as part of their migration to HTTPS, we found that 77 of 251 (31%) sites still link HTTP resources (on their start page). Here, the added benefit of `upgrade-insecure-requests` enables browsers to upgrade URLs before trying to load them, thereby avoiding mixed content warnings or blockage. Given the increasing adoption of HTTPS [35], we argue that Web sites adopting an `upgrade-insecure-requests` policy would have an easy migration to HTTPS, while at the same time not having the burden of making their applications compliant with a strict content-restricting CSP (e.g., by removing event handlers). This fact, however, according to our survey, is less than well-known.

3) CSP for Framing Control: In contrast to previous studies, our findings indicate that CSP is becoming increasingly popular for framing control, now on par with content restriction (attempts) and TLS enforcement. At the same time, the adoption of CSP for framing control is not nearly as widespread as XFO: CSP with `frame-ancestors` is used in 409 Web sites, while XFO is present on 3,253 Web sites as of December 2018. However, we observe that existing Web sites are taking advantage of the additional flexibility on framing control offered by CSP. In fact, out of the 460 sites using CSP for framing control in all of 2018, 321 sites (70%) used whitelists not expressible by XFO, which suggests that the additional expressiveness of CSP for clickjacking protection is useful in practical cases. Moreover, our notifications showed that about two thirds of respondents were not aware of the added benefit of `frame-ancestors`. While our notifications and the feedback we have received suggest that this can be easily changed for operators we could reach, the resources frequently used by the respondents lack crucial information about this fact. As an example, the Mozilla Developer Network only explicitly mentions "[..] certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks." and does not list framing control as a use case [24].

B. Complex Security Policy?

From both the evidence gathered from our longitudinal study as well as the insights provided to us through the conversations in our notification, CSP for script content restriction seems to be a failing mechanism. Even though modifications to CSP should have made it easier for sites to adopt secure policies (e.g., allowing inline scripts through nonces), we could not observe any significant uptake over time. Moreover, the responses indicated that operators still shy away from CSP for content restriction due to its perceived complexity.

One specific issue that CSP for content restriction has is the moving target it represents. As an example, `google.com` stopped using `strict-dynamic` on July 17th, 2018 even though Google engineers originally proposed the new directive [50]. Given the insights shared by the Google team in a recent presentation [49], they have since moved away from using `strict-dynamic`, favoring the explicit propagation of trust by having nonced scripts attach the nonce whenever they add additional scripts. The reasons are seemingly two-fold: first, support for `strict-dynamic` is not universal (e.g., Safari does not support it). Second, using `strict-dynamic` yields any control over which resources can be included, and opens up the potential for Script Gadget attacks [19]. The loss of control over included resources cannot be addressed through a nonce-based policy, given that any nonced script could just use the nonce to add code from elsewhere. The solution, as also proposed by the Google engineers and at least partially deployed as of now by Dropbox, is *policy composition*. There,

a site sets two CSPs, where one carries a nonce and the other carries a whitelist. Since *both* have to be fulfilled, only scripts that carry the nonce *and* are from explicitly whitelisted hosts can be executed. Given these advances and frequent changes in suggested best practices, it is understandable that operators feel overwhelmed by the complexity of the mechanism.

At the same time, while CSP appears to be failing as a means to mitigate XSS, it has become a successful mechanism to enforce TLS, evidenced by the uptick of this use case. Moreover, `upgrade-insecure-requests` allows sites to seamlessly migrate to HTTPS by upgrading all URLs in-flight. Notably, out of the 10K sites in our original dataset, 7,675 were archived via HTTPS. We downloaded the final snapshot for each domain from the IA and found that of the 7,328 sites without `upgrade-insecure-requests`, 435 had a least one HTTP-linked resource (6%). Given the current implementation of browsers, these sites would at the very least trigger a mixed content warning. This, in light of the results of our survey, in which 21/30 respondents claimed to be aware of the isolated usage of TLS enforcement, shows that sites are clearly not making use of CSP's full potential.

Despite the growth of CSP for framing control, unfortunately, it still lags behind the increasing adoption of XFO and more importantly, the complexity of CSP as such seems to confuse operators. This is evidenced by the fact that in December 2018 we observed 362 Web sites using CSP without `frame-ancestors` in combination with XFO. For these Web sites, it would be trivial to use CSP to enforce the same protection, but this is not done. Though this might be caused by outdated CSPs written before framing control support was added, we observed an interesting phenomenon: a third of the Web sites which deploy CSP without using `frame-ancestors` are making use of `upgrade-secure-requests`, which was introduced to CSP only later. This implies that CSP is often perceived as a complex mechanism to restrict content inclusion and not as a meaningful mechanism to control framing, which can even be used without any restriction on included content.

Overall, we find that CSP has grown from a mechanism aimed at restricting content to a multi-use measure to improve the security of Web applications. Our work has highlighted that this shows success with respect to TLS enforcement and framing control but also indicates that operators tend to shy away from deploying CSP, even though it could in many cases easily benefit their security. We believe this is caused by the ever-increasing complexity of the CSP mechanism. Apart from the already existing directives, new features for content restriction, such as more involved mechanisms for securing script code in attributes [32], are being added. In addition, with features such as `navigate-to` [31] and the signaling for Trusted Types [54], CSP is becoming a highly complex, generic Security Policy. This perceived complexity was also echoed in the notification responses, with operators explicitly naming complexity as the hurdle towards CSP deployment.

C. Quo Vadis, CSP?

Given our insights regarding CSP's (in)ability to restrict script content and the reasons we uncovered through our analysis, we propose three actionable steps which we believe can help CSP's adoption and the security of deployed policies.

1) `unsafe-nonced-elements`: One major roadblock to CSP adoption is the inability to use event handlers. While Chrome has recently added support for `unsafe-hashed-attributes` [33], this only enables operators to make their hash-based whitelist apply to event handlers. In practice though, we observed up to 3,344 different event handlers on a single page. While the median is only at around 5, any update to the event handlers needs to be propagated to the CSP header (*for all pages*). Instead, we propose that CSP be extended to allow for nonced elements, i.e., when an element carries a nonce, any event handlers are permitted on that element but not its children. This would remove the need of always resorting to `unsafe-inline`. This proposed changes comes with certain risks, namely nonce-reuse attacks and injections inside nonced elements. The first is acknowledged by the CSP standard authors, who proposed a fix as follows [1]: if a script tag is nonced, it will only be executed if within all of its attributes, no additional opening script tag can be found. This could be easily extended to check for other elements. To understand the feasibility of the approach, for all of the 317 sites with `unsafe-inline` which used event handlers (see Section VIII-B) we checked each snapshot to gauge whether an element with an event handler also contained markup in any of its attributes. Assuming these were to be nonced, but contain markup, CSP would falsely block the nonced element from executing the event handlers. This analysis showed that not a single snapshot had such a case; meaning that our proposal would likely not cause incompatibilities. Second, an attacker could abuse an injection *inside* a nonced element to add additional event handlers. However, this is a significantly lower attack surface than using `unsafe-inline`, for which an injection anywhere in the page is sufficient to allow for an XSS attack. Hence, we argue that this is a viable option to make CSP more usable, while not fully sacrificing security.

2) Incorporate CSP into Development Cycle: As our analysis has highlighted, many operators attempted to deploy CSP to an existing application, only to either end up with a trivially bypassable policy or give up on CSP altogether. With a few exceptions, deploying CSP retroactively to an existing application does not appear to be a viable strategy. This is aggravated by the use of third parties, which are known to dynamically add additional content. Hence, we argue that CSP must be incorporated into the development cycle. In particular, we urge IDE vendors to add checks for CSP incompatible code at development time by, e.g., warning developers to not add inline scripts or event handlers, but instead proposing to externalize the desired functionality. In addition, as prior work has documented, third parties often add (script) content dynamically. We could confirm that by attributing over half the detected content violations to third parties. While prior work, such as Calzavara et al. [7] and Weichselbaum et al. [50], have proposed means to address this issue, we instead argue that third parties should be explicit about their dependencies and their impact on CSP (e.g., if they only add scripts dynamically, thereby enabling support for `strict-dynamic`). In this way, during development, a web developer could decide to incorporate another similar vendor which provides the same service with less CSP interference. This could also be incorporated into IDEs, which could automatically analyze included parties and warn the developers about roadblocks for CSP.

3) Updated Informational Material for Developers: In light of our findings and survey responses, it appears that the complexity for script content restriction gives CSP a bad reputation. Given that this is *not* counteracted by widely used resources pointing out the easy-to-deploy use cases of TLS enforcement and framing control, we advocate for clear communication of the individual goals in such resources. Likewise, we argue that browser vendors are in a unique position to improve upon this situation, by warning developers through the console about inconsistently implemented mechanisms like X-Frame-Options, even providing a quick fix for the issue by deploying CSP. To that end, we have started discussions with both the Chrome and Firefox team on addressing this issue, with the hope of allowing more sites to leverage the easy-to-use capabilities that CSP can offer for better security.

XI. Related Work

1) Large-Scale Analyses of CSP: The CSP deployment in the wild has undergone at least four authoritative studies as of now [6, 8, 50, 51]. The first investigation was published by Weissbacher et al. in 2014 and mostly focused on the challenges of CSP adoption: the authors identified a slow, sub-optimal CSP deployment and proposed techniques for semi-automated policy generation [51]. Weichselbaum et al. in 2016 highlighted that more than 90% of the CSPs in the wild provided no protection against XSS due to trivial bypasses such as the use of `unsafe-inline` or insecure whitelists [50]. The authors then recommended the use of `strict-dynamic` to deploy CSP more securely. In the same year, Calzavara et al. identified similar issues and performed a longitudinal analysis of CSP deployment over 4 months, showing that CSPs change less frequently than needed [6]. Their study was later extended to 6 months, showing that the fraction of sites mitigating XSS increased over time due to nonces, but most policy changes in the wild were not targeted at improving security [8].

Our work improves upon previous analyses of CSP in different ways. First, we present the first analysis of the security impact of expired domains on existing CSPs, complementing previous findings of the insecurity of whitelists [19, 50] with a new delicate aspect. Moreover, our longitudinal lense from 2012 to 2018 not only provides the most comprehensive study on CSP deployment to date, but allows us to document the types of struggles sites face when deploying CSP for content restriction, detailing insights other works could not uncover. We then turn our attention to aspects of CSP which have not been thoroughly evaluated in previous work, namely the use of CSP for framing control and for TLS enforcement, which we also investigate through hands-on experience enabled by a notification campaign. Finally, we complement all of our findings by gathering insights from the field based on email responses to the notification and our follow-up survey.

2) Other Work on CSP: Van Acker et al. studied the inability of CSP to prevent data leaks and proposed mitigation techniques against specific attack vectors [44]. Hausknecht et al. observed that browser extensions may force Web pages into requesting resources which are not whitelisted by their CSP and proposed an endorsement mechanism to solve possible compatibility issues [15]. Somé et al. identified a subtle interaction between CSP and the Same Origin Policy which allowed bypasses of the security guarantees offered by CSP [39]. Calzavara et al. [7] proposed Compositional CSP: an extension of CSP designed to better support the dynamic nature of most modern Web sites. Finally, several researchers have proposed automated techniques to synthesize CSPs for existing Web applications, such as DeDaCoTa [12] or CSPAutoGen [34].

3) Historical Analyses of the Web: The idea of using the Internet Archive for historical security analyses was first employed by Lerner et al. who used it to conduct a study on how Web tracking evolved over the course of 20 years [21]. In 2017, Stock et al. used the same method to investigate the general evolution of client-side security [41]. In particular, they investigated the prevalence of client-side threats (such as Client-Side XSS) and adoption of mitigation techniques. They also noted an uptake in the usage of CSP since 2014, however, did not focus on any particular analysis of the security implications of the deployed policies. Later that year, Lerner et al. showed that the Internet Archive is prone to attacks that leverage externally referenced JavaScript resources, effectively allowing attackers to modify the rendered content [20]. Their work is the reason why the Archive deployed their own CSP.

XII. Conclusion

In this paper, we conducted a longitudinal analysis of the deployment and evolution of CSP since 2012. Leveraging the Internet Archive to collect the historical headers for 10,000 highly ranked websites for seven years, we identified that while CSP was initially meant as a mitigation for script injection, it has evolved into a mechanism that is equally often used to control framing and enforce TLS connections. Our longitudinal analysis allowed us to document the struggle developers face when constructing a secure and functional policy for content restriction, and highlighted that even secure CSPs are prone to bypasses through typos and expired domains. Combined with the lack of adoption of new features such as `strict-dynamic`, this lead us to conclude that script-restricting parts of CSP are unlikely to succeed in the future. Moreover, while CSP is increasingly deployed for framing control and TLS enforcement, their adoption rate is still unsatisfactory. The insights gathered from our survey indicate that CSP has earned a bad reputation due to its complexity in content restriction, resulting in developers shying away from *any* part of CSP. Even though the alternative use cases for CSP are easy to deploy, this bad reputation, unless counteracted by tools, browser vendors, and informational material alike, significantly hampers CSP's ability to improve the Web's security.

References

[1] "Prevent nonce stealing by looking for "<script" in attributes of nonced scripts," https://github.com/w3c/webappsec-csp/issues/98.

[2] "Dataset used in our analysis," https://pastebin.com/NbFxNmc1.

[3] "Google analytics legacy documentation," https://developers.google.com/analytics/devguides/collection/gajs.

[4] Ambroise Maupate, "Nginx CSP example," https://gist.github.com/ambroisemaupate/bce4b760405558f358ae, 2019.

[5] Bruno Scheufler, "Using security-related headers to secure your application against common attacks," https://tinyurl.com/y68c4lpp, 2019.

[6] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content security problems? evaluating the effectiveness of content security policy in the wild," in *CCS*, 2016.

[7] ——, "CCSP: controlled relaxation of content security policies by runtime policy composition," in *USENIX Security*, 2017.

[8] ——, "Semantics-based analysis of content security policy deployment," *TWEB*, 2018.

[9] Can I use..., "Content Security Policy 1.0," https://caniuse.com/#feat=contentsecuritypolicy, 2019.

[10] Common Crawl, "So you are ready to get started," http://commoncrawl.org/the-data/get-started/, 2019.

[11] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *CCS*, 2017.

[12] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "dedacota: toward preventing server-side xss via automatic code and data separation," in *CCS*, 2013.

[13] Experiments with Google, "Content Security Policy," https://csp.withgoogle.com/docs/strict-csp.html, 2019.

[14] GitHub Blog, "GitHub CSP Blog Post," https://blog.github.com/2013-04-19-content-security-policy/, 2013.

[15] D. Hausknecht, J. Magazinius, and A. Sabelfeld, "May i? - content security policy endorsement for browser extensions," in *DIMVA*, 2015.

[16] J. Hodges, C. Jackson, and A. Barth, "RFC6797: Http strict transport security (hsts)," https://tools.ietf.org/html/rfc6797, 2012.

[17] Internet Archive, "About the internet archive," https://archive.org, 2019.

[18] E. Lawrence, "This page frames a victim page in myriad ways," http://www.enhanceie.com/test/clickjack, 2019.

[19] S. Lekies, K. Kotowicz, S. Groß, E. A. V. Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *CCS*, 2017.

[20] A. Lerner, T. Kohno, and F. Roesner, "Rewriting history: Changing the archived web from the present," in *CCS*, 2017.

[21] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *USENIX Security*, 2016.

[22] M. Luo, P. Laperdrix, N. Honarmand, and N. Nikiforakis, "Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers," in *NDSS*, 2019.

[23] MDN, "Data URIs," https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs, 2019.

[24] ——, "Content Security Policy (CSP)," https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP, 2019.

[25] ——, "SecurityPolicyViolationEvent," https://developer.mozilla.org/en-US/docs/Web/API/SecurityPolicyViolationEvent, 2019.

[26] ——, "X-Frame-Options," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options, 2019.

[27] Microsoft, "CSP Level 3 strict-dynamic source expression," https://tinyurl.com/y3d6ljjk, 2019.

[28] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *CCS*, 2012.

[29] OWASP, "HTTP Strict Transport Security Cheat Sheet," https://www.owasp.org/index.php/HTTP_Strict_Transport_Security_Cheat_Sheet, 2018.

[30] ——, "Clickjacking Defense Cheat Sheet," https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet, 2017.

[31] A. Paicu, "CSP 'navigate-to' directive," https://www.chromestatus.com/feature/6457580339593216, 2018.

[32] ——, "CSP: 'script-src-attr', 'script-src-elem', 'style-src-attr', 'style-src-elem' directives," https://www.chromestatus.com/features/5141352765456384, 2018.

[33] ——, "CSP3: unsafe-hashed-attributes," https://www.chromestatus.com/features/5867082285580288, 2017.

[34] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *CCS*, 2016.

[35] A. Porter Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring https adoption on the web," in *USENIX Security*, 2017.

[36] D. Ross and T. Gondrom, "RFC7034: Http header field x-frame-options," https://tools.ietf.org/html/rfc7034, 2013.

[37] Q. Scheitle, O. Hohlfeld, J. Gamba, J. Jelten, T. Zimmermann, S. D. Strowes, and N. Vallina-Rodriguez, "A long way to the top: Significance, structure, and stability of internet top lists," in *IMC*, 2018.

[38] Scott Helme, "Security Headers," https://securityheaders.com, 2019.

[39] D. F. Somé, N. Bielova, and T. Rezk, "On the content security policy violations due to the same-origin policy," in *WWW*, 2017.

[40] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *WWW*, 2010.

[41] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security," in *USENIX Security*, 2017.

[42] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow, "Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications," in *NDSS*, 2018.

[43] P. Toomey, "GitHub's CSP Journey," https://githubengineering.com/githubs-csp-journey/, 2016.

[44] S. Van Acker, D. Hausknecht, and A. Sabelfeld, "Data exfiltration in the face of CSP," in *AsiaCCS*, 2016.

[45] T. Van Goethem, P. Chen, N. Nikiforakis, L. Desmet, and W. Joosen, "Large-scale security analysis of the web: Challenges and findings," in *TRUST*, 2014.

[46] W3C Working Group, "Content Security Policy (Level 2)," https://www.w3.org/TR/CSP2/, 2016.

[47] ——, "Content Security Policy (Level 3)," https://www.w3.org/TR/CSP3/, 2018.

[48] Y.-M. Wang, D. Beck, J. Wang, C. Verbowski, and B. Daniels, "Strider typo-patrol: Discovery and analysis of systematic typo-squatting," *SRUTI*, 2006.

[49] L. Weichselbaum and M. Spagnuolo, "CSP - A Successful Mess Between Hardening and Mitigation," https://tinyurl.com/yyohn6o6.
[50] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *CCS*, 2016.
[51] M. Weissbacher, T. Lauinger, and W. K. Robertson, "Why is CSP failing? trends and challenges in CSP adoption," in *RAID*, 2014.
[52] M. West, "Upgrade insecure requests," https://www.chromestatus.com/feature/6534575509471232, 2018.
[53] ——, "Mixed content," https://www.w3.org/TR/mixed-content/, 2016.
[54] WICG, "Explainer: Trusted Types for DOM Manipulation," https://github.com/WICG/trusted-types#limiting-policies, 2018.

APPENDIX

A. Email notification template

Dear $domain team,

We are a team of academic researchers from $institutions investigating the usage of security headers on the Web.

As part of our analysis, we are investigating the usage of the X-Frame-Options header to control framing on the Web. Based on our analysis, your site is attempting to control framing with the following value: `SAMEORIGIN`

We noticed that this value potentially allows for double-framing attacks with certain browsers such as Internet Explorer and Edge (see https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options#Browser_-compatibility)

The currently proposed way to ensure that all modern browsers properly protect against framing attacks is to use the Content-Security-Policy directive frame-ancestors. In particular, for your value of X-Frame-Options, the corresponding value is: `frame-ancestors 'self'`

Note that in order to protect older browsers, keeping X-Frame-Options in place is recommended. As CSP takes precedence over X-Frame-Options, securing legacy clients without interfering with modern browsers is possible through the usage of the DENY directive in XFO.

For further information on CSP's frame-ancestors, please refer to https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors

As this email is part of a research project in which we are trying to understand the lack of adoption of CSP in the wild, it would be immensely helpful if you could provide us with feedback regarding the lack of CSP's frame-ancestors to protect against framing attacks on your Web site (i.e., were you not aware of the CSP mechanism, that specific CSP directive, or, were you not adopting it, for some other reason?)

Please note that this email is only part of an academic research project and not meant to sell any products or services.

Best regards
$researchers

B. Quotes from Responses

1) Complexity of CSP: "Reading the experience of people that tried to do a full CSP implementation is just scary". "In previous discussions about CSP, we've been worried that the risk of accidentally breaking some interaction we have with other [sitename] systems (or the few third party tools we integrate with) outweighs the benefit of implementing these sorts of changes." "We are pretty certain that there are a lot of pitfalls with implementation of these headers, that might break sections or uses of our site."

2) XFO Dangers: "We were vaguely aware of the frame-ancestors option, but our understanding was that XFO was sufficient for securing all clients". "While we were aware of CSP itself, we were unaware of the fact, that X-Frame-Options allows for attacks under certain conditions, which can be mitigated by using the frame-ancestors directive of CSP."

3) `frame-ancestors`: "In my opinion the only advantage of CSP is to protect against XSS [...]". "As we were not that aware of CSP framing control, we were also not aware of its implementation (no side effects)"

C. Survey Questionnaire

1) Did you know about the inconsistent understanding of browsers of the X-Frame-Options header before our notification (such as the lack of support for ALLOW-FROM in Chrome and Safari as well as the potential threat of double-framing in Edge/IE)? (Yes/No)
2) Why have you implemented the X-Frame-Options header? (Penetration test or consultant suggested it/Tools we used suggested it/Own decision to restrict framing/Other (free text))
3) Did you know about CSP before our notification? (Yes/No)
4) (only if Q3 was yes) Would your site work out of the box if you deployed a script-content restricting CSP today (disallow eval, inline scripts, and event handlers)? (Yes/No/Don't know)
5) (only if Q3 was yes) Do you believe CSP is a viable option to improve your site's resilience against XSS attacks? (Yes/No/Don't know)
6) (only if Q5 was no) Why do you think CSP is not viable for your site? (free text)
7) (only if Q3 was yes) Did you know about the frame-ancestors directive and its improved protection capabilities compared to X-Frame-Options before our notification? (Yes/No)
8) (only if Q7 was yes) Did you know that frame-ancestors can be deployed independently of any other part of CSP before our notification? (Yes/No)
9) (only if Q3 was yes) Did you know that CSP can be used (in isolation) to ensure no HTTP resources can accidentally be loaded (through block-all-mixed-content) and to enforce TLS for all resources (through upgrade-insecure-requests)? (Yes/No)
10) Do you ever use the developer tools to debug or analyze your site? (Yes/No)
11) What kind of tool support would be useful to you to secure your application? (free text)