Why is CSP Failing? Trends and Challenges in CSP Adoption

Michael Weissbacher, Tobias Lauinger, and William Robertson

Northeastern University, Boston, USA
{mw,toby,wkr}@ccs.neu.edu

Abstract. Content Security Policy (CSP) has been proposed as a principled and robust browser security mechanism against content injection attacks such as XSS. When configured correctly, CSP renders malicious code injection and data exfiltration exceedingly difficult for attackers. However, despite the promise of these security benefits and being implemented in almost all major browsers, CSP adoption is minuscule—our measurements show that CSP is deployed in enforcement mode on only 1% of the Alexa Top 100.

In this paper, we present the results of a long-term study to determine challenges in CSP deployments that can prevent wide adoption. We performed weekly crawls of the Alexa Top 1M to measure adoption of web security headers, and find that CSP both significantly lags other security headers, and that the policies in use are often ineffective at actually preventing content injection. In addition, we evaluate the feasibility of deploying CSP from the perspective of a security-conscious website operator. We used an incremental deployment approach through CSP's report-only mode on four websites, collecting over 10M reports. Furthermore, we used semi-automated policy generation through web application crawling on a set of popular websites. We found both that automated methods do not suffice and that significant barriers exist to producing accurate results.

Finally, based on our observations, we suggest several improvements to CSP that could help to ease its adoption by the web community.

Keywords: Content Security Policy, Cross-Site Scripting, Web Security

1 Introduction

The web as a platform for application development and distribution has evolved faster than it could be secured. Consequently, it has been plagued by numerous classes of security issues, but perhaps none are as serious as content injection attacks. Content injection, of which cross-site scripting (XSS) is the most well-known form, allows attackers to execute malicious code that appears to belong to trusted origins, to subvert the intended structure of documents, to exfiltrate sensitive user information, and to perform unauthorized actions on behalf of victims. In response, many client- and server-side defenses against content injection have been proposed, ranging from language-based auto-sanitization [17] to sandboxing of untrusted content [12] to whitelists of trusted content [11].

Content Security Policy (CSP) is an especially promising browser-based security framework for refining the same-origin policy (SOP), the basis of traditional web security. CSP allows developers or administrators to explicitly define, using a declarative policy language, the origins from which different classes of content can be included into a document. Policies are sent by the server in a special security header, and a browser supporting the standard is then responsible for enforcing the policy on the client. CSP provides a principled and robust mechanism for preventing the inclusion of malicious content in security-sensitive web applications. However, despite its promise and implementation in almost all major browsers, CSP is not widely used in practice—in fact, according to our measurements, it is deployed in enforcement mode by only 1% of the Alexa Top 100.

In this paper, we present the results of a long-term study to determine why this is the case. In particular, we repeatedly crawled the Alexa Top 1M to measure adoption of web security headers, and find that CSP significantly lags behind other, more narrowly-focused headers in adoption. We also find that for the small fraction of sites that have adopted CSP, it is often deployed in a manner that does not leverage the full defensive power of CSP.

In addition to our Internet-scale study, we also quantify the feasibility of incrementally deploying CSP from the perspective of a security-conscious administrator using its report-only mode at four websites. Although this is an oft-recommended practice, we find significant barriers to this approach in practice due to interactions with browser extensions and the evolution of web application structure over time.

Finally, we evaluate the feasibility of automatically generating CSP rules for web applications, again from the perspective of an administrator. We find that for websites that are well-structured and do not change significantly over time, rules can indeed by generated in a black-box fashion. However, for more complex sites such as those that make use of third-party advertising libraries in their proper site context, policy generation is significantly more difficult.

To summarize, the contributions of this paper are the following:

- We perform the first long-term analysis of CSP adoption in the wild, performing repeated crawls of the Alexa Top 1M over a 16 month period.
- We investigate challenges in adopting CSP, and why it is not deployed to its full extent even when it has been adopted.
- We evaluate the feasibility of both report-only incremental deployment and crawler-based rule generation, and show that each approach has fundamental problems.
- We suggest several avenues for enhancing CSP to ease its adoption.
- We release an open source CSP parsing and manipulation library. [1]

[1] https://github.com/tlauinger/csp-utils

Table 1: The types of directives supported in the current W3C standard CSP 1.0.

Directive	Content Sources
`default-src`	All types, if not otherwise explicitly specified
`script-src`	JavaScript, XSLT
`object-src`	Plugins, such as Flash players
`style-src`	Styles, such as CSS
`img-src`	Images
`media-src`	Video and audio (HTML5)
`frame-src`	Pages displayed inside frames
`font-src`	Font files
`connect-src`	Targets of XMLHttpRequest, WebSockets

2 Content Security Policy

The goal of CSP is to mitigate content injection attacks against web applications directly within the browser [6, 19]. In the following, we describe CSP as it is currently implemented, and briefly discuss both future extensions and the classes of attacks it is intended to prevent.

2.1 Overview of CSP

Content Security Policy is fundamentally a specification for defining policies to control where content can be loaded from, granting significant power to developers to refine the default SOP. Developers or administrators can configure web servers to include `Content-Security-Policy` headers as part of the HTTP responses issued to browsers. CSP-enabled browsers are then responsible for enforcing the policies associated with each resource.

A content security policy consists of a set of directives. Each directive corresponds to a specific type of resource, and specifies the set of origins from which resources of that type may be loaded. Table 1 explains the directive types supported in the current W3C standard CSP 1.0.[2] The scheme and port in source expressions are optional.

CSP also supports wildcards (`*`) for subdomains and the port, and has additional special keywords: `'self'` represents the origin of the resource, while `'none'` represents an empty resource list and prevents any resource of the respective type from being loaded. The `script-src` and `style-src` directives additionally support the `'unsafe-inline'` keyword, which allows inline script or CSS to be included in the HTML document rather than being loaded from an external resource. Finally, `'unsafe-eval'` allows JavaScript to use string evaluation methods such as `eval()` and `setTimeout()`. If not explicitly whitelisted,

[2] The directive `script-src http://seclab.nu:80`, for instance, allows a protected website to load scripts from the host `seclab.nu` via HTTP on port 80, but blocks all scripts from other sources.

CSP disables these special source types because their use is considered to be particularly unsafe. However, changing websites to remove all inline scripts can be a burden on developers, and increase page load latency by introducing additional external resources.

CSP can operate in one of two modes: *enforcement* or *report-only*. In enforcement mode, compatible browsers block resources that violate a policy. In report-only mode, however, browsers do not enforce policies, but rather report violations that would be blocked on the developer console. Additionally, a special CSP directive (`report-uri`) can be used to instruct browsers to send violation reports to the given URI. This feature can be used to learn policies before enabling enforcement, or to monitor for unforeseen changes or attacks against a website. In this paper, we make extensive use of the report-only mode and violation reports to explore various ways to (semi-)automatically generate policies for websites.

CSP has been widely adopted by the browser manufacturers. It is supported by the current versions of almost all major browsers, including some mobile browsers. It is, however, only partially supported by Internet Explorer.

2.2 Deploying CSP

To prevent XSS attacks, disallowing inline scripts and `eval` is the core requirement to benefit from CSP. Inline scripts should be disabled to prevent the browser from inadvertently executing scripts that have been injected into the site. Eval-constructs, often abused to parse JSON strings, can be used directly by an attacker to execute arbitrary code if she controls the data source. While the `unsafe-inline` and `unsafe-eval` options allow this behavior to be enabled, their presence marginalizes the benefit of CSP.

Therefore, for version 1.0 of CSP, inline scripts should be moved to files and `eval` replaced with a safe equivalent for the corresponding task, such as `JSON.parse()` to parse JSON. Furthermore, JavaScript should be hosted on a domain that only serves static files instead of user content. This separation makes it harder for attackers to execute code in the browser. Also, external scripts should be moved to a server controlled by the website owner, reducing trust in third-party servers. The number of whitelisted sources should be kept to a minimum to increase the difficulty of data exfiltration for attackers.

In the current draft version 1.1, additional features have been introduced to safely support inline scripts as well as functionally replace the `X-Frame-Options` header. As these features are subject to change, we do not address them in this work.

2.3 Attacks Outside the Scope of CSP

CSP can prevent general content injection attacks, and in draft version 1.1 subsumes previous mechanisms such as the `X-XSS-Protection` header, which serves the narrow purpose of enabling browser XSS filters. However, it is not intended to address other web attacks such as cross-site request forgery (CSRF). More

fundamentally, CSP describes which content can be loaded by source, but the order of inclusion is out of scope. Hence, even with strict rules and perfect enforcement, out-of-order inclusion can lead to undesired side effects in JavaScript applications [8]. JSONP (JSON with padding) is a mechanism to bypass SOP restrictions by including a script tag from a remote server and specifying a function to be executed once a result becomes available. Hence, whitelisted JSONP sources can be used for calls to arbitrary functions—or, if input for the callback function is not filtered, arbitrary code execution.

3 HTTP Security Headers

In this section, we describe our data collection of HTTP response headers. We collected this data in an effort to understand the landscape of security headers in the wild, particularly in regards to CSP.

3.1 Methodology

To acquire a long-term overview of CSP adoption, we performed weekly crawls of the web starting in December 2012. We crawled the front page of each site in the Alexa Top 1M most frequently visited websites. For every site `x`, we connected to `http://x`, `https://x`, `http://www.x` and `https://www.x`. We counted a site as using a particular header if any of the four responses served that header. However, our crawler only visited the front page of each Alexa entry. Therefore, sites that employ CSP only on subdomains or areas other than the front page were not detected in the crawl.[3] Furthermore, if the CSP rules are generated based on user agent discrimination, the collected data does not hold for all types of browsers visiting the site. We used a Firefox user agent string, updating version information over time.

Description of HTTP Security Headers

To discuss CSP in context, we provide a brief overview of the other security relevant HTTP response headers. Details about them can be found at IETF, W3C, or in the browser specifications.

Platform for Privacy Preferences (P3P) [2]. Websites can use this header to describe their privacy policy. However, it is not supported by major browsers and has not been actively developed for several years. The header is still in use as Internet Explorer blocks third-party cookies by default if no policy is present. P3P is legally binding and has been used in litigation in the past.

DNS Prefetch Control [1]. DNS prefetching is a technique for browsers to reduce latency by resolving referenced hostnames before a user follows a link. This header allows websites to override the default behavior of the browser.

[3] One example is Twitter, which uses CSP for parts of their site, but not the front page.

XSS Protection [3]. This header can be used to enable or disable client-side heuristic XSS filtering. The `reflected-xss` directive of CSP 1.1 is functionally equivalent.

Content Type Options [4]. As the `Content-Type` header is often not set correctly, MIME type sniffing can be used to detect the actual response content type. The `nosniff` directive is the only option available for this header and disables MIME type sniffing, preventing possible type confusion.

Frame Options [9]. This header allows a website to restrict iframing to prevent UI redressing attacks. CSP draft 1.1 includes these features under the `frame-ancestors` directive, and may replace this header.

HTTP Strict Transport Security (HSTS) [5]. By using HSTS, websites can specify that in the future, the browser should only connect to them via a secure connection, thereby preventing SSL stripping.

Cross-Origin Resource Sharing (CORS) [7]. SOP has proven to be an obstacle for modern web applications, and has been worked around by various methods such as JSONP. CORS allows websites to operate outside the limitations of SOP by extending it, while not completely side-stepping it.

3.2 Adoption of HTTP Security Headers

To measure the popularity of CSP in contrast to other security headers, we looked at the HTTP response headers in our weekly crawls, as well as a static snapshot from the end of March 2014. For the static snapshot, we used the entire Alexa Top 1M, breaking down websites by popularity. We used a snapshot of the Top 10K to track the evolution of response headers back to December 2012.

To compare the adoption of security-related headers between different levels of site popularity, we split Table 2 into brackets. From the data, it is apparent that websites that are less popular use CSP less frequently. For instance, among the 100 most popular sites, only two used CSP (2%), while CSP was enabled for only 775 among the 900,000 least popular sites (0.00086%).

Hence, websites that are less popular use CSP less frequently. In contrast, for CORS, header usage was more evenly spread out, with all brackets between 0.7% and 2.6%.

During our crawls, we noticed that Google enabled CSP headers only occasionally. We performed an additional test of `google.com` with 1,000 requests, finding that 0.8% of the responses included CSP headers. While Google had 18 sites in the top 100, none of them issued CSP headers in the crawl of Table 2.

In Figure 1, we track the evolution of security-related headers of the Alexa Top 10K from March 2014 backwards in time to December 2012. P3P was particularly popular; however, the P3P policies served were often invalid, providing only an explanation for why the website did not support it. We observe that CSP is only slowly gaining traction over time. The main contributing factor for the fluctuation of CSP headers in the data is due to Google.

For the hosts in the Top 10K of this crawl, we identified all servers that had sent CSP rules at any point in time during our study. We found 140 sites that did so; 110 of those belonged to Google (79%).

Table 2: Number of websites with security-related HTTP response headers, grouped by intervals of site popularity, for the Alexa Top 1M ranking.

Header / Alexa Rank	$[1-10^2]$	$(10^2-10^3]$	$(10^3-10^4]$	$(10^4-10^5]$	$(10^5-10^6]$
P3P	47	176	849	6,315	79,600
DNS Prefetch Control	1	0	3	40	461
XSS Protection	26	77	269	2,336	43,045
Content Type Options	10	27	172	1,995	42,150
Frame Options	43	165	581	2,747	21,746
HSTS	5	16	83	476	2,475
CORS	1	26	217	1,228	7,149
CSP	2	2	15	57	775
Any security header	66	304	1,623	11,491	132,347

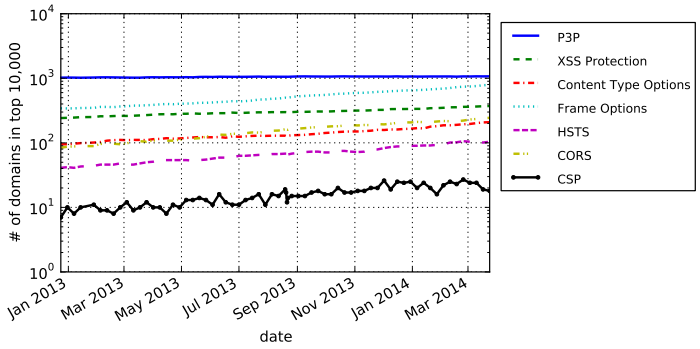

Fig. 1: Popularity of security headers in the Alexa Top 10K.

3.3 Detailed Analysis of CSP Headers

In this part, we describe in detail how websites use CSP, whether they use CSP's reporting feature to learn policies, whether they actively enforce policies, and how effective those policies are in mitigating attacks.

Enforcement vs. Report-Only. During our crawl at the end of March, we found 815 sites in enforcement mode, 35 sites in report-only mode, and no sites that sent both types of headers. Out of the websites in enforcement mode, only 23 collected violation reports.

In the Top 10K, we observed only one site in report-only mode that later switched to enforcement. The Norwegian financial services site `dnb.no` started collecting reports in June 2013, and enabled enforcement in February 2014. Their enforced `default-src` directive consists of 74 sources, including the schemes `chrome-extension`, `chromeinvoke`, and `chromeinvokeimmediate`. Furthermore, `unsafe-inline` and `unsafe-eval` are both enabled. Therefore, this policy appears to provide little benefit over not using CSP at all.

We noticed that several websites use CSP to test for mixed content. Mixed content is the inclusion of unencrypted content into HTTPS sessions, which reduces the benefit of encryption. Google's sampling uses the following report-only policy: `default-src https: data:; options eval-script inline-script; report-uri /gen_204?atyp=csp`. Etsy also samples for mixed content; we found CSP headers in nine out of 2,000 (0.45%) responses. Similarly, `hootsuite.com` tested for mixed content from April 2013 to March 2014 for all responses, but we observed no CSP headers after that.

Types of Sites Using CSP. To further understand the types of websites that use CSP, we looked for similarities in website titles. The largest portion of sites supporting CSP, 417, is due to phpMyAdmin, a PHP-based web application used to manage MySQL databases. phpMyAdmin ships with CSP enabled by default, which allows inline scripts, `eval`, and restricts sources to `'self'`. While this policy does not prevent XSS, data exfiltration is more difficult. These rules can be deployed as the software is fairly static. However, when conducting a search for phpMyAdmin and CSP, we found users having trouble including images when modifying their installations. The general solution offered was to disable CSP in the configuration rather than updating the default policy.

Ironically, on the vendors' demo site `http://demo.phpmyadmin.net/master/`, the operators tried to include Google analytics. While the Google analytics domain is whitelisted using `default-src`, it is not in the `script-src` source list. As specific directives override the `default-src` directive, the script is unintentionally blocked.

We also found 170 OwnCloud instances, which uses CSP by default from version 5.

Prevalence of Unsafe Policies. We identified several patterns in CSP policies that violate deployment best practices as described in Section 2.2. In Table 3, we summarize the observed rules in enforcement over the Alexa Top 1M from March 24th. We split at the 10K rank to discriminate between more popular websites and lower ranking ones. '*' represents either the literal asterisk, or the entire HTTP(S) scheme is whitelisted in one or more of the source lists.

On the majority of sites, `eval` and inline is enabled: eight out of 13 and 11 out of 13 in the Top 10K bracket, 700 out of 802 and 728 out of 802 in the remaining 990,000 sites. This configuration strongly reduces the benefits of CSP for XSS mitigation. Configuring asterisk or a whole scheme as a source in a directive enables data leakage to any host. Six out of 13 and 230 out of 802 websites respectively served such directives. 10 out of 13 sites in the Top 10K bracket had no `report-uri` to collect violation reports. This is surprising as CSP could be used as a warning system.

While CSP in theory can effectively mitigate XSS and data exfiltration, in practice CSP is not deployed in a way that provides these benefits.

Table 3: Overview of enforced policies.

Feature / Alexa Rank	$[1-10^4]$	$(10^4-10^6]$
unsafe-eval	8	700
unsafe-inline	11	728
script-src 'self'	12	789
no report-uri	10	782
#script-src > 10	2	33
*** as source**	6	230
Median #directives	6	4
Median #script sources	4	1
# CSP Policies	13	802

3.4 Conclusions

While some sites use CSP as an additional layer of protection against content injection, CSP is not yet widely adopted. Furthermore, the rules observed in the wild do not leverage the full benefits of CSP. The majority of CSP-enabled websites were installations of phpMyAdmin, which ships with a weak default policy. Other recent security headers have gained far more traction than CSP, presumably due to their relative ease of deployment. That only one site in the Alexa Top 10K switched from report-only mode to enforcement during our measurement suggests that CSP rules cannot be easily derived from collected reports. It could potentially help adoption if policies could be generated in an automated, or semi-automated, fashion.

4 CSP Violation Reports

Web browsers compatible with CSP can be configured to report back to the website whenever an activity, whether carried out or blocked, violates the site's policy. This is meant as a debugging mechanism for web operators, both to develop policies from scratch, and to be informed when an existing policy needs to be updated. Starting with a "deny all" policy in report-only mode, operators can collect information about all resources that need to be whitelisted in order for the site to function, compile a corresponding policy, and eventually switch to enforcement mode. We applied this approach to four websites and analyzed the reports that we received, gaining unexpected insights into the web ecosystem.

4.1 Background

CSP includes an optional `report-uri` directive that allows website operators to specify a sink for violation reports. It is supported in both report-only and enforcement mode of CSP. As an illustration, consider the following policy: `img-src 'none'; report-uri /sink.cgi`. When a user visits the URL `http://seclab.nu/test.html` and that page includes the image resource `http://se`

clab.nu/pic.gif, the browser would send a report similar to the following one: {"blocked-uri": "http://seclab.nu/pic.gif", "violated-directive": "img-src 'none'", "document-uri": "http://seclab.nu/test.html", ...}. From this report, the developer can infer that the policy entry `img-src http://seclab.nu` should be added to the policy.

4.2 Methodology

We deployed CSP on four of our own websites: two personal pages, an institutional page, and a popular analysis service. The policies we used specified empty resource lists for all supported directive types—that is, any browser activity covered by CSP was explicitly forbidden and should generate a report. We deployed the policies in report-only mode to not interfere with the normal operation of the site. Besides the additional CSP headers, the sites were not modified in any way.

During our analysis, we observed that the formats of reports sent by different browser versions varied slightly. Older Firefox versions, for instance, explicitly stated when a violation was due to the special cases `'unsafe-inline'` or `'unsafe-eval'` for script and style directives, as opposed to violations based on a resource URI. All recent versions of browsers, however, reported only an empty `blocked-uri` instead. Unfortunately, this format did not allow us to distinguish between `'unsafe-inline'` and `'unsafe-eval'` script violations.

In order to work around this issue, we leveraged the fact that recent browser versions supported multiple CSP headers in parallel. That is, in addition to the *regular* policy discussed above that captured any CSP event, we added two more policies that caused reports only for *eval* and *inline* violations, respectively:

```
   default-src *; script-src * 'unsafe-inline';
style-src * 'unsafe-inline'; report-uri /sink.cgi?type=eval
   default-src *; script-src * 'unsafe-eval';
style-src *; report-uri /sink.cgi?type=inline
```

We deployed all three policies and distinguished the reports we received using the type parameter in the report URI. We removed duplicate eval and inline violations that were reported for the *regular* policy (30 % on site D). Furthermore we removed some violations reported for the *eval* and *inline* policies that were in fact no eval or inline violations (1.8 % on site D). Those were triggered by a bug in older Firefox versions that did not properly execute multiple policies in parallel. Since newer Firefox versions were not affected, the user agent distributions of the original and the filtered data set were very similar. Table 4 shows the number of reports retained in the filtered data set, which is the basis for the following discussion.

From each report, we derive a policy entry that whitelists the respective violation. We extract the type, such as `img-src`, from the `violated-directive`. For *regular* violations, we append the scheme, host name and port from the `blocked-uri`, such as `http://seclab.nu`. For *inline* or *eval* violations, we append `'unsafe-inline'` or `'unsafe-eval'`. We generate a single policy per site by combining all entries and set `default-src 'none'` to block everything else.

Table 4: Overview of the CSP violation report data sets received from our websites in early 2014, after removing inconsistent reports.

Site	A	B	C	D
Type	personal	personal	institutional	service
# Reports	1.1 K	21.8 K	48.0 K	7.1 M
Median Reports/Day	9	671	2.1 K	348.5 K
# IP Addresses	78	1.6 K	1.2 K	14.4 K
Median Reports/Addr.	7	7	28	85
% Reports/Browser				
Chrome (mobile, derivatives)	46.6 (+5.4)	59.3 (+8.3)	54.3 (+3.7)	61.0 (+2.3)
Firefox (mobile, derivatives)	23.8 (+0.5)	22.2 (+0.6)	30.1 (+0.5)	30.3 (+0.2)
Safari (mobile)	5.7 (+2.3)	2.3 (+3.5)	4.1 (+3.7)	1.5 (+0.5)
Opera	0.5	0.3	0.6	1.9
Googlebot	15.1	3.1	2.0	2.1

Our approach is to generate one single policy that is general enough to cover the entire protected site. Such a site-wide policy is easier to generate than individual policies, since any similarity between pages on the same site reduces the number of violation reports necessary to generate a policy. Furthermore, site-wide policies are easier to configure; a site-wide reverse proxy could insert a static policy into HTTP responses without the need to change application code.

4.3 Results

Table 5 summarizes the policies we generated for each of our sites. We verified manually each entry in the policies and found that many of the whitelisted resources were not actually intended to be included in the websites. The policy generated for site A, for instance, is `default-src 'none'; frame-src https://srv.mzcdn.com; img-src 'self' data: http://1.2.3.11; object-src http://www.ajaxcdn.org; script-src 'unsafe-eval' 'unsafe-inline' http://ajax.googleapis.com http://f.ssfiles.com http://i.bestoffersjs.info http://srv.mzcdn.com http://www.superfish.com https://www.superfish.com; style-src 'unsafe-inline'`. Yet, site A was entirely static and did not contain any script at all. The correct policy for site A would have been `default-src 'none'; img-src 'self' data:; style-src 'unsafe-inline'`. In other words, only 21 % of the policy entries generated from the received reports were legitimate.

On site D, only 2 % of the policy entries were legitimate. Furthermore, many of the legitimate entries simply enumerated all the alternative domain names of the same site (e.g., with or without the `www` subdomain), or they were due to the same resource being loaded over HTTP or HTTPS. When disregarding these details to allow for a fairer comparison, as noted in brackets in the table, the percentage of legitimate policy entries drops to only 0.8 % on site D.

Table 5: Length of policies when whitelisting all violations from the report data set (a), and with an additional filter for URL schemes of browser extensions (b). Most of the policy entries correspond to injected resources; only few are intended to be included. (In brackets, the number of unique policy entries when disregarding the protocol HTTP(S) or alternative domains, such as the www subdomain.)

Site	A	B	C	D
# Entries (a)	14	221	226	1,113
# Entries, extension filter (b)	14	212	215	1,090
Correct Subset	3 (3)	14 (9)	38 (13)	22 (9)

Table 6: Most frequent Chrome extensions observed at site D.

Name	**# Reports**
AdBlock	38 K
AdBlock Plus	29 K
Grooveshark Downloader	9.5 K
ScriptSafe	8.8 K
DoNotTrackMe	8.2 K

Reasons for invalid policy entries. We identified a number of reasons why web browsers sent CSP violation reports for resources that did not exist in the original websites. Many of these reports appeared to be caused by browser extensions that modified the DOM of the page by injecting additional resources such as scripts or images. We observed extensions for blocking advertisements, extensions injecting advertisements, price comparison toolbars, an anti-virus scanner, a notetaking plugin, and even a BitTorrent browser extension. We could automatically identify some browser extensions based on violation reports because they attempted to load resource URIs that contained the `chrome-extension` or `safari-extension` schemes followed by the unique identifier of the extension. AdBlock and AdBlock Plus were the most frequent extensions for the Chrome browser (Table 6), while the most frequent Safari extension was Evernote. Yet, automatically removing these reports (and a few other unexpected schemes, such as `about` and `view-source`) accounted for fewer than 5 % of all incorrect policy entries, as shown in the second row of Table 5. The remaining browser extensions exhibited no such uniquely distinguishing features, often injecting libraries that are used not only in browser extensions but also in many websites, such as Ajax tools, Google Analytics, and resources from large content distribution networks.

When browsers send violation reports for modifications due to browser extensions, the reverse conclusion is that websites enforcing CSP can cause browser extensions to stop functioning. Some browser extensions thus intercept CSP headers and modify them in order to whitelist their own resources or disable CSP. We observed reports caused by one such extension, which were sent because the modification resulted in a semantic error. We cannot quantify how

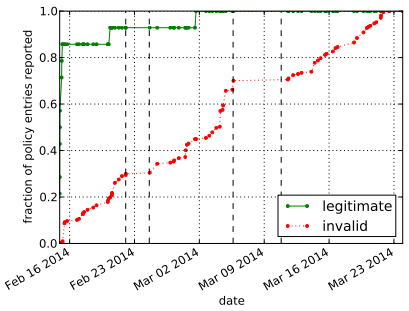

Fig. 2: Fraction of new policy entries discovered over time on site B (measurement inactive during the dashed intervals). It can take some time until all legitimate resources have been accessed at least once; in the meantime, many injected resources are reported.

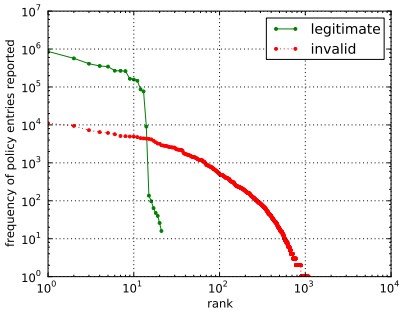

Fig. 3: Frequency of legitimate and invalid violations being reported on site D. Some injected resources occurred orders of magnitude more often than legitimate resources.

often such modifications were successful as they are not observable with our methodology.

In addition to browser extensions, "in-flight" modification of pages by ISPs or web applications such as anonymity proxies can also cause violation reports. The image loaded from `1.2.3.11` in the example above appeared to be injected by a mobile Internet provider. These examples illustrate that even when CSP violations due to browser extensions were filtered (or not reported by the browsers), other non-attack scenarios can still cause websites to receive spurious reports. Administrators who plan to generate a policy from reports submitted by their visitors' web browsers may need to manually verify a large number of policy entries in order to avoid accidentally whitelisting resources injected by browser extensions or ISPs (let alone attackers).

Time delay until a policy can be generated. On site B, it took around two weeks to receive at least one report for each valid policy entry. The last resource that was discovered was an embedded YouTube video. Another resource that was discovered relatively late was an image loaded over HTTPS instead of HTTP; all other valid policy entries could be generated within the first two days of the measurement. For the other sites, the durations were similar. In practice we expect these numbers to vary, thus website operators will need some prior knowledge about the resources used on their website so that they can decide when it is safe to switch from report-only to enforcement mode without causing any disruption. Operators could therefore be tempted to run the observation period for as long as possible in order to minimize the risk of not receiving reports for legitimate resources. However, as Figure 2 shows, the rate of newly observed, invalid policy entries remained relatively constant over time, suggesting

that longer measurement periods can significantly increase the number of policy entries an operator needs to verify manually.

Report frequency as a (poor) distinguishing feature. Only about 4 % of all reports received on site D during our measurement resulted in an invalid policy entry. Hence, one might attempt to use the frequency of a report as an indicator for its validity. However, this approach would be problematic for two reasons. First, an attacker can easily influence the frequency distribution observed by the website by submitting forged reports. Second, even in the absence of attacks, resources injected into websites can be so popular that they cause reports more often than some legitimate, but infrequently accessed, resources.

Figure 3 visualizes this phenomenon. The most frequently injected resource (a script loaded from `superfish.com` for price comparison) was reported more than 22,000 times. In contrast, `connect-src 'self'`, which is used by a progress meter on the site, was reported only 9,000 times, and reports corresponding to alternative domain names of site D were received even less frequently.

4.4 Conclusions

Websites small and large observe CSP violation reports for injected resources. Even in the absence of ostensibly malicious activity, which we did not observe, the high number of injected resources complicates the process of generating a viable policy from the received reports. At the moment, this task is mostly a tedious and, from our own experience, error-prone manual process. As a semi-automated approach to filtering reports, it might be possible to generate signatures for the most common browser extensions, either manually or by leveraging the fact that an installed browser extension usually causes several violations to co-occur (based on time, IP address, and user agent signature). These signatures could be shared with the community and could be used to reduce the number of reports that need to be verified manually.

5 Semi-Automated Policy Generation

An alternative approach to generating a policy from appropriately filtered and verified reports submitted by visitors is to make use of trustworthy reports only. In order to explore this approach further, we developed a proof-of-concept web crawler that generates violation reports in a controlled environment.

5.1 Methodology

Our crawler is implemented as an extension for the Chromium browser based on Site Spider, Mark II. The crawler follows at most 500 internal links on the main domain of the crawled site in a non-randomized breadth-first search. After navigating to a page, the crawler pauses for 2.5 s to load all resources of a document such as images, scripts, and external pages displayed in frames. The

browser accesses the web through an instance of the Squid web proxy with an ICAP module. The proxy inserts the CSP report-only headers described in Section 4.2 and collects the resulting reports. The proxy also intercepts encrypted SSL traffic.

After crawling a site, we discarded all reports that did not match the site's main domain. These reports referred to external documents loaded in a frame and were not necessary to generate a policy for the main document. (In CSP, a document's policy does not transitively apply to nested documents loaded inside a frame.) From the remaining reports, we generated a policy as in Section 4.2.

The crawler should be considered a proof-of-concept to explore the feasibility of automatically generating policies for websites. By following only hypertext links, the crawler cannot detect violations that conditionally occur after load-time, such as clicking the "play" button in a Flash movie, or triggering JavaScript-related events. We leave ways to increase the crawler's coverage to future work.

As a potentially more targeted alternative to automated crawling, we also manually browsed websites in a fresh browser instance and used the proxy to collect reports. This process included no feedback. The goal was to cover all areas of the site and trigger as many different violations as possible by specifically exercising functionality implemented in JavaScript or browser plugins.

5.2 Evaluation

The question of whether semi-automated policy generation for websites is a suitable approach—without requiring modifications to the sites—depends on two opposing goals. First, the generated policy must not break the site. A policy generation mechanism must discover all resources being included by a site, or a superset thereof. Second, the generated policy should be as narrow as possible in order to provide the maximum safety gain. Unnecessary resources should not be allowed by the policy, and unsafe mechanisms should not be used. In the first part of this evaluation, we compare methods of collecting reports for policy generation on sites where we know that a sound policy exists. In the second part, we explore how well different site architectures work with CSP; that is, whether a sensible policy can be deployed without changing the sites.

Crawling and manual browsing of our own sites. From the reports submitted by visitors' web browsers in Section 4.3, we know that stable policies exist for our own four sites. Indeed, the sets of policy entries generated by crawling and manual browsing as shown in the upper part of Table 7 overlap, and only a few entries were found by only one method. Especially when disregarding differences due to alternative domain names and HTTP(S), both methods performed similarly. However, neither method was perfect. The crawler discovered resources in a rather hidden portion of site B that manual browsing did not uncover. On site D, in turn, manual browsing discovered a resource inclusion that the crawler was not able to find, which was due to exercising JavaScript code when submitting

Table 7: Overlap between the sets of policy entries generated by the crawler, through manual browsing and from user-submitted reports. (In brackets, the number of common/different policy entries when disregarding alternative domain names or HTTP(S).) No method was fully reliable.

Site	A	B	C	D
crawler only	0 (0)	8 (8)	0 (0)	0 (0)
both	3 (3)	12 (9)	12 (10)	8 (7)
manual only	0 (0)	2 (0)	1 (0)	9 (2)
crawler only	0 (0)	9 (9)	0 (0)	0 (0)
both	3 (3)	11 (8)	12 (10)	8 (7)
valid user reports only	0 (0)	3 (1)	26 (3)	14 (2)
manual only	0 (0)	3 (2)	0 (0)	2 (0)
both	3 (3)	11 (7)	13 (10)	15 (9)
valid user reports only	0 (0)	3 (2)	25 (3)	7 (0)

content to the site. The policy entries generated from valid user-submitted reports were always a strict superset of those derived from crawling and browsing (as shown in the lower two-thirds of the table), except for site B where we found that a technical mistake had prevented CSP headers from being sent to users in a small portion of the site. We conclude that the crawler and manual browsing techniques need more refinement before they can fully replace user-submitted reports. Since both techniques are complementary, combining them could prove useful to increase coverage.

Crawling and manual browsing of CSP-enabled sites. In order to compare our crawler-generated policies to real-world policies, we generated policies for large public websites that deployed CSP in enforcement mode. As a case study, we provide more detail for Facebook and GitHub.

Our crawl included the public portion of Facebook as well as authenticated sessions. The policy generated by the crawler was a subset of Facebook's actual policy. It listed the specific subdomains of Content Distribution Networks (CDNs) observed during the crawl, whereas Facebook whitelisted all CDN subdomains with a wildcard. Furthermore, while Facebook's policy restricted only `script-src` and `connect-src`, the crawler also generated entries for `img-src`, for instance. Both issues could cause unobserved (but legitimate) behavior to be blocked and illustrate that automatically generated policies are likely to require fine-tuning using domain knowledge before they can be deployed.

On GitHub, the crawler discovered all whitelisted resources of the original policy (which did not use any wildcards, and restricted only `script-src`, `style-src`, and `object-src`). The crawler generated additional entries that were not part of GitHub's policy. Upon manual verification, we found that some resources included in GitHub's blog were not loaded due to missing policy entries. This finding illustrates the importance of monitoring enforced policies when web-

sites evolve; regular crawls of a website could be a useful tool to help detect such changes.

Influence of design choices on CSP. Architectural features of a site can influence whether it is possible to deploy a meaningful policy without changing the site. Our crawls of Twitter, for instance, found a small, stable set of policy entries, while additional manual browsing discovered only one additional policy entry. Most of the resources were internal. Multimedia content included in tweets, for instance, was loaded from internal subdomains with constant names. Such an architecture makes it relatively convenient to deploy CSP without major changes. Indeed, Twitter used CSP in some subdirectories and subdomains.

Other sites such as Amazon, Google, and YouTube dynamically used explicitly named subdomains of CDNs such as `mt{2,3}.google.com`, similarly to Facebook. These subdomains appeared to be used for load balancing and could therefore be considered equivalent from a security point of view. Our crawler was not able to enumerate all these subdomains, but post-processing of the policy such as using a wildcard `*.google.com` could address the issue. A drawback of this approach is that sites such as Amazon that use external CDNs would also be whitelisting other customers' subdomains. A cleaner approach would be to use static domain names at the web application layer and address load balancing transparently at lower layers, as appears to be done by Twitter.

In the examples above, it was possible to compensate for some degree of variability in the sites by broadening the generated policy because the variability was systematic. On certain types of sites such as blogs where users are allowed to include externally hosted content, this may not be possible. The policy used by GitHub shows a possible compromise in such situations: the site allowed images to be loaded from any source and restricted only more sensitive resource types such as scripts and plugins.

Stability of policies. A requirement to successfully deploy an enforceable policy is to predict at policy generation time the external resources that will be included when a page is rendered in a browser. A particularly unpredictable type of external content is advertising. The exact advertisement shown to a user is typically determined dynamically while the page is loading. Dynamic advertising can involve techniques such as Real-Time Bidding (RTB), where the opportunity to display an advertisement to a visitor is auctioned off in real-time, and further dynamic activity such as cookie matching between the host website and the winner of the auction. There are routinely tens to hundreds of potential bidders in RTB [16], each of whom represent a large number of actual advertisers.

In order to better understand how this dynamic activity can be reconciled with the more static requirements of CSP, we performed repeated crawls of two large websites with dynamic advertisements and counted how many new policy entries we discovered in each subsequent crawl (Table 8). Twitter, which we crawled as a control data point, remained stable and resulted in exactly the same policy in all crawls. On the BBC, the crawler discovered between 13 and 61 new policy entries in each of the follow-up crawls; the vast majority of them

Table 8: Additional policy entries discovered in repeated crawls. The high variability due to advertising on the BBC precludes CSP from being used effectively. CNN's way of including advertisement results in a relatively stable (and enforceable) policy.

Crawl number	1	2	3	4	5
BBC	285	+34	+61	+13	+53
CNN	116	+4	+2	+ 1	+1
Twitter	20	+0	+0		

were scripts or other content related to advertising. On CNN, the follow-up crawls discovered only between one and four new policy entries, and only one was unambiguously related to advertising. Since both sites displayed comparable types and amounts of advertisements, the differences must be due to the way advertising was implemented. Indeed, the BBC loaded all advertisement-related resources, including RTB scripts, tracking code, and the final image being displayed, directly into the body of the main document. It would be very challenging to deploy CSP in such a scenario because it seems unfeasible to proactively determine any resource that could potentially be loaded. In contrast, CNN isolated advertisements from the main document by loading them as a separate document displayed inside an embedded frame.

This decoupling significantly eases the deployment of CSP because the main document's policy does not transitively apply to the document inside the frame. In such a deployment, it would be possible to enforce a rather strict policy for the main document and a much more permissive policy for the embedded advertisement document (or none at all). The SOP as well as the HTML5 frame sandboxing mechanism can be used to ensure that untrustworthy scripts in the frame cannot access or modify the main document.

Safety of policies. To assess whether policies generated for a site represent any significant reduction in exposure to attacks, we checked whether the policies included "unsafe" CSP features—that is, inline script or style and calls to `eval`. Among our own sites that included JavaScript, only site B did not require `eval` privileges. Amazon, the BBC, CNN, Facebook, Google, the Huffington Post, and YouTube required all three privileges; Twitter needed inline script and style, and GitHub only inline style. These requirements may be due to code on the sites or in external libraries they include. Even though allowing inline script and `eval` reduces the effectiveness of CSP against XSS attacks, by restricting where external resources may be loaded from, CSP could still make it more difficult for attackers to include custom content such as images or to exfiltrate stolen data.

5.3 Conclusions

Neither naïve crawling nor manual browsing alone are sufficient methods to generate a content security policy for a website. In our approach, a certain amount

of fine-tuning of generated policies is required for all but the simplest sites. Advanced crawling, or applying machine learning to the generated policies, could reduce the importance of manual tweaks. More complex sites may be able to use only a subset of CSP unless they adjust their architecture. Once a policy has been deployed, an additional challenge is to ensure that it is always up to date.

6 Discussion

We saw that only few websites use CSP, and those that do use it do not leverage its full benefits. For this section, we reached out to security engineers behind larger CSP deployments and summarize key points. Furthermore, we suggest several ways in which CSP adoption could be improved.

6.1 Discussions with Security Engineers

To understand implementation decisions behind real-world CSP deployments, we talked to security engineers responsible for three of the measured websites. Out of these sites, two were in the Alexa Top 200, and one in the Top 5,000. The websites used CSP in enforcement mode or report-only for testing. We summarize the key observations in an anonymized fashion.

Websites prefer not to remove inline script. While inline script can be completely removed from websites, this represents significant effort and can lead to more roundtrips when loading the page. Engineers hope to address this issue with the nonce and hash features of CSP draft version 1.1. Hash might be more promising because documents can be distributed over CDNs more easily, whereas for nonce a new document would need to be generated for each response.

Risk of breaking functionality. This was manifested by disabling CSP for browser versions with problematic CSP implementations, including Chrome and Firefox. A website that is secure but not usable can harm business more than occasional XSS. For the future, reliable implementations of CSP in browsers are anticipated.

Enforcement over extensions is considered a bug. CSP rule enforcement can break the functionality of browser extensions. A workaround is to whitelist popular sources. However, extensions could still be unintentionally restricted. A modification of browser implementations or the standard to not enforce rules over extensions could solve this.

6.2 Suggested Improvements

We briefly summarize approaches that could help the adoption of CSP and increase its security benefits when deployed.

Ads should be integrated into iframes instead of the main site. Instead of whitelisting all possible ad networks or developing a mechanism for recursive policy adoption, ads should be moved into sandboxed iframes. This allows the main site to be protected with an effective policy, while the iframe can be more permissive, but isolated. Conflating both the site proper and ads in the

same context is not necessary, since information required by ads can be passed via `postMessage` cross-window communication. However, while not widely available, alternatives such as Security Style Sheets [14] have been proposed that would allow for such separation without moving content to iframes.

More web applications and frameworks should adopt CSP. Introducing CSP to programs that are deployed widely can have a higher impact on the overall security of the web as compared to individual websites adopting CSP. As examples, phpMyAdmin and OwnCloud have adopted CSP, and Django can be configured with CSP. Most desirable would be the introduction of CSP to web frameworks, which could drastically improve adoption of CSP and the safety of the web.

Browsers should not enforce CSP on extensions. As discussed in Section 4, enforcing policies on browser extensions generates many unexpected reports for websites. Websites should not be forced to whitelist extensions since the number of extensions and third-party resources included by those extensions is theoretically unbounded and cannot be predicted by application developers. Furthermore, CSP in its current form is not an adequate mechanism for websites to block potentially undesired extensions and should not be used as such.

7 Related Work

CSP was proposed by Stamm et al. [19], who provided the first implementation in the Firefox browser. Subsequently, CSP became a W3C standard [6] and was adopted by most major browsers. Other publications have addressed limitations of CSP and suggested extensions or modifications to the standard. For instance, Soel et al. [18] proposed an extension of CSP to address shortcomings in `postMessage` origin handling.

CSP was the first widely deployed browser policy framework to mitigate content injection attacks. However, it was not the first one to be suggested. SOMA (Same Origin Mutual Approval) [15] reduces the impact of XSS and CSRF by controlling information flows. Website operators need to approve content sources in a manifest file, as well as content providers need to approve websites to include their content. BEEP [11] can prevent XSS attacks with a whitelist approach for JavaScript and a DOM sandbox for possibly malicious user content. Script tags are whitelisted by hash, a feature that is also proposed in the 1.1 draft of CSP. BLUEPRINT [20] enforces restrictions on the document parse tree in the browser. Web application server components make parsing decisions and transport the DOM structure to the client. By enforcing a consistent document structure, misuse of browser rendering quirks is eliminated. CONSCRIPT [12] supports a variety of policies for JavaScript enforcement, which can be generated automatically. Static policy generation is supported for Script#, a Microsoft tool that generates JavaScript from C# code, as well as a dynamic training mode for other platforms. Weinberger et al. [21] performed an evaluation of browser-side policy enforcement systems. They concluded that security policies for HTML should be a central mechanism for preventing content injection attacks, but need more research to become effective. We performed the first study on CSP

adoption in the wild, analyzing how usage has evolved in the past year on the most popular websites. Also, we investigate how report-only mode can be used to devise policies, and whether those are effective.

Currently, inline scripts are as popular with websites as they are bad for the effectiveness of CSP to prevent XSS. Bugzilla and HotCRP required substantial changes to support CSP [21], while `addons.mozilla.org` required an effort of several hours [19]. Previous work performed automatic rewriting of .NET applications to better support CSP [10]. Recent changes to the CSP draft, such as nonce and hash whitelisting of scripts, represent an approach that relieves developers of removing inline scripts while allowing for control over code. Trust relationships in external script sources have been analyzed by Nikiforakis et al. [13]. 88% of the Alexa Top 10K most visited websites included scripts from remote sources, and the most popular single library was included from 68% of the sites. An outlook on the possible future of web vulnerabilities has been summarized by Zalewski [8]. While CSP addresses a wide range of vulnerabilities, it can not prevent out-of-order execution of scripts, code reuse through JSONP interfaces, and others.

8 Conclusion

In this paper, we have presented the results of a long-term study on CSP as it is deployed on the web. We have found that CSP adoption significantly lags other web security mechanisms, and that even when it has been adopted by a site, it is often deployed in a way that negates its theoretical benefits for preventing content injection and data exfiltration attacks.

In addition, by enabling CSP at four sites, we observed that it is difficult for third parties to deploy CSP, either through incremental deployment using report-only mode or through web application crawling to semi-automatically generate policies.

CSP clearly holds great promise as a web security standard, but we can only conclude that it is difficult for most sites to deploy it to its full potential in its current form. It is our hope that the improvements we suggest here, as well as upcoming features of the 1.1 draft, will allow site operators and developers to make effective use of content security policies and result in a safer web ecosystem.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under grant N00014-12-1-0165. We would like to thank our shepherd Anil Somayaji and the anonymous reviewers for their helpful comments. Furthermore, we thank Collin Mulliner and Clemens Kolbitsch for their help in data collection.

References

1. DNS Prefetching - The Chromium Projects, `http://www.chromium.org/developers/design-documents/dns-prefetching`

2. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification (2002), http://www.w3.org/TR/P3P/
3. IE8 Security Part IV: The XSS Filter (2008), http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx
4. IE8 Security Part V: Comprehensive Protection (2008), http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx
5. RFC 6797 - HTTP Strict Transport Security (HSTS) (2012), http://tools.ietf.org/html/rfc6797
6. Content Security Policy 1.1 (2013), https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html
7. Cross-Origin Resource Sharing, W3C Candidate Recommendation 29 January 2013 (2013), http://www.w3.org/TR/cors/
8. Postcards from the post-XSS world (2013), http://lcamtuf.coredump.cx/postxss/
9. RFC 7034 - HTTP Header Field X-Frame-Options (2013), http://tools.ietf.org/html/rfc7034
10. Doupé, A., Cui, W., Jakubowski, M.H., Peinado, M., Kruegel, C., Vigna, G.: deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In: ACM Conference on Computer and Communications Security (CCS) (2013)
11. Jim, T., Swamy, N., Hicks, M.: Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In: International Conference on World Wide Web (WWW) (2007)
12. Meyerovich, L.A., Livshits, B.: ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In: IEEE Symposium on Security and Privacy (Oakland) (2010)
13. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In: ACM Conference on Computer and Communications Security (CCS) (2012)
14. Oda, T., Somayaji, A.: Enhancing Web Page Security with Security Style Sheets. Carleton University (2011)
15. Oda, T., Wurster, G., van Oorschot, P.C., Somayaji, A.: SOMA: Mutual Approval for Included Content in Web Pages. In: ACM Conference on Computer and Communications Security (CCS) (2008)
16. Olejnik, L., Tran, M.D., Castelluccia, C.: Selling Off Privacy at Auction. In: ISOC Network and Distributed System Security Symposium (NDSS) (2014)
17. Samuel, M., Saxena, P., Song, D.: Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In: ACM Conference on Computer and Communications Security (CCS) (2011)
18. Son, S., Shmatikov, V.: The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In: ISOC Network and Distributed System Security Symposium (NDSS) (2013)
19. Stamm, S., Sterne, B., Markham, G.: Reining in the Web with Content Security Policy. In: International Conference on World Wide Web (WWW) (2010)
20. Ter Louw, M., Venkatakrishnan, V.: BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In: IEEE Symposium on Security and Privacy (Oakland) (2009)
21. Weinberger, J., Barth, A., Song, D.: Towards Client-side HTML Security Policies. In: Workshop on Hot Topics on Security (HotSec) (2011)