Ads, Analytics, & CSP | How the Prevalence of Third Party Scripts Precludes XSS Protection

Trevor Elwell
UC San Diego
telwell@eng.ucsd.edu

Chris Lamb
UC San Diego
c2lamb@eng.ucsd.edu

Kaiser Pister
UC San Diego
kpister@eng.ucsd.edu

ABSTRACT

Content Security Policy (CSP) is a secondary layer of security that helps to detect and mitigate certain classes of attacks, including Cross Site Scripting (XSS) and data injection attacks. In this paper, we evaluate the real-world CSP deployments utilized by the Alexa top 1 million sites [2]. Our crawl of the Internet finds 39,022 hosts using the `Content-Security-Policy` header deploying 6,670 unique CSPs. We perform a quantitative analysis of these policies focusing on the aspects of the CSP specification which are relevant to XSS protection. We identify directive usage trends compared to prior analyses [5, 14], as well as commonly whitelisted values. We identify policies which provide no XSS protection at all, namely those containing the `'unsafe-inline'` keyword in the `'script-src'` directive and examine the functionality of sites using those policies. We determine how many of these policies could be improved by using more secure sources, namely `'sha256'`, `'nonce'`, or `'strict-dynamic'`, while maintaining the original functionality of the sites. Finally, we address the largest obstacle to more secure CSPs: ads and analytics.

KEYWORDS

Content Security Policy; Cross-Site Scripting; Web Security

1 INTRODUCTION

Content Security Policy (CSP) is a security mechanism enforced by browsers to help mitigate the dangers of cross-site scripting (XSS), fraudulent clicks, and mixed content attacks[13]. Of the three, arguably the security community cares most about XSS attacks as they are some of the most prevalent types of vulnerabilities [9] and new types of XSS attacks are being created as the web grows [3]. In fact, MITRE's CVE database shows that as of December 2018 there are 12,724 vulnerabilities with "XSS" in the title, as compared to 2,075 for cross-site request forgery ("CSRF"), a similar style of attack. According to [13] CSP is not intended to be first line of defense against content injection vulnerabilities. Instead, CSP is best used as defense-in-depth, to reduce the harm caused by content injection attacks. As a first line of defense against content injection, server operators should validate their input and encode their output.

Though they are not trivial to implement, CSPs provide a good secondary line of defense against XSS attacks. At its core, CSP allows hosts to indicate which sources they trust to load resources onto their webpages. If a CSP is configured and a website tries to load an a resource from an untrusted source the browser will prevent that resource from loading. All major browsers now support the `Content-Security-Policy` response header [10] and when implemented properly, CSPs can prevent unintended and malicious resources from loading onto a webpage.

We crawled the Alexa Top 1 Million sites [2] to generate a representative sample of how CSPs are utilized in practice. We found that of the 1 million most visited sites, only 3.9% include a CSP header at all and only 18.75% of those restrict the source of JavaScript content. Worse still, 90.69% of the sites which do include the `'script-src'` directive constraining JavaScript usage allow `'unsafe-inline'` scripting on their webpage negating any benefits of enforcing a CSP [8]. Our research shows that at least 99.93% of all websites either lack CSP or misuse them to the point where they provide no protection from XSS attacks.

To remedy the misuse of CSP, previous works have advocated the use of the `'strict-dynamic'` source for the `'script-src'`[14] directive in situations where `'unsafe-inline'` was being used. `'strict-dynamic'` is more restrictive, and accordingly more secure, than `'unsafe-inline'` because it does not allow scripts to run unrestricted, rather it uses the idea of *transitive trust* to vet a root script via a cryptographic nonce or hash and then allows the root script to load other scripts unhindered. In other words: "trust explicitly given to a script present in the markup, by accompanying it with a nonce or a hash, shall be propagated to all the scripts loaded by that root script"[11]. Since only the root script requires verification, the authors of `'strict-dynamic'` believe adoption will be significantly easier in comparison to stricter source requirements, however in our data, we find that use of `'strict-dynamic'` is not prevalent. Only 0.96% of sites which have a `'script-src'` policy use the `'strict-dynamic'` keyword.

Though `'strict-dynamic'` can be useful in many scenarios, we find the trade-off between security and usability to be lacking for the majority of websites. For many sites, `'strict-dynamic'` is no easier to implement than a nonce-based policy, which would prevent *transitive trust* from propagating outside of a set of distinctly named scripts. Similarly, with little effort, almost 30% of websites could implement a hash based CSP which would only allow a specific set of scripts content to be loaded. These script source values are more prevalent than `'strict-dynamic'`, with `'nonce'` being included in 6.57% of policies and `'sha256'` being included in 7.95%.

Perhaps our most striking result from a security standpoint is the prevalence of `'unsafe-inline'` and `'unsafe-eval'`. In seeming disregard for the word "unsafe" in their title, we found that they were used in 90.69% and 86.33% of policies respectively. To put that into perspective, having either of these sources as part of your `'script-src'` directive removes almost all of the benefits of declaring a content-security policy.

Our research makes the following contributions:

, ,
2018. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of* , , https://doi.org/10.1145/nnnnnnn.nnnnnnn.

- We provide an analysis of the Alexa Top 1 Million websites' content security policies as of December 2018. This data gives a glimpse into the adoption rates of CSP and the effectiveness of the current corpus of policies.
- We provide a breakdown of the insecure CSPs (specifically websites which use `'unsafe-inline'`) and the sources and types of JavaScript those websites load. With this breakdown we then advocate how these policies could be hardened to reduce their attack surface and trust-base at limited cost to the developers.
- Finally we present a discussion of advertisement delivery networks presently, why their current state makes restrictive CSPs difficult, and provide recommendations for how the nature of ads could be changed to make CSPs more effective.

2 BACKGROUND

Code injection from cross-site scripting attacks will render many browsers' security models useless, allowing attackers arbitrary control over private users' data. To a first approximation, in order to hack the browser, the attacker will insert their code into the server response, tricking the browser into rendering the malicious payload. Ideally, the same-origin policy would prevent any execution of unknown scripts, however from the perspective of the browser the injected code came from the server itself, and therefore should be trusted. A more in-depth description of the XSS attack is laid out by the Open Web Application Security Project [12].

2.1 Content Security Policy

As a secondary layer of protection from XSS attacks, developers can add the `Content-Security-Policy` header to their sites indicating which types of resources can be loaded from which sources. The form of these policies is as follows:

```
Content-Security-Policy: <directive> <sources>;
```
[10]

where each `directive` establishes a rule that strictly defines the nature of resources the browser can load for a given page. For example, if the origin `bank.com` includes:

```
Content-Security-Policy: script-src 'self';
```

in a response header, then the browser will refuse to load any script resources from an origin other than `bank.com`.

2.2 Directives

There are currently 23 supported CSP directives [10]. We provide descriptions of the directives relevant to our analysis below:

- **`upgrade-insecure-requests`:** Attempts to upgrade all http:// resource requests to https://, blocking any resources it fails to upgrade.
- **`block-all-mixed-content`:** Blocks any http:// requests from an origin with the https:// scheme. This directive is subsumed by `upgrade-insecure-requests`.
- **`frame-ancestor <sources>`:** Establishes the set of parents which can embed into the current page. Valid sources include host origins, schema, `'self'` and `'none'`.
- **`script-src <sources>`:** Defines the valid sources of scripts which can be loaded in the current page. Valid sources include hosts with or without wildcards, schema, `'self'`, `'none'`, `'nonce-<value>'` `'<hash-algorithm>-<value>'`, `'unsafe-inline'`, and finally `'unsafe-eval'`. JavaScript from a source outside of the list provided in the header will be ignored by the browser.
- **`default-src <sources>`:** Serves as a fallback for any resource type not restricted by another directive included in the policy, `default-src` uses source types as `script-src`.

2.3 Sources

Along with common CSP directives, it is necessary to understand some of the most common CSP source types. For the purposes of this paper, these source types are described relative to the `script-src` directive. Though other directives may also utilize them.

- **`self`:** Including `'self'` whitelists all scripts from the origin of the loading page.
- **`host`:** Including host names with or without wildcards whitelists all scripts from originating from that host.
- **`<hash-algorithm>-<value>`:** Hashing is useful when you want to ensure that you are only loading **a specific** script. In this situation, you can run the content of the resource through a hashing algorithm (either `sha256`, `sha384`, or `sha512`) and adding this hash to your CSP. This is useful if, for example, you know you want to load a specific version of `jQuery` which you know will not change on subsequent loads. It's critical to note that in CSP 2.0 *hashes only work with inline scripts*, while in CSP 3.0 allows hashing with external sources [10].
- **`nonce-<value>`:** A nonce is used to specify a script that you would like to whitelist in your CSP. It is useful for dynamic scripts, because you need not know the exact content of the script at loadtime. For instance, if you know that you want to include `b.com/js/safe.js` on your page. You could then load it using the same code as above:

  ```
  <script nonce="1234" src="b.com/safe.js"></script>
  ```

  It's important to note here that `safe.js` is restricted from adding other resources to the page. But, since this is a nonce the exact content of `safe.js` can change.
- **`strict-dynamic`:** Using the `'strict-dynamic'` keyword allows trust to be propagated from a trusted resource to additional resources which it loads. This is best illustrated with an example. If `a.com` includes the following CSP header:

  ```
  Content-Security-Policy: script-src 'nonce-1234';
  ```

  and loads JavaScript via the following code:

  ```
  <script nonce="1234" src="b.com/safe.js"></script>
  ```

  then the resource from `b.com` will be loaded without a problem.[1] `strict-dynamic` comes into play when the resource from `b.com` attempts to load another resource from `c.com`. If `strict-dynamic` is not in our sources, then this request will be blocked as the resource from `c.com` has not been explicitly whitelisted. If the `strict-dynamic` keyword is included then we are allowing our trust in `b.com` to carry over to resources it loads; which

[1] Though a nonce value just has to be a base64 encoded string, it is critical that the nonce change on each page load and be cryptographically secure. Otherwise guessing the nonce and thus bypassing the CSP becomes trivial. '1234' is used as an educational example and should obviously not be used in the wild.

in this case is the resource from `c.com`. In other words, `strict-dynamic` allows you to vet a "root" script using a `nonce` or a `hash`. That "root" script then has the ability to load other resources unmolested.[2]

- **`unsafe-inline`:** Any inline script is allowed. This is very insecure.
- **`unsafe-eval`:** Allows the use of `eval()` in allowed resources. This is also unsafe as it allows scripts to inject any JavaScript onto the page, bypassing any CSPs currently in place.

3 METHODOLOGY

In order to obtain our data set, we performed a simple crawl of the Alexa top 1 million sites [2] using a headless browser in order to collect the `Content-Security-Policy` headers present in each request. With this data, we normalized the policies removing unnecessary white space and sorting both the directives and their associated values in lexicographical order. Additionally, we normalized the `'nonce'` and `'sha256'` values by removing the unique content of the nonce and hash values. After identifying 5,240 sites from our initial corpus that returned CSP headers with the `'script-src'` directive and the `'unsafe-inline'` keyword we revisited those sites, this time using a proxy to capture the content of all scripts loaded by each site. We constructed a graph whose vertices are the origins of all loaded scripts and whose edges represent the loading of a script by another script. Finally, we visited the same 5240 sites again the following day and in order to determine whether each script changed across visits providing and estimate of the percentage of scripts which are dynamically generated.

3.1 Limitations

The Internet is constantly growing and evolving and as a result our data represents only a snapshot of a small portion of the greater Internet. Increasingly websites serve different content based upon a visitors browser cookies, device type, and other factors. Our freshly instantiated, i.e. without any cookies, headless browsing crawler may receive different content than the average visitor to the same websites. Additionally, for any given website, we browse only the landing page, not exhaustively exploring all links on each page.

4 RESULTS

Out of the 1 million sites that we visit 39,022 or approximately 3.9% return the `Content-Security-Policy` header and in total we find 6,670 unique normalized policies. CSP usage is trending upward increasing from 1.6% of sites in 2016 [14] to 1.97% in 2017 [5] to 3.9% in 2018. The 2016 number is based upon a Google search index of approximately 1 billion sites while the 2017 and 2018 numbers are both based upon the Alexa top 1 million sites at the time. The usage frequencies of the various directives are displayed in figure 1. Our data indicates that the most common usage of CSP today is to require the usage of HTTPS. The `'upgrade-insecure-requests'` directive is the most commonly used directive appearing in 56.74% of policies. This represents a dramatic change from the 2016 data from Weichselbaum et al. [14] where it appeared in only 1.87% of policies. The adoption of forced HTTPS is certainly a positive security trend. The `'frame-ancestors'` directive has also gained popularity appearing in only 8.11% of policies in 2016 while appearing in 54.07% of policies in 2018.

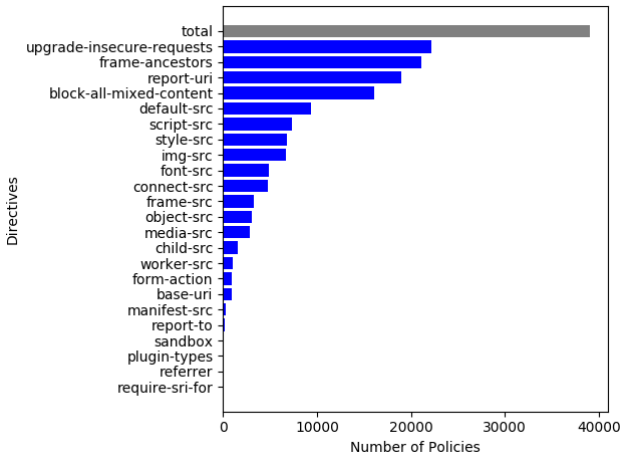

Figure 1: Distribution of CSP directives.

The directive that primarily deals with XSS mitigation is the `'script-src'` directive. In our data this directive appears in 7,317 or 18.75% of policies. Figure 2 shows the frequencies of various types of values included in policies with the `'script-src'` directive. The `'unsafe-inline'` keyword which renders the policy entirely useless for XSS protection appears in 90.69% of policies. The most secure keyword `'sha256'` appears in 7.95% of policies in our data set, up from just 1.65% in 2016. The next best alternative `'nonce'` appears in 6.57% of policies up from 0.92% in 2016. The use of wildcards in sources also presents security risks. Only 18.38% of policies that specify at least one host include exclusively hosts with fully specified paths. Of all policies with the `'srcipt-src'` directive 38.28% include at least one host with a wildcard as and 18.03% include the general wildcard. Whitelisting entire protocols, i.e. https:// or http://, is also extremely insecure and 59.87% of policies include at least one protocol among their sources.

Figure 3 shows the most commonly whitelisted hosts. Google's pervasiveness on the Internet is apparent as 13 of the top 15 most white-listed hosts are Google domains with the remaining 2 coming from Facebook.

Figure 4 shows the origins of the scripts most commonly requested by other third party scripts. The most common use cases for scripts loading other scripts are advertisements and tracking. These account for 59.11% of scripts loaded by other scripts. We define ads and tracking scripts based upon the uBlock Origin [6] default filter settings.

The loading of dynamically generated scripts prevents the use of the most secure keyword, `'sha256'`. According to our data of the 37,087 scripts loaded by sites that include the `'script-src'` directive and the `'unsafe-inline'` keyword in their CSPs 13.06% are dynamically generated and 43.03% of sites load at least one dynamically generated script.

[2]It's important to note that since `eval` is blocked by default when using CSP, in order for the script from `b.com` to inject new JavaScript resources onto the page it will need to use some other API method such as `document.createElement()` which will then need to be added to the `document` through `document.body.appendChild()` or similar.

Count	Percentage	script-src value
6636	90.69%	'unsafe-inline'
6317	86.33%	'unsafe-eval'
4623	63.18%	'self'
582	7.95%	'sha256'
481	6.57%	'nonce'
91	0.97%	'strict-dynamic'
2801	38.28%	host with wildcard
1345	18.38%	host with path
1319	18.03%	general wildcard
1670	22.82%	https:
1089	14.88%	data:
850	11.62%	blob:
563	7.69%	about:
113	1.54%	http:

Figure 2: Most common features in script-src.

Count	Percentage	script-src value
982	13.42%	www.google-analytics.com
973	13.30%	https://www.google-analytics.com
942	12.87%	*.google-analytics.com
824	11.26%	*.google.com
801	10.95%	*.googleapis.com
768	10.50%	https://www.googletagmanager.com
592	8.09%	*.gstatic.com
586	8.01%	connect.facebook.net
542	7.41%	www.googletagmanager.com
527	7.20%	https://connect.facebook.net
463	6.33%	*.googletagmanager.com
443	6.05%	https://www.gstatic.com
437	5.97%	https://www.google.com
430	5.88%	*.doubleclick.net
384	5.25%	*.googleadservices.com

Figure 3: Most common whitelisted hosts in script-src.

Count	Percentage	Requested URL
1095	3.44%	https://www.google.com
740	2.33%	https://www.facebook.com
663	2.09%	https://fonts.gstatic.com
568	1.79%	https://googleads.g.doubleclick.net
557	1.75%	https://pagead2.googlesyndication.com
548	1.72%	https://adservice.google.com
461	1.45%	https://staticxx.facebook.com
450	1.42%	https://tpc.googlesyndication.com
412	1.30%	https://www.google-analytics.com
395	1.24%	https://www.gstatic.com
385	1.21%	https://vars.hotjar.com
319	1.00%	https://fonts.googleapis.com
311	0.98%	https://platform.twitter.com
291	0.92%	https://cm.g.doubleclick.net
291	0.92%	https://s.amazon-adsystem.com

Figure 4: Most common urls requested by scripts.

5 DISCUSSION

Our results shows that a vast majority of sites have objectively bad content-security policies. We explore what it means to have a bad CSP and potential causes. We then look into the role that advertisements play in CSPs, and how the current state of advertisement delivery prevents many sites from using safe policies. Although we don't make a formal recommendation, we explore how small changes in the way ad delivery networks work could allow stronger CSPs.

5.1 Most Sites Have Unsafe CSP

Before discussing why many sites have unsafe policies, we will first define what an unsafe policy could look like, and the factors that make it unsafe.

5.1.1 What is "Unsafe" CSP?. As mentioned in section 1, CSP is designed to help mitigate XSS and other browser hijacking types of attacks [14]. Content-security policies which do not prevent an attacker from exploiting an XSS vulnerability are therefore considered unsafe. Accordingly, CSPs which allow 'unsafe-inline' and 'unsafe-eval' are deemed unsafe, because they allow an attacker with XSS access to either inject an arbitrary script tag onto the page, or to use eval() to accomplish the same effect.

To quote security blogger Troy Hunt, "One of the first things you'll see there is that I can solve [the problem of using inline scripts with CSP by] using the "unsafe-inline" keyword. This completely disables the very defence we're talking about here and by doing this, any script can run. I show people this and they frequently respond with "Whoa - isn't that dangerous?!""[8]. We agree that yes, it is dangerous.

Next we evaluate the security promises of 'strict-dynamic'. Though including 'strict-dynamic' does not inherently indicate a website has a bad CSP, it does open the door to potential risk. The danger in using 'strict-dynamic' originates from the second level, or indirectly loaded resources which the trusted root script loads. Even though a site may vet the root script via a nonce or a hash, that root script can then inject any other script onto the page. For example, a site may trust ads.com to properly serve ads, but with 'strict-dynamic' turned on trust is propagated to *every script* ads.com puts onto the response page, (and in turn any script each of the indirectly loaded scripts loads). We argue that trust propagation expands a site's *trusted resource base (TRB)* in a similar fashion to 'unsafe-inline' and therefore decreases the overall security of that site by allowing a vetted resource to *unintentionally* inject potentially malicious resources onto the webpage. Consider a site example.com which trusts ads.com and vets all scripts from ads.com with a nonce. This resource then tells a client accessing example.com to download ads.com/vulnerable.js. Since the trust example.com has put in ads.com has propagated to the new script, vulnerable.js now has full control of the webpage. Ostensibly Google has vetted this ad script, but there is no guarantee that they have done so and more importantly, there are no repercussions for failing

to do so. There are previous studies which show ad networks that actively open up sites to XSS attacks [7].

5.1.2 Trusted Resource Base. A trusted resource base is defined as the set of resources a website explicitly or implicitly trusts. CSP techniques such as `nonce`, hash, and white listing give explicit trust to specific resources or specific origins to serve resources. They do not increase the amount of implicit resources that you trust since there is no concept of *transitive trust* from these sources.

In contrast, `'strict-dynamic'` increases the amount of resources that a site implicitly trusts. For example, an explicitly trusted root source under `'strict-dynamic'` implicitly trusts any resource it loads. Using `'strict-dynamic'` expands a site's TRB.

Similar to the concept of a trusted computing base[15] in computer security, sites which have a smaller *trusted resource base* comprised of explicitly named resources are preferable to those with large *trusted resource bases* or with many implicit resources. We argue a smaller TRB is better than a large TRB because any resource in the TRB has the potential to provide malicious content, either intentionally or unintentionally. Additionally, we argue that explicit trust is better than implicit trust because explicit trust is given based on a relationship between a provider and a developer based on reputation and information, whereas a developer will have no relationship or knowledge about implicitly trusted resources.

CSPs utilizing `'unsafe-inline'` or `'unsafe-eval'` implicitly trust every resource. Resulting in the largest TRB possible under a CSP.

5.1.3 Why Developers Use "Unsafe" CSP. Content-security policies are not trivial to implement. The CSP 2.0 W3C recommendation said it best: "There is often a non-trivial amount of work required to apply CSP to an existing web application"[13]. Developers and site administrators must consider which resources they are using, which ones they need, and which type of CSP policy will allow them to use these resources while reducing their attack surface.

To make matters worse, even developers who are so inclined to utilize CSP are faced with challenges, especially if their sites utilize ads or analytics. A cursory search on `google.com` for the term "how to use CSP with adsense" [3] shows only unofficial results from StackOverflow and similar resources. A telling example [1] shows a curious developer who asks if "Content Security Policy is incompatible with Google Analytics and Google AdSense". The answer to this question states that "It's not [if you use 'unsafe-inline']". This is not scientific, but it shows how an everyday developer can be forced into poor CSP practices.

One of the age-old problems with computer security is that you rarely see the benefits of good security, and only see the problem with bad security after it's too late. Because of this, we believe, most of the Internet is complacent with regard to CSP because it makes their present lives easier. From testing, we find it impossible to load a resource from the most ubiquitous script source, Google Analytics[4], without at least enabling `'strict-dynamic'`, leading us believe that a majority of developers will not take the time to adequately research and develop a CSP.

[3]last searched on 12/10/2018.

5.2 Ads, Analytics, & CSP

Our discussion ends with some thoughts on ads and analytics, the two most commonly requested classes of scripts. We also believe that it is possible to restructure the way that ad and analytics providers deliver their content so that it would work better with strong content-security policies and at the same time lessen the size of the trusted resource base.

5.2.1 How Are Ads & Analytics Delivered Now? In order to understand how ads and analytics interact with CSP we used Google Adsense and Google Analytics to add advertisements and analysis tracking to a personal webpage. Our evaluation of both scripts shows that they use external JavaScript to dynamically inject ad or tracking JavaScript onto our page. With this model, advertisements and analytics tracking cannot be served to the end user with a CSP stricter than `'strict-dynamic'`; they require *transitive-trust* and cannot be used with just `nonces` or `hashes`.

5.2.2 How Could Ads & Analytics Be Delivered? As discussed in 5.2.1, ads and analytics necessitate the use of what we classify as "unsafe" CSP. While it is unrealistic to ask developers to remove all ads and tracking in favor of strong security policies, we propose a way in which these services could be delivered in conjunction with strong CSP. Here we discuss what would be an ideal delivery system for ads and analytics within the confines of a strong content-security policy and a small trusted resource base (TRB).

Instead of ads and analytics loading scripts which load other scripts, we propose that ads and analytics *be served dynamically directly from the requested resource.* This type of structure could be used with a `nonce`-based CSP and would also vastly reduce the site's TRB.

To illustrate our system, `example.com` wants to load an ad. It will request a script from `ads.com`. Instead of `ads.com` returning a script resource which subsequently loads the ad, we propose that `ads.com` *loads the ad dynamically before servicing it to* `example.com`. Now `example.com` can use a `nonce` to vet the ad script from Google without the need for `'strict-dynamic'`, removing to need to implicitly trust resources from additional unknown sources.

If `ads.com` were a bad actor, then the TRB of `example.com` would not be reduced. For instance, it is still possible for `ads.com` to inject a malicious script onto `example.com`. However, we argue that since `ads.com` is now dynamically creating the ad *it removes the possibility that they inadvertently sent* `example.com` *a malicious script*! `example.com` no longer needs to trust `ads.com` and all resources it could load. Instead, `example.com` trusts only `ads.com` and places the responsibility on `ads.com` to properly vet all scripts it serves. In this setup the provider would need to either be complicit in serving malware or inept at verifying their content. We claim that this system is strictly better than the present setup as it reduces the TRB and the extent to which *transitive trust* is necessary.

5.2.3 Final Thoughts. CSP is a powerful prophylactic which is extremely underutilized. We discuss why we believe that CSPs are not used in practice, specifically because they can be hard to implement correctly and because developers are not incentivized to prioritize security. We also posit that the primary cause of "unsafe" CSPs is the prevalence of advertisements and analytics, which make it difficult, if not impossible, to build secure policies.

Content-security policies provide a great secondary protection layer against XSS attacks, but only if they're implemented properly. If ads and analytics took on the onus of vetting the scripts they utilize, we believe that the world of the Internet could be a much more secure one.

ACKNOWLEDGMENTS

We would like to thank Deian Stefan and Michael Smith for their guidance throughout this project and Cindy Moore for helping us to procure the computation resources to run our crawler.

REFERENCES

[1] 2015. content-security-policy vs. Google Analytics and AdSense. https://github.com/h5bp/html5-boilerplate/issues/1765. (2015). Accessed: 2018-12-10.
[2] Amazon. 2018. Alexa Top Sites. https://aws.amazon.com/alexa-top-sites. (2018). Accessed: 2018-10-23.
[3] Hristo Bojinov, Elie Bursztein, and Dan Boneh. 2009. XCS: Cross channel scripting and its impact on web applications. 420–431. https://doi.org/10.1145/1653662.1653713
[4] Google. 2018. Adding analytics.js to Your Site. https://developers.google.com/analytics/devguides/collection/analyticsjs/. (2018). Accessed: 2018-12-9.
[5] Scott Helme. 2018. Alexa Top 1 Million Analysis. https://scotthelme.co.uk/alexa-top-1-million-analysis-aug-2017/. (2018). Accessed: 2018-12-08.
[6] Raymond Hill. 2018. uBlock Origin. https://github.com/gorhill/uBlocks. (2018). Accessed: 2018-12-08.
[7] Troy Hunt. 2015. How I got XSS'd by my ad network. https://www.troyhunt.com/how-i-got-xssd-by-my-ad-network/. (2015). Accessed: 2018-12-9.
[8] Troy Hunt. 2017. Locking Down Your Website Scripts with CSP, Hashes, Nonces and Report URI. https://www.troyhunt.com/locking-down-your-website-scripts-with-csp-hashes-nonces-and-report-uri/. (2017). Accessed: 2018-12-8.
[9] MITRE. 2018. Common Vulnerabilities and Exposures. https://cve.mitre.org. (2018). Accessed: 2018-12-8.
[10] Mozilla. 2018. Content Security Policy (CSP). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. (2018). Accessed: 2018-12-8.
[11] Mozilla. 2018. CSP: script-src. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src. (2018). Accessed: 2018-12-8.
[12] OWASP. 2018. XSS Filter Evasion. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. (2018). Accessed: 2018-12-9.
[13] A. Barth D. Veditz and M. West. 2016. Content Security Policy Level 2. https://www.w3.org/TR/CSP2/. (2016). Accessed: 2018-12-8.
[14] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*. Vienna, Austria.
[15] Wikipedia contributors. 2018. Trusted computing base — Wikipedia, The Free Encyclopedia. (2018). https://en.wikipedia.org/w/index.php?title=Trusted_computing_base&oldid=826284008 [Online; accessed 10-December-2018].