Injecting CSP for Fun and Security

Christoph Kerschbaumer[1], Sid Stamm[2] and Stefan Brunthaler[3]
[1] *Mozilla Corporation, Mountain View, USA*
[2] *Rose-Hulman Institute of Technology, Terre Haute, USA*
[3] *SBA Research, Vienna, Austria*
ckerschb@mozilla.com, stammsl@rose-hulman.edu, sbrunthaler@sba-research.org

Keywords:
Web Browser Security, Content-Security-Policy (CSP), Cross Site Scripting (XSS)

Abstract:
Content Security Policy (CSP) defends against Cross Site Scripting (XSS) by restricting execution of JavaScript to a set of trusted sources listed in the CSP header. A high percentage (90%) of sites among the Alexa top 1,000 that deploy CSP use the keyword `unsafe-inline`, which permits all inline scripts to run—including *attacker–injected* scripts—making CSP ineffective against XSS attacks. We present a system that constructs a CSP policy for web sites by whitelisting only expected content scripts on a site. When deployed, this auto-generated CSP policy can effectively protect a site's visitors from XSS attacks by blocking injected (non-whitelisted) scripts from being executed. While by no means perfect, our system can provide significantly improved resistance to XSS for sites not yet using CSP.

1 Motivation

Modern web pages are complex web applications that pull scripts from different origins into the same execution context (Nikiforakis et al., 2012). While powerful, this execution scheme also opens the door for cross-site scripting (XSS) attacks and vulnerability studies consistently rank XSS highest in the list of attacks on web applications (OWASP, 2012; The MITRE Corporation, 2012; Microsoft, 2012).

As of today, all major web browsers (Chrome, Firefox, Internet Explorer, Safari) support the `Content-Security-Policy` HTTP header. However, a recent study (Weissbacher et al., 2014) shows that fewer than one percent of Alexa top 100 sites use CSP. Challenges in adoption (Weissbacher et al., 2014), such as lack of framework support, are often barriers to CSP deployment on many Web sites. Additionally, our own crawl of the Alexa top 1,000 sites shows that only 20 deploy CSP; and of those, only two avoid use of the keyword `unsafe-inline`. The keyword `unsafe-inline` was originally introduced to support legacy code while transitioning sites to use CSP. This keyword whitelists all inline scripts for a site, but it also allows attacker–injected scripts code to execute, making CSP ineffective against most XSS attacks.

We present a system that reduces the burden on site authors: it does not require manual effort from site authors to craft, deploy and maintain their own CSP. Our system shifts the work from the site authors to the user agents, by having browsers report hashes of served inline scripts of a web page to a trusted third-party policy server. This policy server then accumulates those script hashes from many individual visitors into a *script profile* for subject web sites. The accumulated *script profiles* are then used to automatically generate a CSP header for each site tracked by the policy server. Subsequent visitors to tracked sites can query this policy server for its generated CSP, and then apply it when assembling and rendering the site content. While all users of our system will continue reporting scripts appearing on tracked sites, they can also enjoy protection from a CSP based on what *other visitors* expect to appear. This crowdsourced approach allows inspection deep into web sites, permitting our system to gather information about inline scripts beyond the initial landing pages.

In this paper, we first establish the threat model we aim to defend against (Section 2) and then contribute the following:

- We measure current CSP deployment (Section 3) by visiting the Alexa top 1,000 sites in the world, show that only two percent of pages deploy CSP, and note how few of those effectively protect against XSS attacks.
- We present design and implementation details of a system (Section 4) based on Firefox (v39.0) that allows any web page to be deployed with a CSP header.
- We evaluate our approach and demonstrate the ability of our system (Section 5) to protect up to half of web pages against XSS attacks. Discussion includes the limitations of our approach (Section 6).

2 Threat Model

Since attacker–supplied JS executes in the target page's context, it can perform many actions as if it were the user; this could lead to reading a user's private data, stealing their authentication credentials, or harvesting sensitive user information like keystrokes. A greedy script may also traverse the Document Object Model (DOM) (W3C - World Wide Web Consortium, 2004) and steal any visible data on a compromised web page (Russo et al., 2009). Attackers often employ these methods of XSS to gain access to confidential user information, or perform actions without the users consent or knowledge.

In this paper we focus on one goal of web site authors: prevent site visitors from leaking sensitive information such as cookies or login credentials to an attacker. Throughout this paper we assume that the attacker's capabilities are limited to JavaScript injection and that attackers can:

1. inject a sequence of bytes into other web pages
2. own and operate their own web sites
3. neither intercept nor control network traffic
4. not exploit any privilege escalation vulnerabilities in the browser

3 Content Security Policy

Within a CSP policy, the directive `script-src` defines from which URLs and contexts a document may load scripts (Stamm et al., 2010; W3C - World Wide Web Consortium, 2014). More critically, when the `script-src` directive appears in combination with the keyword `unsafe-inline`, CSP cannot prevent XSS attacks.

```
script-src * ’unsafe-inline’;
```

A policy like the one defined above provides no additional security for a web site, since it permits all script code whether loaded from an external resource (indicated by the wildcard `*`), inlined by the site, or injected by an attacker (both permitted by `unsafe-inline`). In other words, if an attacker manages to inject script code into a page using this policy, then CSP will not stop the attack code from being executed within that page.

When used correctly however, CSP can stop script injection attacks even if an attacker is able to exploit an XSS vulnerability in the web page. The `script-src` directive allows developers to enumerate specific inline scripts by specifying the scripts' hashes as an allowed source:

```
script-src https://example.com ’sha512-XVHzN
    ...=’;
```

A browser enforcing the above policy will permit external scripts loaded from `https://example.com` and will additionally allow inline scripts to execute if the inline script's hash matches the one `sha512`-hash specified in the policy: `XVHzN...=`.

It is important to note that JS event handlers are also considered inline scripts and hence also subject to the `script-src` directive (W3C - World Wide Web Consortium, 2014). For example, if a page makes use of `<a href='javascript:foo();'>link</a>` then the hash of the scripttext, in that case the hash of `foo();` would have to match any of the hashes defined within the `script-src` directive so that CSP would allow the JS event handler to execute.

In contrast to the first policy displayed above, this second policy *does not* allow all inline scripts to execute. By being more restrictive, the second policy can prevent attempted script injections.

3.1 Browser Enforcement of CSP

A browser handling a CSP policy makes its decision whether to allow or block any inline script by first consulting the `script-src` directive (see Figure 1). If the `script-src` directive is not defined within a site's CSP, then a browser's CSP

	AR	*Page*	*IS*	`default-src`	`script-src`	*safe*
1	2	`facebook.com`	12		`unsafe-inline`	
2	12	`twitter.com`	4		`unsafe-inline`	
3	25	`yandex.ru`	15	`unsafe-inline`	∅	
4	30	`pinterest.com`	9	`unsafe-inline`	∅	
5	36	`mail.ru`	78		`unsafe-inline`	
6	65	`cnn.com`	14		`unsafe-inline`	
7	107	`vimeo.com` (R)	8	`unsafe-inline`	∅	
8	126	`dropbox.com`	17		`unsafe-inline`	
9	198	`github.com`	0		✓	✓
10	378	`yandex.ua`	17	`unsafe-inline`	∅	
12	436	`w3.org`	1		`unsafe-inline`	
12	438	`fbcdn.net`	12		`unsafe-inline`	
13	440	`zendesk.com`	18	`unsafe-inline`	∅	
14	519	`steamcommunity.com`	5		`unsafe-inline`	
15	558	`ya.ru`	4	`unsafe-inline`	∅	
16	643	`mega.co.nz`	0		✓	✓
17	695	`yandex.com.tr`	9	`unsafe-inline`	∅	
18	787	`behance.net` (R)	13	∅	∅	
19	801	`yandex.kz`	15	`unsafe-inline`	∅	
20	995	`chefkoch.de`	50	∅	∅	

Table 1: Alexa top 1,000 sites that deploy CSP and are protected against XSS. (AR=Alexa rank, R=report only, IS=Inline scripts, safe=protected against XSS)

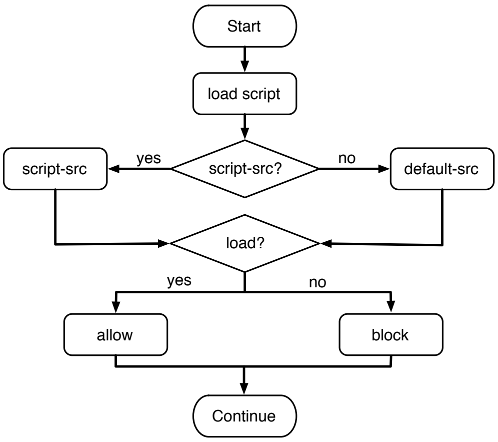

Figure 1: CSP policy enforcement in the browser.

implementation consults the `default-src` directive before making its decision whether to allow or block the load. If neither directive is present, CSP executes the script.

3.2 The State of CSP Deployment

In order to identify CSP headers deployed on the Web, we implemented a web crawler that automatically visits the Alexa Top 1,000 web sites using Firefox (v39.0). All the data in our paper reflect snapshots of frequently changing web pages, taken on January 16th and March 16th, 2015. The results of our crawl confirm our assumptions and overlap with the findings of Weissbacher et al. (Weissbacher et al., 2014).

As illustrated in Table 1, only 20 out of the top 1,000 sites in the world use CSP. The two sites `github.com` and `mega.co.nz` employ a fine-grained, secure CSP; neither of these two sites use inline scripts. Both sites define a `script-src` directive, and omit the keyword `unsafe-inline` to prevent execution of inline scripts. Such a tight and security–enhancing policy leverages the full potential of CSP: to prevent unauthorized script execution.

Unfortunately, the other 18 sites with CSP do not use its full potential. Those sites use the `unsafe-inline` keyword either in the `script-src` directive itself, or they do not specify a `script-src` directive at all and instead include `unsafe-inline` in the fallback directive, `default-src`. For example, the CSP of `yandex.ru` does not specify a `script-src` directive. Hence, before loading any of this site's 15 inline scripts, a browser's CSP implementation consults the `default-src` directive, where for the example case of `yandex.ru` it finds the keyword `unsafe-inline` and allows all inlined content scripts—including injected attack scripts—to execute.

Two sites, `behance.net` and `chefkoch.de`, use neither the `script-src` nor the `default-src` directive, which means that CSP will not pro-

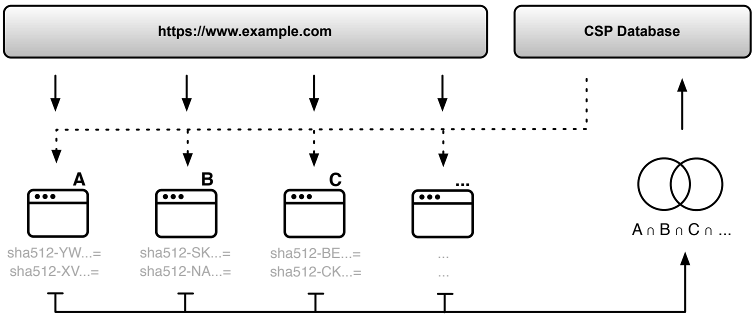

Figure 2: System overview: Various users (A, B, C) visit `https://www.example.com` and report hashes of executed inline scripts to a third party policy server which creates an intersection of all the reports and generates a CSP header. Future visitors to `https://www.example.com` benefit from the additional layer of security provided by a CSP header delivered to them from the third party.

tect them against any kind of script injection attacks. Note that `behance.net` deploys its CSP in report-only mode, meaning browsers will not actively block any resource loads or script executions, but will transmit reports for any detected CSP violations to a URI identified in the CSPs `report-uri` directive.

4 Design and Implementation

We implemented a prototype of our proposed client-server architecture for equipping every web site with a CSP using Firefox (v39.0). First, we patched *Gecko* (the rendering engine in Firefox) to record hashes of inline scripts found on each visited site. Our approach allows users to record inline scripts as they browse deeply into complex sites. The advantage of a crowdsourced approach like this lies in recording real-world interactions—including for parts of the site behind authentication—which a web crawler can not inspect.

4.1 Collecting Hashes

Figure 2 shows various users, each requesting a web site and viewing it in their browser (illustrated as A, B, C). Each user's browser records a `sha512`-hash of each inline script found on the page. Once a page is fully loaded (our prototype implementation listens to the `onload()` event handler), our modified browser asynchronously reports those hashes to a trusted third party policy server, which collects and evaluates all reported hashes. For example, user A navigates to `https://www.example.com` and encounters the following inline script:

```
<script>alert('Hello, world.');</script>
```

In such a case user A would record the hash of that script. Once the page is fully loaded, user A sends the gathered hashes of all the encountered inline scripts to a third party policy server (named CSP Database in Figure 2).

Later, user B also browses to `https://www.example.com` but views a different subpage and hence loads a different inline script. Eventually user C and various other users also visit `https://www.example.com`. All users record encountered inline scripts and send the collected hashes of loaded inline scripts to the third party policy server that processes all the reported hashes and assembles a CSP header for the entire web site.

4.2 Assembling a CSP Policy

Lets assume user A and various other users encounter the same script and report the same hash (`sha512-YWIzO...=`) to the policy server. Lets further assume that user B and various other users report the same hash (`sha512-XVHzN...=`) to the policy server. The CSP policy our system assembles and employs for that page resembles the following:

```
Content-Security-Policy: script-src * 'sha512-
    YWIzO...=' 'sha512-XVHzN...='
```

Please note that our injected CSP always contains the wildcard (`*`) within the `script-src` directive to allow all external scripts to load (Stamm et al., 2010; W3C - World Wide Web Consortium, 2014). For a detailed explanation of how our injection algorithm works, we defer the reader to Section 4.3.

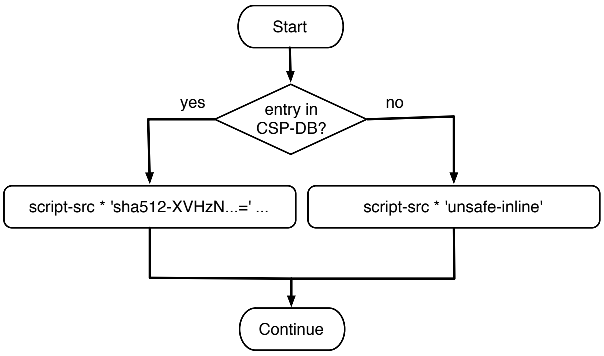
Start
yes
entry in CSP-DB?
no
script-src * 'sha512-XVHzN...=' ...
script-src * 'unsafe-inline'
Continue

Figure 3: Conservative CSP header generation.

When generating CSP headers our system needs to be *conservative*: an incorrect header that replaces `unsafe-inline` in the `script-src` directive that contains incorrect hashes will impact user experience. As a result, we implement a conservative CSP header generation strategy, illustrated in Figure 3.

Algorithm 1: Report inline script to CSP Database

```
Data: source url, hash h of an inline script
      or indicator of no scripts (ε), the
      reporting IP-Address, and threshold
      function T() to determine large
      enough sample size.
1  $S \leftarrow$ query(CSP-Database, url);
2  if $h \notin S$ then
3      $R \leftarrow$ query(CSP-Candidate-Database,
       url, h);
4      $S \leftarrow \omega$(distinct($R$, "IP")
       − IP-Address);
5      if $T(url) < |S| + 1$ then
6          delete(CSP-Candidate-Database,
           url, h);
7          update(CSP-Database, url, h);
8      else
9          update(CSP-Candidate-Database,
           url, h, IP-Address);
10     end
11 else
12     if $h = \varepsilon$ then
13         update(CSP-Database, url, ε);
14     end
15 end
```

Our system only generates and deploys a CSP header for a given site when we have collected a sufficiently large number of reports. Algorithm 1 shows our implementation of reporting inline scripts for each URL. The threshold function $T()$ controls how many reports we collect until we generate the CSP header, and is empirically determined (line 5). A scalar threshold value does not capture real-world Internet use, since the distribution of users usually is multimodal: there are pages with many users and pages with few users. For example, if our system were to uniformly use a high threshold value, pages with a recurrent user population of less than the threshold would never benefit from our system.

We compute our threshold T as follows. First, we collect user statistics on a per page basis to estimate its recurrent user population. Then we determine the sample size such that we have small confidence intervals. Finally, we set T to the high end of the confidence interval.

Our reporting procedure relies on two databases, the CSP Database and the CSP-Candidate-Database. The CSP Database holds hashes for URLs that have crossed the site-specific threshold value, whereas the CSP-Candidate-Database only holds reported hashes. Using two separate databases means that reports for sites that serve dynamic inline scripts will never cross their site-specific threshold T, because the reported hashes constantly change. As a result, there will be many reported entries in the CSP-Candidate-Database with different hash values.

We count reports for each IP-address only once, which prevents "spamming" the system with spurious reports from a single IP-address (see `distinct` on line 4). Furthermore, Algorithm 1 also shows how to weigh reported hashes to prevent tampering with our system (function ω on line 4). For example, a serious attacker might attempt to use a botnet to report hashes of an injected script. In general, there are two strategies to counter tampering attempts. First, ω could use statistical techniques to measure reports. One could, for instance, take a random sample of all incoming reports and see whether the sample exceeds our threshold $T()$. Another statistical procedure would be to measure all reports to identify and exclude outliers. Second, we could use third party information sources to discard spurious reports from the CSP-Candidate-Database. For example, both Google and Microsoft maintain databases indicating whether certain sites are suspicious or not. Our system could take entries blacklisted by either initiative and sanitize candidate records.

Algorithm 1 shows on lines 12-14 how our system handles pages that contain no inline scripts.

Algorithm 2: Generate CSP header

```
   Data: source url and threshold function
         $T()$
   Result: a script-src directive for the
           specified url
1  $S \leftarrow \emptyset$ ;
2  $R \leftarrow$ query(CSP-Database, url);
3  $R' \leftarrow$ query(CSP-Candidate-Database, url);
4  for $r \in R$ do
5      $S \leftarrow S \oplus r$;
6  end
7  if $S \neq \emptyset \wedge |\omega(R')| < T(\text{url})$ then
8      return "script-src:  * " $\oplus S$;
9  else
10     return "script-src:  *
       'unsafe-inline'";
11 end
```

Algorithm 2 shows how we generate a CSP header for a page. We generate a functional CSP `script-src` directive for each reported hash of an inline script when there are no pending reports in the CSP-Candidate-Database. This conservative strategy ensures that we only generate a CSP header when the system reaches a stable state, where the CSP-Candidate-Database contains no significant reported hashes anymore. This stable state indicates that a page only serves static inline scripts, all of which have been reported sufficiently often and successfully passed the statistical safeguards implemented in our weighing function ω.

Generating a CSP header using our system is also compatible with updates to a page. Whenever a developer changes his page to serve a new inline script, our system will trigger new reports to the CSP-Candidate-Database. While the weighing function ω indicates that we have seen fewer reports than the threshold function $T()$ requires, we will continue to generate the old CSP header (see line 7). Once we cross the threshold, however, there are two possible cases. First, Algorithm 1 will add the new hash h to the CSP Database and delete pending records from the CSP-Candidate-Database. Second, the new script is dynamic and the hashes collected in the CSP-Candidate-Database will exceed the threshold, causing Algorithm 2 to render the permissive `unsafe-inline` keyword instead. The condition also means that whenever an author changes a page, our system requires at least an amount of threshold function $T()$ reports until it can make a decision.

4.3 Injecting CSPs

Whenever a user browses to a web site using our modified CSP enhancing web browser, our system performs one of the following three actions which allows our system to provide the most fine-grained CSP header possible for that site:

(1) Webpage does not deploy CSP: In case the web site does not deploy a CSP header, then our modified browser simply queries the corresponding CSP for that webpage from the CSP Database. The browser applies the CSP as if a CSP header was received from the web page itself.

(2) Webpage deploys CSP with `unsafe-inline`: If the web page deploys a CSP but uses the keyword `unsafe-inline` within the `script-src` or the `default-src` directive, then our system removes the keyword `unsafe-inline` and replaces `unsafe-inline` with the script hashes obtained from the CSP Database. Additionally specified directives within the CSP of the webpage, such as `img-src`, `style-src`, or `connect-src` (Stamm et al., 2010; W3C - World Wide Web Consortium, 2014) remain untouched by our injection algorithm since such directives provide additional security for the webpage. We consider crafting a tight CSP policy as good practice and hence our prototype always injects the script-hashes within the `script-src` directive, even if the original CSP does not contain a `script-src` directive and `unsafe-inline` was defined within the fallback directive: `default-src`.

(3) Webpage deploys CSP without `unsafe-inline`: If a webpage deploys a CSP, and that policy does not include `unsafe-inline` within the `script-src` or the `default-src` directive, then our system does nothing to change the user's interaction with the site. In this rare case where a webpage crafts its own tight CSP policy, our modified browser does not perform any additional actions other than enforcing the CSP policy like any other browser that supports CSP. As discussed in Section 3.2, we encountered such a tight policy only on `github.com` and `mega.co.nz`.

Intermittently Blocking Valid Content Scripts: As discussed in Section 4.2, our system is able to respond site changes (specifically changes to inline scripts) very quickly since hashes for inline scripts are reported by our system's users at a constant pace. In turn the CSP generated for tracked sites is regularly updated with new information. However, there is a short window of time while the system updates policies after site changes, and it is possible our system may block a legitimate (non-attack) inline script. In this case, the user has the option to manually override the injected CSP.

5 Evaluation

We examine the capabilities and limitations of our system by monitoring the Alexa top 1,000 sites over a period of two months (January 16th to March 16th, 2015) and record the number of inline scripts per site.

In this section we discuss the feasability of our approach by inspecting the overall usage of inline scripts on the Alexa top web sites (Section 5.1) and discuss tradeoffs of our approach for pages using frequent updates of their inline scripts over pages that rarely update their inline scripts (Section 5.2). Finally, we compare the spectrum of effectiveness of our approach by providing recommendations for web sites on the lower end and highlight the practicallity of our approach for web sites on the upper end of that spectrum.

A word about our web crawler: Unfortunately distribution of our prototype to gather statistical data for a real world study is not feasible. Hence we set up a web crawler that visits the front page of the Alexa Top 1,000 sites and records all inline scripts for those sites. Even though a web crawler can not inspect the deep web hidden behind logins we argue that a web crawler returns sufficient data to evaluate inline script usage, record the code change of inline scripts and therefore allows us to reason about the feasibility of our approach. Note that a toplevel CSP does not apply to content in subelement `iframes` (Stamm et al., 2010; W3C - World Wide Web Consortium, 2014). Our crawler supresses the loading of each page's `iframes` to avoid polluting our statistics with incorrectly accounted inline scripts.

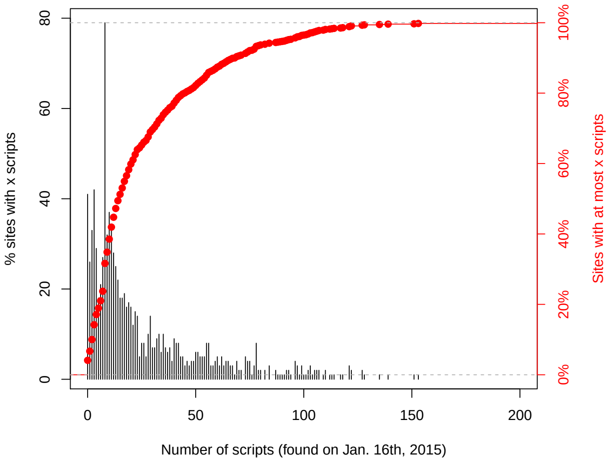

Figure 4: Number of sites with a given number of scripts and the CDF (Cumulative distribution function): percent of all sites with at most a given number of scripts.

5.1 Inline script usage

In sum, our web crawler detected 26,743 inline scripts when visiting the top 1,000 sites on January 16th, 2015. This indicates that every site hosts on average 27 inline scripts, as illustrated in Figure 4.

Unique scripts: Although our crawler detected 26,743 instances of inline scripts, we found only 16,519 unique scripts. We attribute this phenomenon to web developers frequently including third-party functionality such as analytics and/or social media code to enrich a user's browsing experience. Quite commonly such small scripts are copied and pasted, and inlined directly within a page. For example, we found the same user agent detection script on 393 out of the Alexa top 1,000 sites. Once copied and pasted on a page, these inline scripts do not get updated by the web site author. This complements our approach and causes fewer intermittent script blockages as explained in Section 4.3.

Pages hosting many inline scripts: We found two sites that are using more than 200 inline scripts. The pages `repubblica.it` and `buzzfeed.com` use 278 and 217 inline scripts, respectively. Those two pages are statistical outliers, but we observed 415 sites that use between 20 and 199 inline scripts.

As a general recommendation for sites with more than 20 inline scripts: we suggest refactoring such pages to load most of the scripts from external resources. We understand the performance cost of loading JS from an external resource rather than inlining the script (Grigorik, 2013), but we argue that using more than 20 inline scripts will be performance–hampering for a web page.

Pages hosting few inline scripts: In contrast to sites using many inline scripts, we identified 583 sites that employ fewer than 20 inline scripts. 43 of those sites do not use any inline scripts at all. The architecture of such web pages is ideal for our approach, because our system uses a CSP header of '`script-src *`' for all of those 43 pages, allowing all external scripts to execute but preventing all inline scripts and thus most XSS attempts.

5.2 Static vs Dynamic Scripts

In this section we examine inline scripts on web sites during our monitoring phase and highlight the fraction of inline scripts that we consider static vs. the fraction of inline scripts we consider dynamic.

Definition of static and dynamic scripts: *We define an inline script to be static if the script does not change its contents across time points in our investigation. We further consider a script to be dynamic if the script contents gets updated iteratively and changes across differently-timed accesses.*

Figure 5 illustrates the ratio of static to dynamic scripts on the Alexa top 1,000 sites. On January 16th, we detected 26,743 inline scripts. On March 16th, we detected 27,268 scripts. When doing our second crawl we compared the hashes of the scripts of the two runs and noticed that 10,893 scripts did not change within those two months. In other words, each site in the Alexa top 1,000 sites hosts on average 27 scripts, from which we consider 11 to be static and 16 to be dynamic, or iteratively updated.

Pages with most dynamic scripts: Figure 6 (left) shows the ten pages with the most dynamic scripts encountered on the top 1,000 pages (changed between the two visits by our web crawler). Our recordings do not show any static scripts on `bloomberg.com` as well as `vitaminl.tv` and both pages make use of 60 and 90 inline scripts respectively. Our system may intermittently block legitimate scripts that are updated frequently on such pages. Since all of these ten web pages make extensive use of dynamic scripts, the CSP Database might not have generated an updated CSP header for such a site before a subsequent user visits the page and hence queries and outdated CSP from the CSP Database. As discussed in Section 4.1, a user of our modified web browser can manually override the injected CSP and allows all inline scripts to run.

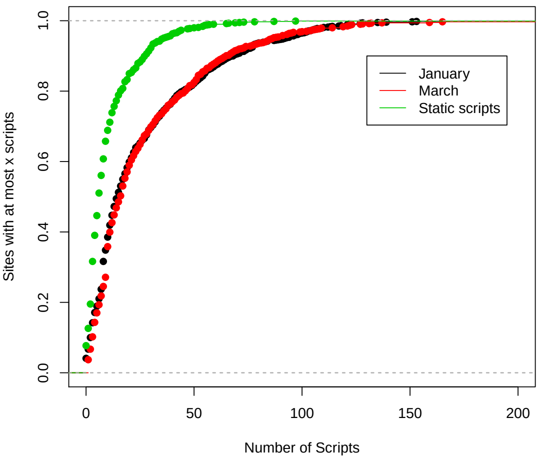

Figure 5: CDF for the number of sites with at most a given number of scripts for January to March 2015

To overcome the problem of intermittently blocked scripts, we recommend that pages hosting more than 20 inline scripts refactor their pages and load most (if not all) of their inline scripts from an external resource. Loading scripts from an external resource has the advantage that web site authors only have to whitelist the URL where scripts are loaded from once (using the `script-src` directive) instead of constantly updating script hashes within the CSP header whenever a script gets updated.

Pages with most static scripts: Figure 6 (right) shows the ten web pages relying on static inline scripts and few dynamic inline scripts. For example, the page `chase.com` uses 27 static inline scripts and the page `caixa.gov.br` uses 29 static inline scripts. Both pages use only one dynamic inline script that changed its hash value between our two visits by our crawler. Pages like the ten highlighted in Figure 6 (right) as well as pages that do not include any inline

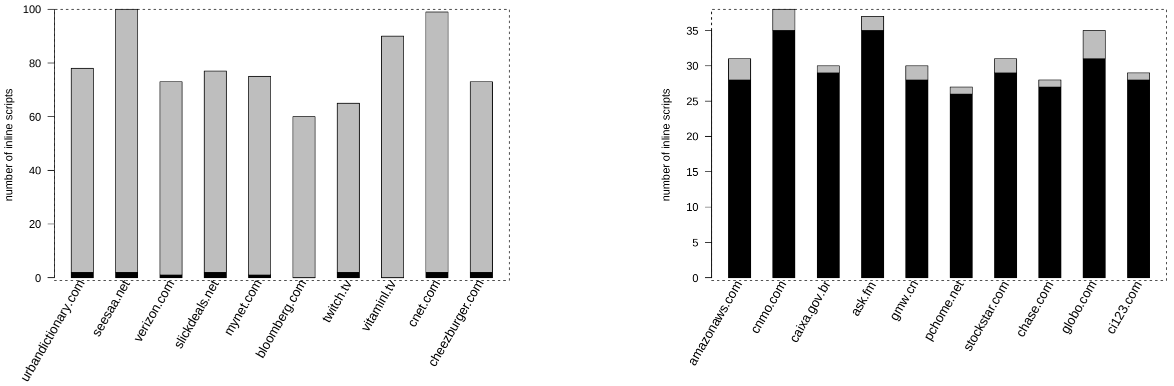

Figure 6: (left) Sites with the most dynamic scripts. (right) Sites with the most static scripts on the Alexa top 1,000.

scripts at all (discussed in Section 5.1) exhibit ideal conditions where our approach of injecting a CSP header substantially increases protection against script injection attacks.

Customized inline scripts per visitor: To evaluate our approach we must also account for webpages that dynamically generate content on the server and serve that content in form of inline scripts to the visitor. We performed yet another crawl of the Alexa top 1,000 on March 16th, 2015, to evaluate how frequently this occurs. We use the same setup as for the first crawl but use a different IP-address to evaluate server side inline script customizations on pages. Overall, we encountered 4,194 scripts generating a different hash for every visitor. We manually inspected scripts that differ and found that such scripts' functionality is mostly identical but such scripts serve data in addition to functionality.

For example, the page `picmonkey.com` serves nine inline scripts, where eight of them are identical for two visitors. One script is different for every visitor because it includes the following code line: '`servertime: new Date(123456789)`', where `123456789` is the time specified by the server before it is shipped to a client. Inline scripts in this fashion will differ for every user and so every visitor reports a different hash for such a script to the policy server. Unfortunately, pages that serve inline scripts that not only contain functionality but also data are not very well suited for our approach. As explained in Section 4.2 our approach will not generate a CSP header for such pages because reported hashes will never make its transition from the CSP-Candidate-Database into the CSP Database.

Feasibility of CSP header injection: We found that 483 pages serve the same inline scripts for different visitors which makes those 483 pages perfectly suitable for our approach. Those 483 pages serve only functionality within their scripts and such scripts' hash value only changes if indeed an iterative update of the code within the inline script happens—which our system recognizes and explicitly supports. Based on these promising results, we are confident that our system scales beyond the Alexa top 1,000 and will bring about an important improvement in web security without requiring extensive manual intervention from page authors. Given that currently only two percent of pages deploy CSP, any increase enabled by our system—ideally to the reported 50%—will be a big win.

6 Discussion

Though the research community has not explored areas of configuration and software package management with CSP, they are vitally important to deployment. A stable toolchain (compiler, linker, package installer) is key in creating systems that are easy to deploy.

Today, end users miss out on the increased security guarantees provided by CSP. As discussed, in Section 3 we find that only two of the Alexa top 1,000 sites deploy CSP that has any XSS prevention benefit. In this paper we present a system that allows web sites to be deployed with CSP not relying on web site authors to provide, update and maintain their own CSP policy.

Approach limitations: Our system helps prevent the most common type of XSS, namely nonpersistent (or reflected) XSS (Hope and Walther, 2008). Exploiting a reflected XSS vulnerability allows an attacker to inject JS code, most commonly in HTTP query parameters or in HTML form submissions, which the web page immediately renders as regular page content, granting the script access to user sensitive information on that page. As explained in Section 4.2, our system only whitelists and adds script hashes to the CSP Database if hashes are reported by various users and only once a pre-defined number of reports for a given script is reached.

Our system further relies on web sites performing standard user input sanitization(Jovanovic et al., 2006; Balzarotti et al., 2008; Bisht and Venkatakrishnan, 2008) before storing any user data within the sites internal database. Once malicious user input bypasses server side sanitization filters, our system can not distinguish between legitimate content scripts and injected scripts. Such a persistent XSS vulnerability can not be detected or stopped by our approach and the hash for the malicious inline script will be whitelisted and added to the CSP Database once reported by enough users.

Prototype limitations: A full–scale deployment of our system should use standard anonymization techniques so as not to reveal the content and source of reports sent to the third party policy server. For example, a production implementation could use a traffic anonymization system such as TOR (The Tor Project, 2012), or rely on RAPPOR (Erlingsson et al., 2014), a technology that allows anonymous crowdsourcing statistics from end-user client software.

At the moment our approach focuses on XSS prevention and to eliminate `unsafe-inline` from CSP headers and does not cover other protections offered by CSP. Future work would cover more of the CSP directives not aimed at preventing XSS. For example, the auto-generated CSP could cover the `img-src` directive so that images can only be loaded from expected resources. Covering more CSP directives and hence allowing resource loads only from expected origins could additionally increase security by not allowing attackers to perform requests to their own servers.

7 Related Work

CSP Analysis: As discussed in Section 3, our findings align with the presented results of Weissbacher et al. (Weissbacher et al., 2014). In their paper they analyze the top one million Alexa sites' CSP deployment. Their work shows that one percent of web pages in the Alexa top 100 deploy CSP in enforcement mode and conclude that CSP will not be easily deployed without substantial framework support.

Security tools relying on CSP: In 2013, Doupé et al. (Doupé et al., 2013) present a system called *deDacota*. Using static analysis their system automatically separates code and data of web applications and relies on CSP to enforce that separation in the browser.

In 2015, Schwenk et al. (Schwenk et al., 2015) present *JSAgents*. Even though their approach is not relying on CSP per se, they use their presented *JSAgents* to implement large portions of CSP with the intent to fine grain CSP so that every element in the DOM relies on a JSAgent instead of having the full document rely on one CSP. Similar to our intention, both of these approaches aim to secure legacy web applications.

The framework "Confinement with Origin Web Labels" (COWL (Stefan et al., 2014)) uses CSP as the foundation and provides a labeling mechanism on top of CSP and other security features within a browser.

Distributed Analysis: In 2011, Greathouse et al. (Greathouse and Austin, 2011) show that a large population, in aggregate, can analyze larger portions of a program than any single user individually running the full analysis of a program.

In 2013, Kerschbaumer et al. (Kerschbaumer et al., 2013) present a crowd sourced based approach where individuals report security violations to a third party system which allows to identify malicious web pages and feed the results to URL blacklisting services such as Microsoft's smartscreen filter (Microsoft, 2012) or Google's safebrowsing (Provos, 2012) initiative.

Third-Party Security systems: In 2011, Thomas et al. present a system called Monarch (Thomas et al., 2011) and Canali et al. present a system called Prophiler (Canali et al., 2011). Both approaches aim to detect malware on the Web relying on machine learning techniques. Even though our system does not try to classify malicious webpages, it fits well within this theme of letting trusted third party systems provide security features for the web.

8 Conclusion and Outlook

Today, web sites and their users do not benefit fully from XSS protection offered by CSP. Even though most major browsers acknowledge the CSP header, only 20 out of the Alexa top 1,000 sites deploy CSP, and only two pages effectively use CSP to protect their users against code injection attacks and XSS. It is clear from our findings that industry requires better framework support for easy and efficient deployment of CSP with a web page.

Our proposed system allows deployment of CSP for web sites without requiring web site authors to manually update and maintain their CSP header whenever they perform any kind of update on their page.

We discussed the limitations of our approach to account for pages that not only ship functionality but also data within their inline scripts. Our preliminary results show that our approach is feasible to deploy a CSP header for up to half of web sites on the Internet and thus help protect them from XSS attacks.

ACKNOWLEDGEMENTS

Thanks to the everyone in the Security Engineering Team of Mozilla for their feedback and insightful comments.

REFERENCES

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 387–401. IEEE.

Bisht, P. and Venkatakrishnan, V. (2008). Xss-guard: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer.

Canali, D., Cova, M., Vigna, G., and Kruegel, C. (2011). Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the international conference on World wide web*, pages 197–206. ACM.

Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C., and Vigna, G. (2013). dedacota: toward preventing server-side xss via automatic code and data separation. In *CCS*, pages 1205–1216. ACM.

Erlingsson, Ú., Pihur, V., and Korolova, A. (2014). RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1054–1067. ACM.

Greathouse, J. L. and Austin, T. (2011). The potential of sampling for dynamic analysis. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 3:1–3:6. ACM.

Grigorik, I. (2013). *High Performance Browser Networking*. O'Reilly.

Hope, P. and Walther, B. (2008). *Web Security Testing Cookbook*. O'Reilly.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 6–pp. IEEE.

Kerschbaumer, C., Hennigan, E., Larsen, P., Brunthaler, S., and Franz, M. (2013). CrowdFlow: Efficient information flow security. ISC, Springer.

Microsoft (2012). Microsoft security intelligence report, volume 13. `http://www.microsoft.com/security/sir/default.aspx`. (checked: August, 2015).

Microsoft (2012). SmartScreen Filter. `http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter`. (checked: August, 2015).

Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. (2012). You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM.

OWASP (2012). The open web application security project. `https://www.owasp.org/`. (checked: August, 2015).

Provos, N. (2012). Safe browsing - protecting web users for 5 years and counting. `http://googleonlinesecurity.blogspot.com/2012/06/safe-browsing-protecting-web-users-for.html`. (checked: August, 2015).

Russo, A., Sabelfeld, A., and Chudnov, A. (2009). Tracking information flow in dynamic tree structures. In *Proceedings of the European Symposium on Research in Computer Security*, pages 86–103. Springer.

Schwenk, J., Heiderich, M., and Niemietz, M. (2015). Waiting for CSP: Securing Legacy Web Applications with JSAgents. In *Proceedings of the European Symposium on Research in Computer Security*, page TBA. Springer.

Stamm, S., Sterne, B., and Markham, G. (2010). Reining in the web with content security policy.

In *Proceedings of the ACM International Conference on World Wide Web*, pages 921–930, New York, NY, USA. ACM.

Stefan, D., Yang, E. Z., Marchenko, P., Russo, A., Herman, D., Karp, B., and Mazieres, D. (2014). Protecting users by confining javascript with cowl. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

The MITRE Corporation (2012). Common weakness enumeration: A community-developed dictionary of software weakness types. `http://cwe.mitre.org/top25/`. (checked: August, 2015).

The Tor Project (2012). Tor (anonymity network). `https://www.torproject.org/`. (checked: August, 2015).

Thomas, K., Grier, C., Ma, J., Paxson, V., and Song, D. (2011). Design and evaluation of a real-time url spam filtering service. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 447–462.

W3C - World Wide Web Consortium (2004). Document object model (DOM) level 3 core specification. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf`. (checked: August, 2015).

W3C - World Wide Web Consortium (2014). Content Security Policy Level 2. `http://www.w3.org/TR/CSP2/`. (checked: August, 2015).

Weissbacher, M., Lauinger, T., and Robertson, W. (2014). Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Research in Attacks, Intrusions and Defenses*, volume 8688, pages 212–233. Springer International Publishing.