Content Security Problems?

Evaluating the Effectiveness of Content Security Policy in the Wild

Stefano Calzavara
Università Ca' Foscari
calzavara@dais.unive.it

Alvise Rabitti
Università Ca' Foscari
alvise.rabitti@unive.it

Michele Bugliesi
Università Ca' Foscari
michele.bugliesi@unive.it

ABSTRACT

Content Security Policy (CSP) is an emerging W3C standard introduced to mitigate the impact of content injection vulnerabilities on websites. We perform a systematic, large-scale analysis of four key aspects that impact on the effectiveness of CSP: browser support, website adoption, correct configuration and constant maintenance. While browser support is largely satisfactory, with the exception of few notable issues, our analysis unveils several shortcomings relative to the other three aspects. CSP appears to have a rather limited deployment as yet and, more crucially, existing policies exhibit a number of weaknesses and misconfiguration errors. Moreover, content security policies are not regularly updated to ban insecure practices and remove unintended security violations. We argue that many of these problems can be fixed by better exploiting the monitoring facilities of CSP, while other issues deserve additional research, being more rooted into the CSP design.

CCS Concepts

•**Security and privacy** → **Browser security;**

Keywords

Content Security Policy; Measurement

1. INTRODUCTION

The same-origin policy (SOP) is the baseline defence implemented in web browsers to provide confidentiality and integrity guarantees for contents provided by unrelated websites. Under the SOP, data from `https://www.mybank.com` is only accessible to scripts from `https://www.mybank.com` and shielded from read or write attempts by scripts from other web origins. Though apparently secure, it is well-known that the SOP can be bypassed by *content injection* attacks. In these attacks, attacker-controlled contents are injected in benign web pages and become indistinguishable from legitimate contents, thus inheriting their privileges.

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978338

The most effective way to defend against content injection is *sanitization* of inputs, preventing dangerous contents like (unintended) script tags from entering benign web pages. Unfortunately, sanitization is often difficult to get right and content injections are still pervasive on the Web [6]. This motivated the development of complementary in-depth defence mechanisms aimed at mitigating the effects of a successful content injection [14, 18, 19, 23, 11]. Among these, Content Security Policy (CSP) is by far the most popular and well-established solution, being standardized by the W3C and supported by all major web browsers [23, 3].

CSP is a language for defining restrictions on the functionality of web pages, ideally to limit their capabilities to the least set of privileges they need to work correctly. Most notably, CSP mitigates the dangers of a successful content injection by disallowing the execution of inline scripts and by banning a few dangerous functions used for turning strings into code, like the infamous `eval`. Moreover, CSP allows the specification of constraints on content inclusion based on a white-listing mechanism, whereby different content types, like images or scripts, are bound to the sole set of origins which are allowed to supply those contents. This way, injected markup elements can only be abused to load contents from white-listed web origins.

CSP is a client-server defence mechanism: policies are specified by web developers using HTTP(S) headers or meta elements in HTML pages, while they are enforced at the browser side.

1.1 Research Goals

Our main goal in the present paper is to assess the state of the art in the use and effectiveness of CSP as a security mechanism for websites against content injection attacks. We start our analysis by measuring the extent to which CSP is supported by modern browsers and adopted by websites. We then analyse two further aspects that provide insight on how web developers use the mechanisms provided by CSP. Specifically, we look at configuration correctness and maintenance. Indeed, the effectiveness of CSP crucially depends on the ability of web developers to configure it so as to make websites resistant to attacks, while at the same time preserving their full functionality. Similarly, modern websites are highly dynamic and constantly evolving, hence CSP security is crucially dependent on the ability of web developers to keep their policies up-to-date.

1.2 Contributions and Roadmap

We briefly anticipate the techniques used in our analysis and the main findings and results:

1. *browser support*: we design a set of experiments to test the browser implementations of the CSP specification. We run the experiments in all major web browsers, including their mobile variants. We report on the outcome of the experiments, highlighting the cases where at least one web browser does not behave as expected and discussing their security import. Our investigation reveals a dangerous behaviour of Microsoft Edge and a subtle quirk in all browser implementations, which deserves a careful security analysis (Section 3);

2. *website adoption*: we collect the CSP headers from the Alexa Top 1M websites [1] and we analyse them to shed light on the current state of the CSP deployment, which turns out to be quite limited. We also investigate which features of CSP are popular among web developers and which ones are largely unused, identifying many common bad practices (Section 4);

3. *correct configuration*: we identify three common classes of errors made by web developers when writing content security policies. We provide representative examples of errors and we discuss their security and usability import (Section 5). We then perform a systematic security analysis of existing content security policies. In particular, we identify sufficient conditions on policies which allow an attacker exploiting a content injection to run arbitrary malicious code in CSP-protected web pages. Based on this, we show that the very large majority of the websites we surveyed (92.4%) deploy content security policies which do not provide robust defences against code injection (Section 6);

4. *constant maintenance*: we repeat the crawling of the Alexa Top 1M for 14 weeks, automatically collecting both CSP headers and violations to the policies contained therein. We identify websites committing to CSP or abdicating from it during this timespan and we analyse how existing policies changed during the 14 weeks, discussing good and bad practices observed in the wild. Finally, we investigate correlations between changes to enforced policies and policy violations, concluding that content security policies change less frequently than needed (Section 7).

We present our perspective on the main findings in the paper in Section 8. Our take is that many of the problems we found can be fixed by better exploiting the monitoring facilities of CSP, while other issues deserve more research, being more rooted into the CSP design. Before this study, two research papers about the state of the CSP deployment have been published [28, 21]: we refer to the related work section (Section 9) for an in-depth comparison with them.

2. CONTENT SECURITY POLICY

2.1 Overview

A content security policy is a list of *directives*, restricting content inclusion for web pages by means of a white-listing mechanism. Directives bind content types to lists of sources from which a CSP-protected web page is allowed to include resources of that specific type. For instance, the directive `img-src https://a.com` enforces that a web page can only load images from the host `a.com` via the HTTPS protocol. Content security policies are specified inside HTTP(S) headers or using meta elements in the body of an HTML page. Policy enforcement is performed by CSP-enabled web browsers on a per-page basis: different pages of the same website may specify different content security policies.

Table 1 reports selected directive types available in CSP: if a content security policy does not include a directive for a given content type, the `default-src` directive applies to it.

Table 1: Selected CSP Directives

Directive	*Restricted Contents*
`img-src`	Images
`script-src`	JavaScript, XSLT
`style-src`	Stylesheets (CSS)
`connect-src`	Targets of XMLHttpRequest
`default-src`	Contents w/o explicit directives

Allowed sources for content inclusion are specified using *source expressions*, a sort of regular expressions used to express sets of web origins in a compact way. Content inclusion from a URL is only allowed if the URL *matches* any of the source expressions specified for the appropriate content type. The relevant details of the matching algorithm will be introduced throughout the paper whenever needed.

The semantics of a content security policy can be summarized as follows:

1. the execution of inline scripts is blocked, unless the source expression `'unsafe-inline'` is included in the `script-src` directive (or in the `default-src` directive in absence of `script-src`);

2. the application of inline styles is blocked, unless the source expression `'unsafe-inline'` is included in the `style-src` directive (or in the `default-src` directive in absence of `style-src`);

3. the conversion of strings into code via `eval` and similar functions is blocked, unless the source expression `'unsafe-eval'` is in the `script-src` directive (or in the `default-src` directive in absence of `script-src`);

4. some dangerous methods of the CSS Object Model like `insertRule` are blocked, unless the source expression `'unsafe-eval'` is in the `style-src` directive (or in the `default-src` directive in absence of `style-src`);

5. the inclusion of a content of type t from a URL u is only allowed if one of these conditions holds:

   (a) u matches a source expression in t-`src`;

   (b) there is no t-`src` directive and u matches a source expression in `default-src`;

   (c) there is neither t-`src` nor `default-src`.

If more than one content security policy is deployed on the same web page, each policy must be individually enforced following the rules above.

2.2 CSP Level 2

The core of CSP is a fine-grained mechanism for white-listing content inclusion, but other features have been recently added to the original standard CSP 1.0 [2], leading to CSP Level 2 [3]. One of the major changes in the new standard with respect to CSP 1.0 is the introduction of a mechanism to relax the above restrictions on inline scripts and stylesheets, without falling back to the dramatic absence of security guarantees provided by `'unsafe-inline'`. Specifically, it is now possible to white-list individual inline scripts and styles by using *nonces* or *hashes*. The `'nonce-$value'` source expression white-lists inline scripts or styles with a `nonce` attribute equal to `$value`, while the `'shaXXX-$value'` source expression white-lists inline scripts or styles whose hash (computed using `shaXXX`) is `$value`.

CSP Level 2 also introduced a few new directives. Among them, we mention here the `frame-ancestors` directive, used to control whether browsers should be allowed to embed a CSP-protected web page inside other documents (e.g., by means of iframes). The directive is meant to supplant the existing `X-Frame-Options` HTTP response header used for frame busting [22].

2.3 Enforcing and Reporting

Content security policies can be run in two modes. The *enforcement* mode applies all the content restrictions specified by the policy, while the *report-only* mode does not restrict the website functionality, but it just tells browsers to log policy violations in the JavaScript console. In both modes, the `report-uri` directive can be used to specify a URI where browsers should send JSON-based reports when a policy violation occurs.

3. TESTING BROWSER SUPPORT

We devised a number of experiments to test the implementation of the CSP specification [3] in major web browsers. Our goal was finding both subtle corner cases of the specification which deserve clarification and plain deviations with respect to expected browser behaviours.

3.1 Methodology

We created a small set of HTML pages sending content security policies in enforcement mode, designing them so that the browser behaviour upon policy enforcement is made explicit by visual clues. We make these pages available online, along with a brief explanation of each of them[1]. We do not claim that our investigation tested all the corner cases of the specification, but we are confident about the effectiveness of our test suite in providing a good coverage of the most relevant aspects of CSP which are commonly used, as well as of the CSP semantics.

We visited the web pages with different browsers: Mozilla Firefox 46, Chromium 50, Opera 36, Safari 9.1 and Microsoft Edge 25.10586.0.0, as well as their mobile variants. Notice that Safari and Microsoft Edge do not yet implement CSP Level 2, but only CSP 1.0. Features specific to CSP Level 2 have not been tested on those browsers.

3.2 Results

We first present a set of tests passed by all major browsers and then, separately, a few cases where at least one browser behaves unexpectedly. We comment on the security import of these inaccurate implementations. All our findings have been disclosed to browser vendors via bug reports.

3.2.1 Passed Tests

All the browsers successfully passed the following tests:

1. *Composition of multiple directives*: The syntax of CSP allows the inclusion of multiple directives for the same content type (e.g., `script-src`) in the same header. The expected behaviour in this case is that only the first directive is enforced, while the other ones are ignored ([3], Section 4.1.1);

2. *Default scheme assignment*: The syntax of source expressions includes host source expressions of the form `a.com`. In these cases lacking an explicit scheme, the CSP specification mandates a default scheme assignment based on the scheme of the page deploying the policy: `a.com` must be interpreted as `https://a.com` in HTTPS pages, while it must interpreted as both `http://a.com` and `https://a.com` in HTTP pages ([3], Section 4.2.2);

3. *Wildcard*: In CSP Level 2, the `*` source expression is a wildcard matching any URL whose scheme is not `blob`, `data` or `filesystem` ([3], Section 4.2.2). These schemes are considered dangerous, since the content of URLs with these schemes is often derived from a response body and may be under the control of an attacker. Notice that in CSP 1.0 the wildcard simply matches any URL ([2], Section 3.2.2.2);

4. *Ambiguities on inline scripts*: The `script-src` directive may include both `'unsafe-inline'` and nonces or hashes white-listing individual inline scripts. In this case, the CSP specification mandates that only inline scripts white-listed using nonces or hashes are allowed to run ([3], Section 7.15). Nonces and hashes are not available in CSP 1.0.

3.2.2 Enforcing Multiple Policies

Multiple content security policies can be specified for the same web page in different headers. The CSP specification recommends that, if multiple policies are present on the same page, each of them must be enforced individually ([3], Section 3.5). Our experiments assessed that all browsers behave according to the specification, but for Microsoft Edge, which merges policies included in different headers using a custom algorithm. For each content type, the algorithm selects the first directive encountered in the policies to be merged, and drops subsequent directives of the same type found in other headers. For instance, if the first header includes the directive `script-src a.com b.com` and the second header includes the directive `script-src a.com c.com`, the protected page can load scripts from both `a.com` and `b.com`, though only `a.com` should be a valid source for script inclusion based on the CSP specification.

Though the presence of multiple headers with different directives for the same content type may sound strange at first, this situation may happen in presence of security gateways and web application firewalls run by large organizations [3]. In these cases, the behaviour of Microsoft Edge is more liberal than the specification and may lead to attacks on CSP-protected websites.

[1] http://www.dais.unive.it/~csp

3.2.3 Blocking Inline Elements

A central design choice of CSP is that inline scripts are disabled unless otherwise specified [23], for instance by using `'unsafe-inline'`. However, we observed in all the tested browsers a weird, unexpected difference in the treatment of inline scripts between the following two policies:

1. `img-src www.example.com;`
2. `img-src www.example.com; default-src *.`

Our experiments revealed that the first policy allows the execution of inline scripts, but the second one does not, despite the fact that the default sources for script inclusion should be set to the wildcard `*` in both cases [3]. This mismatch is hard to explain, confusing for web developers and not compliant with the CSP specification.

Fortunately, despite our initial concerns, the security import of this unexpected behaviour looks minor. Indeed, neither of the two policies puts any restriction on the set of URLs white-listed for script inclusion. This means that an attacker does not really need to inject an inline script to attack a website deploying any of the two policies above, which are equally vulnerable in practice: indeed, under both policies, arbitrary script injection could be performed by first hosting a malicious script on an attacker-controlled website and then injecting a script tag loading the script in the target web page.

3.2.4 Scheme Relaxation

The `'self'` source expression identifies the origin of the web page deploying a content security policy. Since web origins are defined as triples including a scheme, a hostname and a port number [8], a directive like `img-src 'self'` enforced at `http://a.com` should only allow the inclusion of images from `a.com` over HTTP. We noticed that only Microsoft Edge and Safari strictly follow the CSP specification when interpreting `'self'`. Mozilla Firefox and Chromium are instead more liberal, since the previous directive actually allows the inclusion of images from `a.com` over both HTTP and HTTPS. We observed that Mozilla Firefox and Chromium implement this scheme relaxation also in other cases, i.e., any origin with an HTTP scheme in a directive also allows the inclusion of contents served over HTTPS from the same domain.

Though it is not mentioned in the CSP specification, the scheme relaxation mechanism implemented in Mozilla Firefox and Chromium looks perfectly sensible, since it is secure and convenient for writing policies. Indeed, we noticed that this more liberal behaviour is recommended in the draft of the upcoming CSP Level 3 specification [4].

4. ANALYSIS OF CSP DEPLOYMENT

To evaluate the deployment of CSP and investigate the trends in its adoption, we performed a crawl of the homepages of the Alexa Top 1M [1] websites in March 2016. We accessed each website using both HTTP and HTTPS, and we collected the CSP headers sent in the corresponding HTTP(S) responses.

4.1 Adoption of CSP

Overall, we found CSP headers in 7,702 HTTP responses and in 6,793 HTTPS responses, delivered by 8,133 distinct websites. An earlier study [28] conducted in March 2014 identified only 850 websites deploying CSP in the Alexa Top 1M, so it seems that the popularity of CSP has significantly improved in the last two years, approximately of a ten factor. However, a more careful inspection of these numbers gives insights about the real state of the CSP deployment. Out of 8,133 websites sending CSP headers, we only found 3,220 websites running CSP in enforcement mode (39.6%). We then identified 5,016 websites (61.7%) using the report-only mode of CSP, which means that 103 websites (1.3%) implement both modes. Though the existence of such websites may be unexpected, combining enforcement and report-only mode is encouraged by the CSP specification as a good way to enforce a relatively weak policy, while monitoring the possibility of enforcing a stricter one.

The fact that only the 39.6% of the websites sending CSP headers is actually enforcing a content security policy is an interesting finding, since previous work from March 2014 [28] identified 815 websites deploying CSP in enforcement mode out of 850 websites making use of CSP (95.8%). We investigated why the majority of the websites runs CSP in report-only mode nowadays and we concluded that this is mostly due to the adoption of web development frameworks and content management systems. For instance, we found 3,432 websites deploying the very same content security policy in report-only mode, with the only difference being the `report-uri` directive used to collect violation reports. These websites are the 42.2% of all the websites deploying CSP and all of them are based on Shopify, a popular content management system for e-commerce websites, which ships a default (lax) content security policy. We also identified other frameworks and content management systems deploying CSP in report-only mode, though they are not nearly as widespread as Shopify. Based on these numbers, we think that a significant amount of web developers making use of CSP may not even be aware of the adoption of CSP on their websites!

4.2 Analysis of CSP Headers

We analysed the collected CSP headers to understand which are the most commonly used features of CSP. We mostly focused on the 3,220 websites running CSP in enforcement mode, since for these websites we assume a deliberate and fully-aware adoption of the standard.

4.2.1 Inline Elements and Unsafe Eval

Web developers are strongly encouraged to remove all the inline scripts from their websites to reap the biggest benefits out of CSP and limit the risks of XSS [29]. However, previous studies assessed that moving inline scripts to external resources is not a trivial task [26] and showed that the large majority of CSP-enabled websites just resorts to activating `'unsafe-inline'` [28]. Nonces and hashes have been introduced in CSP Level 2 to give web developers the possibility of white-listing individual inline scripts and stylesheets. These mechanisms were designed to simplify a large-scale adoption of CSP and it is important to understand whether or not they have been successful so far in replacing the insecure `'unsafe-inline'`.

Out of 3,220 websites running CSP in enforcement mode, 1,669 include `'unsafe-inline'` in a `script-src` directive (51.8%). Only 48 websites make use of hashes to white-list their inline scripts (1.5%), while 37 websites rely on nonces (1.1%). This shows that the majority of the web developers still disregards the dangers posed by inline scripts and

trades security for programming convenience, despite the existence of new tools designed for minimising code changes to existing websites. Similar findings apply to stylesheets: 1,564 websites use `'unsafe-inline'` in a `style-src` directive (48.6%), while only 2 websites use hashes to white-list stylesheets and none relies on nonces.

We also found 1,680 websites (52.2%) including `'unsafe-eval'` in a `script-src` directive and 126 websites (3.9%) including it in a `style-src` directive. This suggests that the majority of the websites using CSP resorts to dynamically transforming strings into code, despite the fact that this is well-recognised as an insecure programming practice.

4.2.2 Violation Reports

We assessed the adoption of the `report-uri` directive to collect CSP violation reports. This is important, since violations to content security policies without this directive are only logged in the JavaScript console and are much more difficult to systematically detect for web developers, because all the violations triggered by website users are lost. We observed that only 694 out of 3,220 websites running CSP in enforcement mode (21.6%) specify a `report-uri` directive, hence most websites do not implement a robust monitoring of their content security policies. This is surprising, since it is very easy to activate the reporting facilities of CSP and to parse the violation reports.

We also found 94 websites running CSP in report-only mode, but without a `report-uri` directive. This is a very small fraction (1.9%) of the 5,016 websites making use of the report-only mode of CSP, but these cases are particularly strange, since the lack of `report-uri` significantly reduces the benefits of reporting. We investigated these cases and we noticed 42 websites affiliated to Envato, an online market of digital assets. These websites just implement redirects to personal pages hosted at Envato and they all share two common content security policies, configured for the Envato website.

4.2.3 Frame Busting

Among the 3,220 websites running CSP in enforcement mode, we found 697 websites (21.6%) using CSP just to implement frame busting. These websites deploy very simple content security policies like `frame-ancestors 'self'`. Remarkably, the goal of these policies is orthogonal to the original scope of the CSP specification, which aimed at introducing "restrictions that give web application authors control over the content embedded on their site" [23].

The `frame-ancestors` directive is an exceptionally popular mechanism introduced in CSP Level 2, with 1,474 websites making use of it in our dataset (45.8%).

5. ERRORS IN CSP POLICIES

To evaluate whether web developers can correctly specify content security policies, we performed an in-depth inspection of the CSP headers collected from the Alexa Top 1M, looking for different types of errors.

5.1 Methodology

Systematically detecting errors in CSP policies is subtle, as one needs to define a meaningful notion of error, independent of the specific use case and which does not require to speculate on whether web developers actually enforced what they wanted to enforce. We focus on three classes of *factual* errors made by web developers:

1. *typos and negligences*: syntactic errors in policy specification. In these cases it is completely clear what web developers wanted to enforce, but they specified it incorrectly, e.g., the name of a directive was misspelled;

2. *ill-formed policies*: a first form of semantic error in policy specification. These policies have an unclear intended meaning, e.g., they contain contradictory or unexpected information;

3. *harsh policies*: a second form of semantic error in policy specification. These policies are too strict and trigger CSP violations upon navigation of the website.

We defined these classes of errors after a preliminary manual investigation of our dataset and we devised appropriate queries to automatically catch these errors in all the websites we visited.

5.2 Typos and Negligences

Syntactic errors in CSP headers are very easy to catch and fix, but their import on security is significant, since all the major web browsers ignore unknown directives in content security policies and just output a warning in the JavaScript console. If web developers are not careful enough, they may deploy unexpectedly weak content security policies.

In our analysis, we found 5 websites sending CSP headers containing unknown directives, due to obvious typos like:

```
defalt-src 'self'
nfont-src www.myfonts.com
report-uri/csp-report
```

We clarify the security import of this kind of trivial mistakes: the typo in the first directive leads to the `default-src` directive being missing from the policy, actually white-listing every website as a valid provider of contents lacking an explicit directive. Similar considerations apply to errors like the second one, which allows browsers to load fonts from any website (in absence of a stricter default directive). Errors like the third one prevent the correct generation of CSP violation reports, which may lead to attacks and policy issues going undetected for a long time. We also found 3 websites where the name of a directive was written incorrectly, not necessarily due to a typo, but most likely because the web developer was inaccurate in checking the CSP specification (e.g., writing `frame-ancestor` rather than `frame-ancestors`).

We also noticed 8 websites misusing punctuation symbols next to directives. A few representative examples are:

```
"default-src 'self'; ..."
default-src: 'self'
default-src=self
```

All these cases lead to (a portion of) the content security policy being skipped by the browser and not correctly enforced, with the risks described above.

Misquoting special source expressions like `'self'` or missing the terminating colon when writing a scheme like `http:` is another prominent kind of errors, resulting in the white-listing of a non-existing host. This may lead to the deployment of content security policies which are more restrictive than intended. The impact of these errors on security is thus

limited, though they may lead to severe usability issues for website users: for instance, white-listing `self` rather than `'self'` prevents the browser from loading same origin contents on a CSP-protected web page. Notice that this may even convince uncaring web developers to abandon CSP to prevent further usability issues. We found 19 websites with such errors in source expressions.

5.3 Ill-Formed Policies

We noticed a number of content security policies with an unclear meaning or using the CSP directives in an unexpected way. These cases are typically hard or even impossible to fix without contacting the original authors of the policies, since it is unclear what they wanted to enforce. To illustrate, we identified 6 websites whose content security policies have the following format:

```
script-src a.com b.com; c.com
```

There are at least two legitimate intended interpretations for an incorrect policy like this one. The first possibility is that `c.com` should be actually part of the `script-src` directive: it is plausible that the web developer included this entry after the semicolon by accident. The second possibility, instead, is that the developer forgot to insert the name of the directive preceding `c.com`. Interestingly, the first error would make the policy more restrictive than intended, while the second error could also make it more liberal, e.g., in absence of a `default-src` directive.

We also found 50 websites whose CSP header just contained the character `*`. This was surprising, since such a policy does not contain any directive and it is ignored by web browsers. The quirk was readily explained when we realised that all the 50 websites were developed using ASP.NET, so this is likely a default behaviour implemented by the web framework when CSP support is not properly configured.

Finally, we found 22 websites repeating the same directive (e.g., `script-src`) multiple times. In these cases, all the browsers we tested enforce the first occurrence of the directive and ignore the other ones, correctly following the CSP specification. It is unclear whether web developers are really aware of how web browsers enforce such policies and why repeated directives are not just removed, so it is legitimate to deem these cases at least as bad practices.

5.4 Harsh Policies

We developed a Chromium extension which intercepts the CSP headers of incoming HTTP(S) responses and changes the `report-uri` directive so that any CSP violation report is redirected to a web server under our control. We then used Selenium to drive Chromium into navigating all the websites deploying CSP in enforcement mode, while running our extension. This way, we were able to automatically detect CSP violations triggered when accessing the homepage of these websites. Surprisingly, we observed that 553 out of 3,220 websites (17.2%) trigger at least one CSP violation when their homepage is accessed by our crawler. Notice that this is a subset of the real violations which may be triggered upon navigation, since the crawler does not exercise any website functionality besides page loading. It is interesting to note that 414 of these websites (74.9%) do not use the `report-uri` directive to collect CSP violation reports, so it is perfectly plausible that these violations went unnoticed by web developers. Overall, we collected 921 violation reports: we summarise the reasons for the violations in Table 2.

Table 2: Summary of CSP Violations

Type of Violation	*#Violations*	*#Websites*
Inline scripts	12	9
Inline styles	82	80
Use of `eval`	6	6
`data:` or `blob:`	43	33
Remote inclusion	778	441

We observed 12 inline scripts blocked by CSP in 9 websites. Most of these scripts are related to advertisement or other third-party functionalities injected in the web pages, like site metrics. Interestingly, we also found 82 inline styles blocked by CSP in 80 websites. After a manual investigation of these cases, we noticed they are due to a high number of tiny styles applied to individual page elements, e.g., to draw borders around text boxes, which probably went unnoticed.

We then found 6 websites where a call to `eval` was stopped by CSP. One site used `eval` to invoke `decodeURIComponent` on the base64 encoding of an email address, which is thus not rendered correctly; one site invoked `eval` to populate some global variables needed to apply style elements to the homepage; one other site made use of `eval` to check whether the web browser accessing the site was implementing CSP correctly. The last 3 cases were more involved and harder to understand by code inspection, though we noticed that 2 of them seem to be related to AngularJS[2].

We also detected 43 violations in 33 websites due to the `data:` or `blob:` source expressions being missing in a directive. Most of these cases are related to images (16 violations) and fonts (23 violations), with probably just minor visual consequences.

We finally performed a more systematic evaluation of the 778 violations fired upon requests for HTTP(S) resources which had not been white-listed in the content security policy. We observed in particular two recurrent patterns, which cover almost half of the violations we encountered. First, we noticed 232 violations (29.8%) in 198 websites which are due to advertisement or tools loaded from websites owned by Google, i.e., whose hostname contains the strings `google`, `gstatic` or `doubleclick`. Part of these violations are due to the fact that `google.com`, often correctly white-listed in the content security policy, actually enforces a redirection to `google.tld`, where `tld` is a national top-level domain. There is no easy way to accommodate this use case in the current CSP specification, since the syntax of policies does not allow source expressions of the form `google.*` [3]. Second, we observed 78 violations (10.0%) in 54 websites due to requests targeted at the same domain of the site or some sub-domain of it. Besides the obvious cases where web developers forgot to include the site domain (or some sub-domain of it) in the content security policy, we noticed two other main reasons for this kind of violations:

1. HTTPS websites requesting contents over HTTP, despite a strong content security policy which prevents this behaviour. These cases often occur when source expressions like `'self'` or `a.com` are included in the policy, since they only white-list HTTPS contents when deployed on HTTPS pages. Some of these violations have no visible import, since modern browsers already

[2]http://docs.angularjs.org/api/ng/directive/ngCsp

block requests for active contents sent over HTTP from HTTPS pages in accordance with the mixed content policy [5];

2. websites like `http://www.a.com` loading contents from `http://a.com`, though only `http://www.a.com` is declared as a valid source for content inclusion (or vice-versa). These cases often occur when the policy uses the `'self'` source expression, since `'self'` only whitelists same origin contents, but `http://www.a.com` and `http://a.com` are different origins. The occurrence of these violations thus depends on the user typing the `www` prefix or not in the address bar when accessing the website, which is undesirable.

6. SECURITY OF CSP AGAINST XSS

Though CSP is becoming a complex and variegate standard, the main threat addressed by CSP is still content injection [23, 29]. Content injection may take different forms and be exploited to mount a number of attacks, like UI redressing. In our view, however, the most dangerous form of client-side content injection on the Web is XSS, where arbitrary attacker-controlled scripts are injected in benign web pages. Our goal here is understanding to which extent websites running CSP use it to protect against XSS.

6.1 Methodology

Systematically detecting whether a website is vulnerable to XSS is still an open research challenge, because it would require an automatic analysis of its sanitization routines [27]. Since we are interested in CSP in the present paper, we tackle a slightly different research question: do existing content security policies provide enough protection to ensure that a content injection cannot be exploited to run arbitrary code? We thus identify sufficient conditions on content security policies under which any content injection can be turned into XSS, despite a correct policy enforcement. The threat model we consider is the standard *web attacker* from the literature, normally used when reasoning about content injection. The web attacker operates a malicious website, `attacker.com`, and can respond to HTTP(S) requests sent to this website with arbitrary content. We assume the attacker set up HTTPS on his web server.

Having defined the threat model above, we can introduce the following notion of *liberal* source expression.

Definition 1 (Liberality). *A* liberal *source expression is the wildcard* `*` *or any of the schemes* `http:`, `https:` *or* `data:`.

Liberal source expressions constitute a poor mechanism to specify content restrictions, since the set of sources they denote includes attacker-controlled contents. The wildcard `*` and the HTTP(S) scheme include `attacker.com` as a valid source for content inclusion, while the `data:` scheme is dangerous since data URIs provide a way to include inline contents in web pages just as if they were external resources.

Based on this auxiliary notion, we can introduce the main definition of interest.

Definition 2 (Vulnerability to XSS). *We say that a content security policy is* vulnerable to XSS *if at least one of the following conditions holds true:*

1. `script-src` *includes* `'unsafe-inline'`*, but it does not include hashes or nonces;*
2. `script-src` *includes a liberal source expression;*
3. *there is no* `script-src` *directive and* `default-src` *includes* `'unsafe-inline'`*, but it does not include hashes or nonces;*
4. *there is no* `script-src` *directive and* `default-src` *includes a liberal source expression;*
5. *both* `script-src` *and* `default-src` *are missing.*

If any of the previous conditions holds true and content injection is possible, then arbitrary code injection becomes possible. Condition (1) is trivial: if a website allows arbitrary inline scripts, the attacker can directly inject malicious scripts in the website. Under condition (2), inline scripts are not allowed to run, but the attacker can inject arbitrary code by fetching a remote script from `attacker.com` or by using a suitable data URI. Condition (3) is similar to condition (1), while condition (4) is reminiscent of condition (2), since `default-src` applies when `script-src` is missing. Condition (5) covers websites which do not restrict in any way their set of allowed sources for script inclusion.

There are a few points which are worth discussing in the definition. First, the definition is based on CSP Level 2: it can readily be adapted to CSP 1.0 by dropping the side-conditions about nonces and hashes, which would deem more content security policies as vulnerable. Second, there are no clauses predicating on `'unsafe-eval'`, since the contents which are passed to `eval` or any similar function may undergo a sanitization process, thus preventing the attacker from injecting arbitrary code on a web page. Finally, it is worth stressing that the clauses only define *sufficient* conditions for turning a content injection into an arbitrary code injection: for instance, websites making use of nonces may still be subject to XSS attacks in some cases[3].

6.2 Results

Based on the previous definition, we observed that 2,974 out of 3,220 websites running CSP in enforcement mode are vulnerable to XSS (92.4%). We report in Table 3 the number of websites which violate each of the conditions of Definition 2. The sum is higher than 2,974, since the same website may violate more than one condition.

Table 3: Reasons for Vulnerability to XSS

Reason for Vulnerability	*#Websites*
`'unsafe-inline'` in `script-src`	1,619
liberal src. exp. in `script-src`	259
no `script-src` +	
`'unsafe-inline'` in `default-src`	275
liberal src. exp. in `default-src`	175
no `default-src`	1,024

The majority of the websites in this set renounces to security by including `'unsafe-inline'` in `script-src` or `default-src` without making use of hashes or nonces: this is the case for 1,894 websites, amounting to the 63.7% of

[3]http://blog.innerht.ml/csp-2015/

the vulnerable websites. We tried to assess whether inline scripts are really needed for these websites by developing a Chromium extension which strips `'unsafe-inline'` from incoming CSP headers and redirects violation reports to a web server under our control. We then used Selenium to navigate the 1,894 websites, accessing them with Chromium after installing the extension. Overall, we found violations due to the presence of inline scripts in 1,591 out of 1,894 websites (84.0%). This confirms that inline scripts are still pervasive nowadays, but it also suggests that 303 websites could at least attempt the deployment of a stricter and safer content security policy.

Finally, we had a more in-depth look at the 246 websites whose content security policies violate all the clauses of Definition 2 and are thus deemed as potentially robust against XSS. We observed that 56 of these websites (22.8%) deliver the very same policy, blocking inline scripts and only allowing the inclusion of contents from the same origin of the website. While we were not able to identify a common framework or owner for these websites, which appear unrelated and developed using different technologies, we think that the content security policy enforced by them may have been copied verbatim from some online source. Other secure policies are extremely simple, using only the `default-src` or the `script-src` directives to white-list local contents.

7. EVOLUTION OF CSP DEPLOYMENT

We conducted a series of experiments to estimate how the adoption of CSP and existing content security policies are evolving over time. Our goals were detecting relevant trends and assessing whether web developers keep their content security policies constantly updated. To evaluate this, we performed weekly crawls of the homepages of the Alexa Top 1M websites [1] from March to June 2016, collecting their CSP headers. We also built a dataset of CSP violations using our Chromium extension and we looked for correlations between changes to existing content security policies and website functionality being reduced by policy enforcement.

7.1 Changes in CSP Adoption

Let $t_1, \ldots, t_n$ be the sequence of the weekly crawls. We say that a website w *commits* to CSP if there exists a crawl t_i such that w does not enforce CSP in $t_1, \ldots, t_{i-1}$, but w enforces CSP in $t_i, \ldots, t_n$. Conversely, a website w *abdicates* from CSP if there exists a crawl t_i such that w enforces CSP in $t_1, \ldots, t_{i-1}$, but w does not enforce CSP in $t_i, \ldots, t_n$.

Figure 1 reports the number of committing and abdicating websites we found during our weekly crawls. We observed many more websites committing to CSP rather than abdicating from it in all our crawls, which testifies a constant growth in the CSP deployment. Overall, we found 931 committing websites and 268 abdicating websites in the considered timespan. Interestingly, we noticed that 79 abdicating websites (29.5%) triggered at least one CSP violation during our crawls. We believe that this non-negligible amount of policy violations may quite possibly have influenced the choice of abdicating from CSP.

Another relevant trend we analysed in the CSP adoption is the transition from report-only to enforcement mode, which should be the most desirable outcome of a preliminary reporting phase. Overall, we found 26 websites changing their policies from report-only to enforcement mode during our crawls, thus fully embracing CSP. However, we also found 6

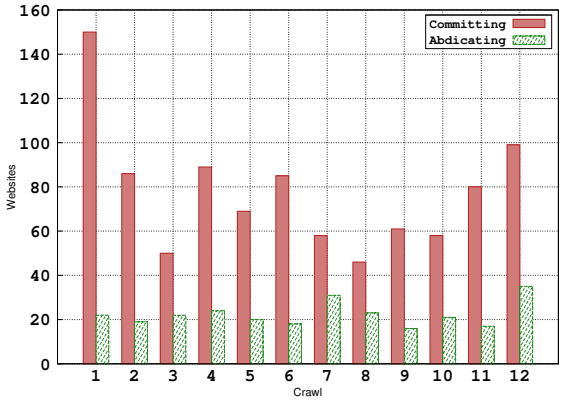

Figure 1: Committing and abdicating websites

websites switching from enforcement mode to report-only.

Finally, we observed an interesting phenomenon of *temporary* CSP support. Let $t_1, \ldots, t_n$ be the sequence of the weekly crawls, we say that a website w had temporary CSP support if w does not enforce CSP in t_1 and t_n, but there exists t_i such that w enforces CSP in t_i. We found 2,862 websites with temporary CSP support in our weekly crawls. An investigation of these websites revealed that this phenomenon was mostly due to Blogger, a famous blogging service hosting several websites in the Alexa ranking: 2,576 websites with temporary CSP support contain the strings `blogger` or `blogspot` in their hostname (90.0%). Notably, their policies only included `upgrade-insecure-requests`, a newly proposed CSP directive [7] not present in the official specification. This directive asks web browsers to upgrade to HTTPS a number of HTTP requests sent by CSP-protected websites, so as to simplify their full transition to HTTPS while avoiding mixed content errors. The policies were likely removed after a full transition of Blogger to HTTPS. It is interesting to notice that Blogger never used CSP for its original purposes of restricting content inclusion, but just to benefit of this recent addition to the standard.

7.2 Changes in Content Security Policies

We evaluated how frequently websites change their content security policies. To get uniform and reliable results, we focused on the 2,784 websites enforcing CSP in all the weekly crawls. The majority of these websites never changed their content security policies: this is the case for 1,855 websites (66.6%). Remarkably, however, there are also 929 websites which updated their policies at least once.

Figure 2 reports the distribution of these 929 websites with respect to the number of observed policy changes. As expected, the majority of the websites changed their policies only once or twice. However, it is not uncommon to observe websites which changed their policies more frequently. The most surprising cases in this respect are websites which changed policy basically every week: a manual investigation revealed several pornographic websites including apparently random strings as valid hostnames for content inclusion. In these cases, it seems that the same contents are regularly relocated on different domains, likely due to legal reasons.

An important aspect we investigated on our data is how the treatment of dangerous constructs like inline elements

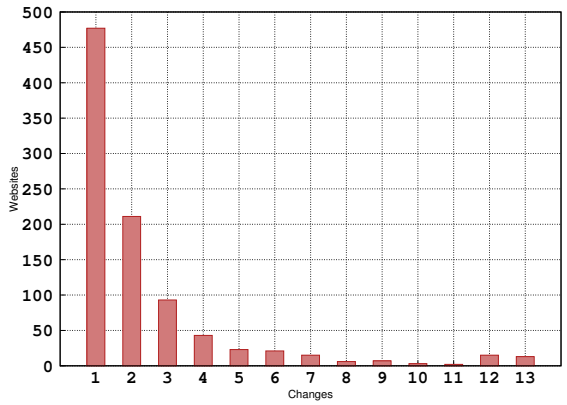

Figure 2: Number of changes to CSP policies

and `eval` changed over time. It seems websites are reluctant to update their policies to change the browser behaviour for these constructs and, even worse, the trend is not positive from a security perspective. Overall, we only found 6 websites improving their security by dropping `'unsafe-inline'`, replacing `'unsafe-inline'` with hashes or nonces, or by dropping `'unsafe-eval'`; however, we identified 16 websites which made the opposite choices. We also found 2 websites which tested the possibility of removing `'unsafe-inline'` for a few weeks, but eventually resorted to adding it back in their policies.

Other insightful observations on policy changes come from our dataset of CSP violations. We detected 682 violations during our crawls which eventually disappeared at some point in time. For these cases, we compared the content security policies enforced before and after the disappearance of the violation, to investigate whether the changes performed by the web developers were aimed at fixing the violation or not. We identified 95 interventions making policies more liberal to enable a blocked functionality, while in 587 cases the changes were not related to the violations we collected. It is interesting to note that 587 violations disappeared though the underlying content security policy was not actually changed to prevent them: this behaviour is likely due to the dynamic nature of modern websites and to the widespread practice of including advertisement. We think that the volume of these transient violations is worrisome, since it means that it is difficult to keep content security policies constantly updated.

Further insight about the need for policy changes comes from the analysis of *persistent* CSP violations. We found 322 violations in 241 websites being triggered at every visit of our crawler. These cases call for policy changes, but apparently web developers are not aware of them or have been unable to fix them properly. We think that the first possibility is the most likely, since 190 out of the 241 websites (78.8%) do not make use of the `report-uri` directive to systematically collect CSP violation reports.

8. PERSPECTIVE

As a result of our investigation, we found two main classes of problems for CSP in its current form. The first class of problems comes from a lack (or loss) of useful feedback for web developers when writing content security policies. Though the reporting facilities of CSP are excellent, the large majority of the web developers do not benefit of them, since the 78.4% of the websites running CSP in enforcement mode do not activate the `report-uri` directive. A simple change we propose is making browsers output a warning in the JavaScript console when parsing a policy lacking the `report-uri` directive: none of the browsers we tested provides this warning. We think that recommending the usage of `report-uri` would be very helpful to make web developers aware of the importance of reporting and to prevent the deployment of policies which are too strict (Section 5.4). Moreover, we propose that the `report-uri` directive should also be leveraged to collect CSP violation reports whenever unknown directives or ill-formed policies are parsed by the browser. This would be useful to prevent the numerous errors discussed in Sections 5.2 and 5.3. These errors are a serious problem in practice, since the syntax of CSP is very liberal and browsers are tolerant when parsing policies for the sake of backward compatibility.

Unfortunately, the second class of problems affecting CSP is more rooted into its design. Banning inline scripts is certainly important to mitigate code injection, but too many web developers still resort to activating `'unsafe-inline'` on their web pages (58.8%). Nonces and hashes are a step in the right direction, but their adoption is minuscule: less than the 3% of the websites running CSP in enforcement mode use them and the trend does not seem to be changing. Moreover, inline scripts are not the only attack vector for code injection: 434 websites (13.5%) include a liberal source expression in their white-list for script inclusion. This leads us to the more general observation that white-lists require web developers to strike a very delicate balance between security and usability. Carefully designed white-lists are difficult to write and to maintain, as testified by the large number of CSP violations we encountered on existing websites: as a result, web developers resort to white-listing liberal source expressions to prevent functionality issues. To make matters worse, CSP violations are often due to elements which are not totally under the control of the author of the content security policy. In our analysis, we noticed that redirects and advertisement systems are particularly troublesome in this respect. Redirects trigger security violations when a white-listed origin forces a redirection to an origin which is not white-listed. Advertisement systems, instead, have a very dynamic and unpredictable behaviour which hardly fits the nature of a white-list, hence they end up triggering transient security violations. More research is needed to find robust solutions to these issues.

9. RELATED WORK

Before this study, two research papers about the state of the CSP deployment have been published [28, 21]. Patil and Frederik performed an analysis of the CSP adoption on the Alexa Top 100k in October 2013 [21]. After assessing a limited adoption of the standard, the authors proposed a tool, UserCSP, to automatically synthesise content security policies for existing websites. Weissbacher *et al.* evaluated the deployment of CSP on the Alexa Top 1M in March 2014 [28]. They then discussed challenges for practical CSP adoption and techniques for semi-automated policy generation.

There are many important differences between the present paper and previous work [28, 21]. First, the focus of the works is different, since we are only interested in assessing

the trends and the effectiveness of the current CSP adoption, while [28, 21] put great emphasis on (semi-)automated policy generation. Finding effective ways to generate content security policies is definitely an important and intriguing research challenge, which we plan to pursue in future work. On the other hand, the evaluation of the CSP effectiveness in [28, 21] is not nearly as comprehensive and systematic as ours: [28, 21] do not include any evaluation of the CSP support in existing browsers, nor any analysis of common errors in policy specification. Also the security analysis in [28, 21] is much more limited than ours, less rigorous and not as exhaustive in covering the possible attack vectors for XSS. Only [28] briefly touches on the point of the evolution of CSP, but it is limited to three websites (BBC, CNN, Twitter), while we focus on 2,784 websites.

Besides these methodological aspects, there are also good technological reasons motivating further, up-to-date research about CSP. When the studies in [28, 21] were performed, CSP Level 2 did not yet exist, so there is no published research on the latest additions to the CSP standard, e.g., hashes and nonces. Moreover, the adoption of CSP has significantly increased in the last two years: [21] identified only 27 websites running CSP in enforcement mode, [28] found 815 websites, while the present work identified 3,220 websites. Such a larger scale calls for a more systematic evaluation like the one pursued in this paper.

Van Acker *et al.* studied the current inability of CSP at preventing data exfiltration attacks [24]. Their work provides empirical evidence that no major web browser implements defences against data exfiltration in presence of DNS and resource prefetching, even when the strongest content security policy is put in place. Their paper also proposes mitigation techniques against these issues.

Hausknecht *et al.* focused on the tension between content security policies and browser extensions [12]. Since browser extensions can modify the DOM, they may end up making web pages request external resources which are not whitelisted by the underlying content security policy. The paper proposes a mechanism to endorse CSP modifications performed by browser extensions, so as to strike a good balance between websites security and extensions functionality.

Hothersall-Thomas *et al.* developed BrowserAudit, a web application implementing a series of more than 400 automated security tests for web browsers [13]. Notably, BrowserAudit also includes a set of 226 tests for CSP 1.0 divided in 10 main families. The compliance tests for CSP implemented in BrowserAudit are simple and quite low-level, likely because the scope of BrowserAudit is not limited to CSP, but rather embraces browser security as a whole.

Johns identified three limitations of CSP leaving room for dangerous code injections: no prevention of insecure server-side assembly of JavaScript code, lack of control over the content of white-listed external scripts, and lack of control over the number and the appearance order of script tags [15]. His paper proposes a framework, called PreparedJS, which complements CSP with solutions (or mitigations) against these attack vectors.

The present paper positions itself in the popular research line of large-scale security evaluations of the Web [25]. Just to mention a few relevant works, previous evaluations focused on other aspects of web security, like remote JavaScript inclusion [20], DOM-based XSS [17], mixed content websites [10], authentication cookies [9] and HSTS [16].

10. CONCLUSION

We performed a large-scale, systematic analysis of four key factors to the effectiveness of CSP: browser support, website adoption, correct configuration and constant maintenance. Though browser support is largely satisfactory, with the exception of few notable issues, the other three points present significant shortcomings. The deployment of CSP is still quite limited in practice and, more importantly, there are many errors and weaknesses in existing content security policies, which leave room for security or usability issues. Moreover, content security policies are not regularly updated to ban insecure practices and remove unintended security violations. We argue that many of the problems we found can be fixed by better exploiting the reporting facilities of CSP, but other issues deserve additional research, being more rooted into the CSP design.

Overall, CSP is growing, but not nearly as fast and effectively as desirable. Given the still relatively limited adoption of the standard, this could be an excellent moment for a retrospective look at its design and motivations based on the observations we collected.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. The present paper was supported by the MIUR project ADAPT.

11. REFERENCES

[1] Alexa top sites. http://www.alexa.com/topsites.
[2] Content Security Policy 1.0. https://www.w3.org/TR/2012/CR-CSP-20121115/.
[3] Content Security Policy Level 2. https://www.w3.org/TR/CSP2/.
[4] Content Security Policy Level 3. https://w3c.github.io/webappsec-csp/.
[5] Mixed content. https://www.w3.org/TR/mixed-content/.
[6] OWASP Top 10 Threats. https://www.owasp.org/index.php/Top_10_2013-Top_10.
[7] Upgrade insecure requests. https://www.w3.org/TR/upgrade-insecure-requests/.
[8] The web origin concept. https://tools.ietf.org/html/rfc6454.
[9] Stefano Calzavara, Gabriele Tolomei, Andrea Casini, Michele Bugliesi, and Salvatore Orlando. A supervised learning approach to protect client authentication on the web. *TWEB*, 9(3):15, 2015.
[10] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. A dangerous mix: Large-scale analysis of mixed-content websites. In *ISC*, pages 354–363, 2013.
[11] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security*, 31(4):612–628, 2012.
[12] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. May I? - Content Security Policy endorsement for browser extensions. In *DIMVA*, pages 261–281, 2015.
[13] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. BrowserAudit: automated testing of browser security features. In *ISSTA*, pages 37–47, 2015.

[14] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, pages 601–610, 2007.

[15] Martin Johns. Script-templates for the Content Security Policy. *J. Inf. Sec. Appl.*, 19(3):209–223, 2014.

[16] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS*, 2015.

[17] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, pages 1193–1204, 2013.

[18] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *S&P*, pages 331–346, 2009.

[19] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[20] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *CCS*, pages 736–747, 2012.

[21] Kailas Patil and Braun Frederik. A measurement study of the Content Security Policy on real-world applications. *I. J. Network Security*, 18(2):383–392, 2016.

[22] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *W2SP*, 2010.

[23] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with Content Security Policy. In *WWW*, pages 921–930, 2010.

[24] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data exfiltration in the face of CSP. In *ASIA CCS*, 2016.

[25] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale security analysis of the web: Challenges and findings. In *TRUST*, pages 110–126, 2014.

[26] Joel Weinberger, Adam Barth, and Dawn Song. Towards client-side HTML security policies. In *HotSec*, 2011.

[27] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Eui Chul Richard Shin, and Dawn Song. A systematic analysis of XSS sanitization in web application frameworks. In *ESORICS*, pages 150–171, 2011.

[28] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. Why is CSP failing? Trends and challenges in CSP adoption. In *RAID*, pages 212–233, 2014.

[29] Mike West. An introduction to Content Security Policy. http://www.html5rocks.com/en/tutorials/security/content-security-policy/.