Assessing the Impact of Script Gadgets on CSP at Scale

Sebastian Roth, Michael Backes, and Ben Stock
CISPA Helmholtz Center for Information Security
{sebastian.roth,backes,stock}@cispa.saarland

ABSTRACT

The Web, as one of the core technologies of modern society, has profoundly changed the way we interact with people and data. One of the worst attacks on the Web is Cross-Site Scripting (XSS), in which an attacker is able to inject their malicious JavaScript code into a Web application, giving this code full access to the victimized site. To mitigate the impact of markup injection flaws that cause XSS, support for the Content Security Policy (CSP) is nowadays shipped in all browsers. Deploying such a policy enables a Web developer to whitelist from where script code can be loaded, essentially constraining the capabilities of the attacker to only be able to execute injected code from the said whitelist.

As recently shown by Lekies et al., injecting *script* markup is not a necessary prerequisite for a successful attack in the presence of so-called *script gadgets*. These small snippets of benign JavaScript code transform non-script markup contained in a page into executable JavaScript, opening the door for bypasses of a deployed CSP. Especially in combination with CSP's logic in handling redirected resources, script gadgets enable attackers to bypass an otherwise secure policy. In this paper, we, therefore, ask the question: is securely deploying CSP even possible without a priori knowledge of *all* files hosted on even a partially trusted origin? To answer this question, we investigate the severity of the findings of Lekies et al., showing real-world Web sites on which, even in the presence of CSP and without code containing such gadgets being added by the developer, an attacker can *sideload* libraries with known script gadgets, as long as the hosting site is whitelisted in the CSP. In combination with CSPs matching logic for redirects, this enables us to bypass 10% of otherwise secure policies in the wild. To further answer our main research question, we conduct a hypothetical what-if analysis. Doing so, we automatically generate sensible CSPs for all of the Top 10,000 sites and show that around one-third of all sites would still be susceptible to a bypass through script gadget sideloading due to heavy reliance on third parties that also host such libraries.

ACM Reference Format:
Sebastian Roth, Michael Backes, and Ben Stock. 2020. Assessing the Impact of Script Gadgets on CSP at Scale. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), June 1–5, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3320269.3372201

ASIA CCS '20, June 1–5, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/06…$15.00
https://doi.org/10.1145/3320269.3372201

1 INTRODUCTION

In today's society, the Web is an essential part of everyday life. It has grown not only into a platform that enables us to keep in touch with friends via social media, but more importantly, it has transformed into a full-fledged application ecosystem, hosting even complex applications. Given this ever-increasing importance, any threats specific to the Web are endangering the security of delicate data. One of the gravest threats are so-called Cross-Site Scripting (XSS) vulnerabilities. These vulnerabilities allow an attacker to execute JS code within the context of a flawed Web site, essentially enabling the attacker's code to conduct any action the site's own JS could. This enables an attacker to steal a victim's credentials, leak sensitive information, or perform actions on behalf of a victim. Thus, XSS can cause severe damage, especially if present in security-critical applications. To mitigate the impact of this vulnerability, a developer can make use of the Content Security Policy (CSP). Since XSS means that code not intended by the developer is executed within their application, CSP follows a whitelisting approach of the developer's intended code. In particular, a developer can provide a whitelist of *scripting resources* that are allowed to be loaded, and thus executed, in the Web application's context.

Notably, while several papers have shown the inability of site operators to deploy CSP in a secure fashion [3, 4, 27, 28], Lekies et al. [14] highlighted a new threat: script gadgets. They dub script gadgets such pieces of JS code, which turn non-script data into executing code. Hence, these snippets, often contained in widely used libraries like AngularJS, enable an attacker to exploit injection flaws in an application *without* the necessity to inject actual script payloads. As non-script data is not governed by CSP, this enables an adversary to successfully exploit an injection vulnerability in the presence of a script gadget. In order to leverage a script gadget, the containing library either needs to be already loaded into the site, or a host containing such needs to be whitelisted. Prior work from Weichselbaum et al. [27] has indicated that 56% distinct policies discovered in the wild could potentially be bypassed through gadget-hosting sites in whitelists. Notably though, their paper reported a vulnerable CSP configuration in case any version of AngularJS could be found on a whitelisted host. As we discuss in Section 5.2, however, the mere presence of a host that contains a library flagged as AngularJS is not sufficient for a successful exploit.

Hence, we extend both the works from Lekies et al. [14] and Weichselbaum et al. [27] by showing real-world exploitability of strict CSPs discovered in the wild. In particular, given a CSP which restricts scripts to a fixed number of hosts, we determine if any known vulnerable version of AngularJS is hosted on these sites.

As CSP is meant as a last line of defense against XSS injections, we assume that a site deploying CSP could be susceptible to XSS. We hence simulate an injection vulnerability and *sideload* the gadget library into the target application, combining it with a non-script payload. Only if the exploit can then be triggered, we mark a site

as being susceptible to a bypass through script gadget sideloading. In doing so, we find that for the mere 248 sites that make use of a sane CSP, while 29 contains whitelist entries pointing to AngularJS-hosting sites, only 24 (9.6%) are bypassable through script gadget sideloading. Importantly, while the set of sites that deploy CSP in a meaningfully secure way are small, to begin with, the bypassability of their policy through script gadgets highlights the heavy burden that the reliance on third parties puts on building a policy.

Prior work has proposed to enable deployment of CSP through automated tools (e.g., [18]) based on the scripts that are required for the site's functionality. To understand how badly script gadgets impair such tools' abilities to generate a functional, yet secure CSP, we study the issue of script gadgets from a second angle: a hypothetical what-if analysis of the top 10,000 sites. To that end, based on the JS sources used by these sites, we generate sane CSPs and show that more than one-third of these applications could be successfully attacked through a sideloaded script gadget, merely due to their reliance on third-party hosts for vital functionality. To make matters worse, to avoid leaking path information across origins, the matching algorithm of CSP ignores the path component of a source expression if the resource loaded is as the result of a redirect [8]. Therefore even if the developer did not whitelist the entire source domain, but only specific scripts, we can still sideload a library of our choice if at least one of the whitelisted sources suffers from an open redirect. Notably, 114 different domains that are part of the top 10,000 frequently used Web sites or are resources that are loaded by these Web sites suffer from this vulnerability.

Overall, our findings indicate that on top of the massive engineering effort necessary to deploy a meaningful CSP, site operators are faced with even more pitfalls due to their heavy reliance on third parties. This, in combination with CSP's logic around redirects, makes mitigating XSS through CSP even harder than already pointed out by previous work [3, 4, 27, 28]. In summary, our work makes the following contributions:

- We examine the prevalence of libraries that contain known script gadgets at a large scale, and outline how open redirects can be used to further widen the attack surface of script gadget based CSP bypasses (Section 4).
- Based on the real-world deployment of CSP, our discovered gadgets, and redirects, we show that 10% of otherwise secure real-world CSPs can be bypassed through script gadget sideloading (Section 5).
- To further document troubles in developing a sane CSP, we furthermore conduct a hypothetical analysis, showing that script gadgets would likely undermine the security of around one-third of sites if the deployed host-based CSPs (Section 6).
- Based on the insights gathered throughout our analysis, we discuss the root cause issue behind the outlined exploitation, better CSP creation strategies, and call on parties capable of addressing this problem at scale (Section 7).

2 TECHNICAL BACKGROUND

This section describes the various technologies used in this work. In particular, we outline Cross-Site Scripting, CSP as a mitigation against the attacks, as well as script gadgets as presented by Lekies et al., and the concept of open redirects.

2.1 Cross-Site Scripting

Including content from third-party pages is commonplace on the Web: from advertisements to map services, all sorts of content are loaded via frames into applications. If there were no separation mechanism, this inclusion of content from different sources would have severe security consequences. Therefore, the Same-Origin Policy (or *SOP* for short) is the most basics security mechanism on the Web, ensuring that only documents from the same Web Origin [1] can access each other. This means that any JavaScript running inside a given document can only access other documents' content if their protocols, hostnames, and ports match. Therefore, to gain access to another document's content, the code of an attacker must be running the same origin; e.g., through a code injection vulnerability in the targeted application. This attack is called Cross-Site Scripting (or *XSS*), as the attacker can inject code into another site. Thus, the malicious code can do whatever legitimate code can do, such as modifying the page to the attacker's liking, exfiltrating sensitive information such as session cookies, or perform any action in the name of the victimized user.

2.2 Content Security Policy

As previously outlined, XSS can cause massive damage to a Web application. To mitigate the impact of such unintended JavaScript code execution, the Content Security Policy (or CSP for short) was introduced by Stamm et al. [22]. Such a policy can be deployed via HTTP headers or meta elements, consisting of multiple directives separated by a semicolon. A list of source expressions follows each directive name. These expressions represent the sources from which resources of the type defined by the directive name may be included. For example, to allow only Google Analytics and self-hosted (i.e., on the same origin) code as allowed script sources and restrict any other resource (e.g., objects, frames, media) to load, the following policy can be used:

```
default-src 'none';
script-src 'self' www.google-analytics.com
```

Whenever either `script-src` or `default-src` (as fallback) is specified, CSP also prohibits the use of inline scripts, event handlers, and functions that perform a string-to-code transformation. However, these restrictions can be relaxed by adding `'unsafe-inline'` or `'unsafe-eval'` to the directive. In its original candidate recommendation from 2012 [24], CSP only supported whitelisting of host names and URLs. Later on, in CSP Level 2 [25] this inflexibility of Level 1 is addressed, especially concerning inline script and event handlers. To make whitelisting of them easier, the standard added support for hashes and nonces to whitelist scripts. By using hashes, the developer can explicitly whitelist inline scripts by adding an SHA hash of the script code to the `script-src` directive. Alternatively, when a nonce is present, all scripts (both inline and external) which carry that nonce as an attribute are whitelisted.

Using a whitelist exclusively containing hashes makes adding additional script resources by those whitelisted scripts impossible. For nonces, a script could theoretically read its own nonce, and when adding new scripts, attach said nonce to them. Notably, though, the current W3C Working Draft (CSP Level 3 [26]) added a feature to address this issue in particular: `'strict-dynamic'`. When this

```
let buttons = document.querySelectorAll(
↪  "[data-role=button]");
for (let b in buttons) {
  buttons[b].innerHTML = buttons[b].getAttribute(
  ↪  "data-text");
```

Figure 1: Example for a script gadget.

```
<button data-role='button' data-text='<img src=foo
↪  onerror=alert(1)>'>
```

Figure 2: Attack payload for Figure 1.

expression is deployed, any script whitelisted through nonces or hashes can *programmatically* (i.e., using createElement and appendChild, not `document.write`) add additional scripts. This enables whitelisted scripts to propagate the trust put into them. Moreover, `'strict-dynamic'` disables any host-based whitelist. As it can only ever be deployed with hashes or nonces, having `'strict-dynamic'` also means `'unsafe-inline'` has no effect, as this is ignored in the presence of hashes or nonces. Notably, though, support for `'strict-dynamic'` is limited and not currently supported by Safari or Microsoft Edge [16].

After a blog post from Homakov [8], which showed that CSP could be leverage to leak sensitive path information when a resource is redirected, CSP Level 2 adopted a relaxed path matching scheme. In particular, assume the following CSP:

```
script-src https://redir.com https://cdn.com/benign.js
```

In this case, the policy whitelists only a single script from cdn.com. However, assuming that redirect.com contains a URL which, instead of delivering actual content just sends a 30x redirect to the browser (see also Section 2.4), CSP's matching algorithm will ignore the paths component of any entry in the whitelist, and allow scripts to included from any *origin* for which at least one resource is whitelisted. In this example, this would enable an adversary to include a script from https://redir.com?target=https://cdn.com/vulnerable.js. Therefore, in case a whitelisted element contains a redirect where the target is under the control of an attacker, CSP's restriction capabilities are partially crippled.

2.3 Script Gadgets

Lekies et al. [14] discovered that it is not necessary to directly inject malicious markup with JS code into a Web site to perform an XSS attack. Instead, fragments of legitimate JS code, so-called *script gadgets*, can be used to inject or execute malicious payloads. Figure 1 shows an example of such a script gadget. The code iterates over all buttons (lines 1 and 2), extracting the attribute `data-text`, and setting the corresponding button's innerHTML property. Therefore, instead of injecting a script tag, the attacker can simply add a button shown in Figure 2. If injected into the page itself, this would not be executed as JavaScript since the `data-text` attribute has no special meaning, which would require evaluation as HTML markup. However, when the script gadget accesses the property and sets the innerHTML property, the attacker's code is executed.

```
<!-- injected by an attacker -->
<div id="someelement">
<script src="//attacker.org/attack.js"></script>
</div>
<!-- existing code -->
<div id=otherelement></div>
<script nonce=random>
$("#otherelement").html($("#somelement").html());
</script>
```

Figure 3: jQuery example for a script gadget.

Important for our work is a script gadget's ability to bypass existing CSP restrictions. In the simplified example, the attacker's code (after written by the gadget) is contained in an event handler, which would not be executed if CSP is deployed without `'unsafe-inline'`. Moreover, writing a script tag does not help either, since an assignment of innerHTML will not execute any script tags according to the HTML specification [29].

To understand the impact of such gadgets on CSP, we look at a slightly more involved example using jQuery, as shown in Figure 3. We assume that scripts are whitelisted through nonces, and `'strict-dynamic'` is used. Notably, the html function in jQuery is actually more than a wrapper around innerHTML. In particular, if script elements are detected within the HTML being passed to the function, those script elements will either be passed to `eval` (in case of inline scripts) or result in a programmatic addition to the document (for external scripts). Hence, while CSP would stop the browser from directly loading the attacker's injected script, the script gadget would parse the script, and add it to the DOM through createElement and appendChild. Since this script is nonced and `'strict-dynamic'` is used, the browser would now load and execute the attacker's code.

2.4 Open Redirects

An HTTP redirect is an automatic redirection of one URL to another, usually indicated by the 3xx HTTP status code [2]. This redirection of URLs can, for example, be used to temporarily redirect requests to another server if the original server is under maintenance. In order to create a more dynamic way of redirecting to other pages, the target URL may be specified by for example HTTP parameters as depicted below.

```
$redirect_url = $_GET['redir'];
header("Location: ".$redirect_url);
```

However, if the target of the redirect is not validated properly on the server-side, an attacker can use this to cause a redirect to an arbitrary target. Generally speaking, this can be used to trick users into hiding the actual source of content. As an example, open redirects may be used by attackers in phishing campaigns, as users may only check the URL before clicking it (pointing to a seemingly benign site), but not after a redirect has occurred (to the attacker's page). For our use case, such an *open redirect* becomes especially problematic in combination with CSP, as CSP's matching algorithm for whitelisted sources ignores any paths when a resource is loaded as the result of a redirect.

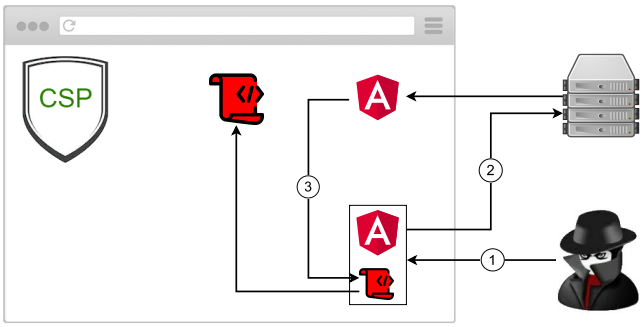

Figure 4: Attacker Model for CSP bypass

3 ATTACKER MODEL & RESEARCH QUESTIONS

In this section, we explain our threat model and its preconditions in detail. Furthermore, we present our main research question and the intermediary goals of our work.

3.1 Threat Model

In this work, we investigate to what extent CSPs can be bypassed by sideloading libraries with script gadgets from whitelisted parties.

Therefore, we assume a Web site is using a CSP that is not trivially bypassable in order to mitigate the impact of markup injections. In this CSP, the `script-src` directive has whitelisted all necessary JavaScript sources for the Web site, including a script-rich third-party domain (e.g., a CDN) that also hosts a library containing a script gadget. We do not assume that the site itself necessarily makes use of this library. However, we do assume that this site suffers from a markup injection vulnerability, allowing an attacker to insert arbitrary markup into the application. The actual attack for bypassing the CSP is divided into the following steps, which are also pictured in Figure 4:

(1) The attacker utilizes the markup injection to add the attacker payload to the Web site. This payload consists of two parts: First, a script tag that points to a library containing a script gadget hosted by a whitelisted party (e.g., AngularJS). Second, a piece of markup which itself will not be executed (e.g., a button as discussed before).
(2) The injected script tag loads the gadget-containing library into the Web site. The CSP does not intervene, because the third party that hosts the script is whitelisted as a trusted script source, meaning the library is added to the site's execution context.
(3) Now that the script gadget is present on the Web site, the second part of the injected payload triggers this gadget to execute the attacker's malicious JavaScript code.

In addition to the straight-forward loading of such a gadget library from a whitelisted host, we also consider the case of open redirects. As explained before, when a redirect occurs, CSP's matching rules do not consider the path of a resource anymore. Hence, assuming that a site requires some resources from a site which *also* hosts libraries with gadgets, but whitelists those scripts explicitly by their full URL, a single whitelisted host with an open redirect suffices for a bypass. In that case, the attacker injects a script tag pointing to the open redirect site, pointing the redirect target to the gadget library. CSP then checks to see if the host of the said library is contained in any whitelist entry, and loads the script.

Thus, if any of the whitelisted sources suffers from an open redirect vulnerability, we can sideload the gadget via this redirect, although only another script from the gadget source is whitelisted.

3.2 Research Question and Goals

The main research question of this work focus is: *How badly do script gadgets impair a site operator's ability to deploy a secure CSP?* This question can be divided into the effectiveness of bypasses in real-world applications as well as the theoretical exploitability if all Web sites would deploy a CSP that restricts script sources. To answer this question, we need to reach a number of intermediary goals. For actually bypassing a policy using a script gadget, it is necessary to know the sources that host the corresponding libraries, such that we can sideload them into the targeted Web site. By collecting these sources, we can determine how many different sites host libraries with script gadgets. To further increase the effectiveness of our attack, we need to learn open redirect URLs. With those vulnerabilities, we can utilize the redirect-based path relaxation attack described in Section 2.2 to still sideload a script gadget, although the source is only whitelisted as a URL to other scripts on the same domain. The collection of the open redirects in real-world applications allows us to determine to what extent these can be used for additional bypasses. Since the bypassability of most of the real-world CSPs was shown by several publications [3, 23, 27], we first focus our investigations on those CSPs that are not trivially bypassable. This allows us to investigate how script gadgets and open redirects can still undermine many of the handful of secure policies on the Web. Therefore we first need to define how a policy aimed at restricting script content has to be designed such that it works effectively. By collecting the CSPs that are present in modern Web applications and analyzing them according to our effectiveness definition, we can see how many Web sites are using CSP for effectively mitigating XSS attacks. Finally, given that the number of sites that make use of a sensible CSP is minuscule, we also conduct a hypothetical analysis based on CSPs generated from the used script sources of the top 10,000 Web sites.

4 BYPASS PREPARATIONS

The first step towards analyzing how susceptible sites are to sideloading script gadgets is to gather two important pieces of information. This entails the detection of URLs on which known script gadget libraries are hosted. To that end, using an automated crawler, we investigate the scripts regularly used by the top 10,000 Web sites and classify each script to determine if it is known to contain script gadgets. In addition to this, given the decision of the CSP standard's authors to relax host matching in case of redirects, we also need to create a list of URLs that have open redirects. As part of our crawling setup, we, therefore, record all network requests, particularly focussing on resources that redirect to yet another URL. Subsequently, we apply two matching algorithms to determine if the redirect target was contained in the original request's URL and, if so, mark that URL as an open redirect.

4.1 Dataset Curation

Our first goals are to assess how many sites are hosting libraries that carry script gadgets and determine which sites allow for open redirects. To address these, we crawl data from the main page as well as all same-site subpages of the Tranco [12] Top 10,000 Web sites list, created on May 10, 2019, with a maximum of 1,000 distinct URLs per site. To visit these pages, we utilize Google's browser instrumentation framework `puppeteer` [6] that instruments a Chromium Web browser. During each visit of a page, the crawler intercepts every HTTP response regardless of the content type. If such a response is the result of a redirect, we check whether the URL that triggered the redirection has its target mentioned in one of the parameters. Here we considered not only the full URL but also domain, path, or their base64 encoded equivalents being present. As soon as one of those occur in the URL, we replace it with the URL of a script hosted by GoogleAPIs[1]. Then we request the URL with the new target and check if the response matches this script. If so, we consider this redirect to be open and thus attacker-controllable.

In this procedure, we also capture whole chains of redirects, because each and every redirect results in an additional request being issued. Additionally, if the content type is related to JavaScript and the actual source code of the requested script is present in the response, we use `Retire.js` [17] to investigate whether this JavaScript code contains known frameworks or libraries. We decided to use `Retire.js`, because in contrast to other applications to analyze JS libraries (such as Wappalyzer), it is free, can do detection locally without relying on (possibly rate-limited) APIs, and is easy to embed in our crawling infrastructure. Using this information, we create a mapping between hosts and the publicly available libraries or frameworks, which contain script gadgets provided by them as well as domains that use those libraries in their execution context. To use the collected library sources for our CSP bypass, we intersect the libraries with the script gadget containing libraries shown by Lekies et al. [14], and created exploits based on their PoCs [2].

Thus, we are able to see how common the usage and the hosting of these libraries are in the wild. Later on, we can reuse this dataset in our bypass generation to sideload a script gadget containing a library from a whitelisted source.

With the collected redirects, we then investigate which of the query parameters in the redirection URL determines the target of the redirect. Therefore, we search for the target URL or its base64 encoded equivalent in the query parameters of the URL that triggered the redirect. In case of a match, we then change the corresponding part of this URL to a script source of our choice. To validate whether the redirect we found is indeed an open redirect, we load each of the redirection URLs with the new target of our choice. In case of successful loading the new target via this link, we store the redirection URL as an open redirect.

4.2 Script Gadget Prevalence

Using our dataset, we are able to detect 28 different libraries loaded from 9,909 different sites. To investigate the prevalence of script gadgets, we intersect these libraries with the libraries known to contain script gadget as identified by Lekies et al. [14]. Furthermore, we investigate how many hosts contain these libraries on how many distinct URLs. Notably, as soon as only a query parameter has changed, we count this as a new URL. The results of this investigation are shown in Table 1. The number of different hosts is much higher than the size of the actually crawled dataset, given that inclusions often occur from sites outside the Top 10,000.

Table 1: Distribution of script gadget libraries.

Library	URLs	Hosting Sites
jquery	54,659	9,242
bootstrap	8,259	2,851
angularjs	2,346	947
dojo	116	82
backbone.js	318	161
vue	51	27
ember	8	7

We can attribute this to the fact that we not only consider the crawled Web site itself, but also all third parties that are used by those Web sites. The jQuery library is by far the most frequently used library in the wild that contains a script gadget in specific versions. However, the gadget from jQuery only enables an attacker to bypass `strict-dynamic` in a CSP, and also to bypass several XSS filters, but not the bypass of a host-based CSP whitelist. Additionally, the script gadget in jQuery cannot be triggered without an explicit call to a jQuery function with attacker-controlled input. The second most frequently used gadget library, Bootstrap, also only contains gadgets to bypass `'strict-dynamic'` and XSS filters. However, the third entry, AngularJS, is hosted on 947 different sites, and its script gadget enables us to bypass CSP host-based whitelists and execute our payloads without additional prerequisites. Other script gadget libraries that are capable of bypassing host-based whitelists are AureliaJS and PolymerJS. However, we did not find any occurrence of them in our dataset. The absence of these libraries is either because we only crawled the first-level subpages or because, in some cases, Retire.js does not correctly classify libraries because the Web developer customized their version.

In addition to the libraries we discovered this way, we augmented our dataset with a list [3] of publicly known sources for AngularJS. We checked this list to ensure all the listed URLs still host the libraries in question, making sure they are viable targets for exploitability. Notably, this adds `gstatic.com` to the list of viable hosts, which proved to be one of the most successful bypass enablers. This was not discovered in our crawl as it is seemingly not used by any site (according to a search on PublicWWW[4]).

4.3 Open Redirects

We found 4,902 URLs, from 114 distinct domains that can be used as an open redirect. As depicted in Table 2 on some domains nearly 500 distinct URLs can be used to perform open redirects. Note that these domains are not necessarily part of the top 10,000 Tranco list

[1] https://ajax.googleapis.com/ajax/libs/angularjs/1.5.6/angular.min.js
[2] https://github.com/google/security-research-pocs/blob/master/script-gadgets/bypasses.md
[3] https://github.com/google/csp-evaluator/blob/master/whitelist_bypasses/angular.js
[4] https://publicwww.com/websites/%22gstatic.com%2Ffsn%2Fangular_js-bundle1.js%22/

Table 2: Top 8 hostnames with the highest number of URL that allows for open redirects.

#	Hostname	URLs
1	m.adnxs.com	491
2	sync.mathtag.com	390
3	ssum-sec.casalemedia.com	289
4	ml314.com	242
5	sync-tm.everesttech.net	236
6	pixel.tapad.com	233
7	pm.w55c.net	221
8	image6.pubmatic.com	200

Table 3: Top 8 URL query keys with the highest number of distinct domains that use them for targeting.

#	key	Domains	URLs
1	redir	22	1612
2	url	14	203
3	r	12	230
4	redirect	8	172
5	cburl	8	43
6	*no key*	6	314
7	rurl	6	308
8	cb	3	304

but rather domains that were used by those sites to load resources. Most of the top domains with the highest number of open redirect URLs are ad or analytic providers. This fact is especially important for our generation of CSP bypasses because ad and analytics scripts are whitelisted frequently in real-wprld CSPs. We also investigated which query parameter is how frequently used to define a target of a redirection. Therefore we analyzed each of the validated open redirect URLs to extract the query parameter with the placeholder as value. As depicted in Table 3 the most prominent parameter used for redirects is `redir` (1,612 distinct URLs). Also, many of the URLs (314) use the target directly as a parameter instead of specifying a key for defining the target value. With this list of frequently used query parameters, one can use customized search queries, like Google Dorks, to find open redirect vulnerabilities without the necessity of crawling thousands of Web sites.

5 REAL-WORLD IMPACT

After having collected real-world data on both URLs pointing to known script gadget libraries as well as a list of open redirects, we now turn to our main research question, namely the severity of the impact of script gadget sideloading on CSPs. To achieve this, we first collect the CSPs sent by the Web applications in our top 10,000 list. Based on the notion of a meaningfully secure policy for script content restriction, we then evaluate how many sites could potentially fall victim to sideloaded script gadgets due to the presence of whitelisted hosts with known script gadgets. Contrary to Weichselbaum et al. [27], we do not simply assume that this

```
<div ng-app ng-csp>
  <div ng-click="x=$event" id=f tabindex=0>
    <h1 id="trigger">Click me!</h1>
  </div>
  <div ng-repeat="(key, value) in x.view">
    <div ng-if=key=="window">
    {{ value.alert = [1].reduce(value.alert, 1) }}
    </div>
  </div>
</div>
```

Figure 5: Example injection to trigger alert via the script gadget present in the AngularJS library.

matching is sufficient, and instead build a method to confirm exploitability in practice by simulating an injection flaw. This allows us to highlight a shortcoming of this prior work, namely that not all libraries detected as AngularJS are, in fact, scripts that can be leveraged for a gadget-based attack.

5.1 Methodology

To generate exploits that can bypass a CSP, we need information about the CSP that is used by the target, an HTML markup injection to execute our payload, a URL for a library containing a script gadget, as well as a payload which can trigger the script gadget. In some cases, we may also use one of the discovered open redirect vulnerabilities to sideload a script gadget library. During the crawling procedure explained in Section 4 we also collected all CSP HTTP headers as well as all policies that are deployed as HTML meta tags on the target sites. We parse every collected policy according to the parsing instructions of the CSP Level 3 standard [26]. Given our attacker model, which aims to bypass a policy that restricts script content by sideloading a gadget, we only consider Web sites that have a meaningfully secure CSP for content restriction. In particular, we consider that a site has such a policy iff:

(1) It uses the `script-src` directive or the `default-src` directive as a fallback
(2) It does not use the `*` as a wildcard, which would whitelist any possible script source.
(3) It does not whitelist entire schemes like `data:` or `https:`.
(4) It does not contain `'unsafe-inline'`, which would allow the use of inline JavaScript.

As an actual bypass of a host-based CSP script whitelist, we use a script gadget found by Lekies et al. [14], which is present in the AngularJS library. Figure 5 shows an example markup that uses the script gadget present in AngularJS to call the JavaScript `window.alert` function, without injecting any script tag or event handler. The script gadget in AngularJS is one of the most severe ones that allow for arbitrary code execution, and AngularJS has the best distribution according to our data from Section 4. Furthermore, in comparison to other gadgets, it has no preconditions except for the presence of the AngularJS library. The restriction to only use AngularJS as a script gadget should have no severe impact on our bypassability results. According to our data from Section 4, most of the library providers are CDNs. If our attacker has the capability to

sideload a script from one of these CDNs, it makes no difference which of the available script gadget libraries is used for the bypass.

We used the dataset from Section 4 to create a list of sources that hosts the AngularJS library. For each Web site with a secure CSP according to our definition, we generate an exploit that simulates a markup injection to inject a script tag that loads AngularJS from one of the angular sources that are whitelisted in the respective CSP. If a Web site whitelists only one specific script from a CDN, our exploit generator uses an open redirect vulnerability found by the procedure explained in Section 4 in one of the whitelisted sources such that the exploit is still capable of sideloading the AngularJS library from the CDN. Now that the vulnerable library is present on the Web site, we abuse the script gadget to execute our own JavaScript code. Our crawler simulates a markup injection to validate whether the bypass of the policy works in the wild. For simulating a markup injection, we utilize puppeteer's `page.evaluate` function to add our markup to the Web site. Importantly, this markup in the absence of the AngularJS library does not result in code execution.

5.2 Results

In total we were able to successfully access 998,712 URLs. From this dataset we collected CSPs from 2,076 different Web sites. Only 965 of those domains actually used the `script-src` or `default-src` directive. Out of these, 248 have securely done this with respect to our definition from Section 5.1.For the 248 Web sites that use a secure policy, we generated exploits for 29 of those sites, of which 24 could be successfully bypassed using our attack. While this number is very low when compared to initially crawled dataset, we stress that the bypass ratio of around 10% is for sites with a tight CSP in the first place. Hence, while the outlined attack does not impact a large body of sites, it does underline that even high-profile sites can be prone to bypasses through script gadgets and open redirects, indicating the dangers of not having full knowledge about code hosted on third-party sites.

To understand why some of the generated exploits could not be validated, we manually investigated these cases. We found that these could be attributed to misclassifications by retire.js, which incorrectly detected AngularJS when only a module of Angular (such as `angular-sanitize`) was present. Most of the Angular sources used for the exploits are only successfully used as a bypass for one Web site. This is because those are CDNs that are not publicly used CDNs, but rather CDNs that are used by specific Web sites such as `alicdn.com` for `aliexpress.com`. However, three of the Angular sources are used for more than one of the bypasses, as depicted in Table 4. Notably, some of the sites were vulnerable to multiple bypasses because they whitelisted multiple of the angular sources. Thus, the numbers shown in this table are intersecting. The AngularJS source `www.gstatic.com` enables us to bypass the CSP of 20 distinct domains. In our dataset collected for Section 4, we found evidence that at least four of the exploited websites used JavaScript that originates from `www.gstatic.com`. In all cases, it seems that they are only using gstatic to load the reCaptcha API or the Chrome Cast Application Framework. Thus they would have been able only to whitelist those specific URLs, such that an attacker would need an open redirect vulnerability in another whitelisted source to actually bypass their CSP. The bypasses that we found in the wild were all possible due to direct sideloading the script gadgets. Only two of these nevertheless exploitable bypasses would also be possible via open redirect-based sideloading (as this disables CSP's path matching). Thus, those would have been vulnerable even if, e.g., reCaptcha were to be whitelisted with the entire URL.

Table 4: Top 3 AngularJS sources used in real-world exploits.

Angular Source Domain	Affected Domains
gstatic.com	17
cloudflare.com	5
googlesyndication.com	4

5.3 Selected Case Studies

To get a better understanding of how those bypasses can happen, we take a closer look at two examples that deploy a seemingly secure CSP but are still exploitable when script gadgets are sideloaded. At the time of writing, we have notified both parties about the issue with their script-src directive.

5.3.1 Snapchat. First, we focus on `snapchat.com`, which deployed a CSP with the `script-src` directive depicted in Figure 6. At the first look, this script restricting directive appears to be a strict and secure policy. The policy itself does not contain any dangerous expressions like `'unsafe-inline'`. Moreover, only resources from the site itself and a single other party, namely Google, are trusted as third-party scripts. However, one of the whitelisted sources is `www.gstatic.com`, which enables our attacker to sideload https://www.gstatic.com/fsn/angular_js-bundle1.js and subsequently abuse the script gadget from AngularJS to execute arbitrary malicious payloads. Weissbacher et al. [28] have shown that creating a secure CSP requires a massive effort. While this policy may suffice to protect against regular types of XSS (importantly, the whitelisted hosts do not contain JSONP endpoints which could be used to execute code [27]), Snapchat's trust in `www.gstatic.com` makes them susceptible to our outlined attack; effectively rendering the mitigation through CSP ineffective against an XSS attacker. During our data collection procedure, we have not found any evidence that `gstatic.com` is actually used by Snapchat. This might originate from the fact that we automatically crawled the Web site without our crawler being logged into the Web application. Thus, only a fraction of the features which are available for real-world users were actually visited by us.

5.3.2 Spotify. The second example we consider is `spotify.com` (see Figure 7). Here, we again find that `www.gstatic.com` is whitelisted, opening it up to the previously outlined attack. Assuming

```
script-src 'self'
  https://www.google.com/
  https://www.gstatic.com/
  https://apis.google.com/
  https://www.google-analytics.com
```

Figure 6: CSP `script-src` of `snapchat.com`.

```
script-src 'self'
  'unsafe-eval' 'sha256-HASH'
  www.google.com
  www.gstatic.com
  sb.scorecardresearch.com
  ...
```

Figure 7: Abbreviated CSP of `spotify.com`.

that Spotify became aware of the issue with sideloaded gadgets, they could choose to explicitly whitelist on the necessary script, namely the reCaptcha API. This, however, would not suffice to secure the side from a script gadget-based attack. The reason is the fact that `sb.scorecardresearch.com` is whitelisted. Our analysis showed that this host has an open redirect. Therefore, this could be used by the attacker to add a script resource pointing to the open redirect, making sure that the redirection target is AngularJS on `gstatic.com`. As discussed before, CSP disables the path matching if a resource is loaded as the result of a redirect, meaning that any script from `gstatic.com` would be allowed. This example highlights the potential impact an open redirect can have on a CSP's security. Hence, in order to secure one's application properly against the dangers of sideloaded scripts, next to ensuring that no hostnames, from which Angular or other libraries could be loaded, are whitelisted, a site operator also has to make sure that none of the other whitelisted entries contain an open redirect.

6 HYPOTHETICAL IMPACT

Our findings thus far have shown highlighted a number of insights already: first, only around 10% of all sites in our dataset even deploy CSP. Applying our notion of a reasonably secure policy, only one in four of those sites can be considered to be secure against regular script injection. Notably, though, our analysis has shown that of those, around 10% are susceptible to bypasses, including major organizations like Snapchat and Spotify. Given that these high-profile sites are already trusting only a handful of entities (e.g., Snapchat only whitelists their own site and Google properties), we cannot assume that the average Web site could deploy a similarly strict policy. Thus, we extend our research question such that it also covers a hypothetical case: what if every site in the top 10,000 were to deploy a sane CSP today? To understand the severity of the impact of script gadgets on this desirable future of CSP deployment, we first discuss how we generate such policies for the Tranco Top 10,000 sites. We then follow up with an analysis on how many of those policies could be bypassed with script gadgets.

6.1 Methodology

To further investigate the impact of the third-party based CSP bypass, we take a look at the prevalence of this issue under the assumption that all sites were to deploy a sensible CSP. For this, we rely on the information already collected (see Section 4.1) about scripts that are included by the analyzed sites. Although there are more involved approaches to generate a CSP [5, 9, 18], we resort to generating CSP in a light-weight fashion. In particular, while many sites could not deploy CSP because of their reliance on inline scripts, we focus on curating only host-based whitelists, as this is

```
scriptSrc = set()
data = clusterSources(scriptSources)
for host, urls in data:
  scriptSrc.add(host) if len(urls) > 1 else scriptSrc.add(urls[0])
```

Figure 8: Generate `script-src` algorithm.

the primary target for the attacks. While removing inline event handlers is infeasible for many sites (and is a major contribution factor to CSP's lack of success [19], our hypothetical experiment is meant to understand the impact of script gadgets on CSP's ability to mitigate script injection. Therefore, we assume that each site could get rid of inline handlers and nonce all inline scripts, i.e., the generated CSP would be a host-based whitelist only.

Our CSP script source generation algorithm first clusters the script sources used within the crawled site based on their host. This provides us with a mapping of hostnames to script URLs on said hosts. If a given host only hosts a single script, we add the complete URL to the CSP. For all hosts with multiple URLs, we align our implementation with what our observations in the wild indicate: we add the full hostname to the CSP.

Notably, our procedure of automatically generating script-src directives based on the script usage we found is only a lower bound for the CSP. Due to this natural limitation of automated crawling, we might have missed the usage of libraries that would allow exploitation, as mentioned in Section 5.3. Therefore also our results regarding the exploitability of the hypothetical generated CSPs for the Top 10,000 Tranco Web sites, is only a lower bound.

A code example of this algorithm that generates sensible script content restricting directives is depicted in Figure 8. With this algorithm we automatically generated `script-src` directives that are secure with respect to our definition from Section 5.1. We reuse our exploit generation and validation presented in Section 5 to investigate whether they would be susceptible to a bypass or not. In our preliminary tests, we found that not all libraries, which are detected as being AngularJS, actually contain a script gadget; in particular, both the modules `cookie` and `sanitize` of Angular are detected as being AngularJS. To ensure that our analysis does not yield false positives, we built a simple test page, hosted locally. This page contained the payload to be consumed by the gadget. Next, for each URL pointing to what was identified as AngularJS, we included this as a script and determined if the injected payload was executed. If not, we marked the URL as a non-working version (w.r.t. our attack) of AngularJS. In doing so, we found that 1,399 of 2,349 angular-labeled libraries were usable for our attack. Based on this list, we determined if the generated CSP would allow the inclusion of at least one of the 1,399 scripts; which would essentially allow an attacker to sideload a working version of AngularJS.

6.2 Results

For the remaining sites, we were either redirected off-site (e.g., to a data protection regulation interstitial), or they did not respond to HTTP requests. One such example is `samsungcloud.com`, which does not resolve to an IP address (with or without `www`), but works for `support.samsungcloud.com` (according to a check on Google).

On average the generated `script-src` directives for the remaining pages contained 9.82 source expressions. The most popular

Table 5: Top 8 AngularJS sources used for direct hypothetical bypasses.

Angular Source Host	Affected Sites
ajax.googleapis.com	931
www.gstatic.com	477
tpc.googlesyndication.com	217
cdn.jsdelivr.net	127
cdn.spotxcdn.com	53
g.alicdn.com	29
nebula-cdn.kampyle.com	28
cdn.playbuzz.com	23

Table 6: Top 8 Open Redirect sources used in the hypothetical exploits.

Open Redirect Host	Affected Sites
securepubads.g.doubleclick.net	482
sb.scorecardresearch.com	37
bs.serving-sys.com	33
ib.adnxs.com	24
dsp.adfarm1.adition.com	19
secure.adnxs.com	18
gum.criteo.com	17
contextual.media.net	10

external source expression was https://www.google-analytics.com/analytics.js, which would need to be whitelisted by 5,216 distinct sites. The Web site with the highest number of whitelisted entries (226) was `www2.deloitte.com`. However, this number is only that high because our CSP generation only worked with distinct hostnames and ignored if it would be easier to whitelist the parent domain. Therefore `sp<randomString>.guided.ss-omtrdc.net` is whitelisted 159 times instead of only once. In contrast to this high number, 2,024 sites only needed one source expression in their CSP, which was, in most cases, only the `'self'` expression. We note that our analysis does not consider inline scripts, i.e., to even reach such a comparatively secure state, those sites would have to refrain from inline scripts or deploy nonces or hashes.

Out of the 8,330 Web sites for which our CSP generation yielded a `script-src`, we found that the policies for 3,441 sites were actually bypassable through sideloading. Table 5 shows the hosts most frequently chosen by our exploit generation to include AngularJS. It is worth noting that contrary to the real-world exploitability, `www.gstatic.com` is not the most impactful host. This is due to the fact that our CSP generation resorts to whitelisting full URLs if only a single script is included from a given host. In particular, we found that while resources from `www.gstatic.com` are widely used, in most cases, the only used library is the reCaptcha API. Instead, the best AngularJS source used as direct sideloading targets for our hypothetical exploitation, with 931 affected sites, is Google's CDN (`ajax.googleapis.com`). One reason why this CDN, or CDNs in general, perform that well is that, developers tend to use the same CDN for multiple libraries, leading to a fully whitelisted CDN host.

Another major difference in comparison to the real-world exploitability is that in the case of the hypothetical bypasses, 1,654 Tranco sites would only be attackable due to the open redirects. This again can be attributed to our CSP generation only whitelisting full URLs when a single resource from a host is required. As depicted in Table 6 most of those open redirects (482) were possible due to `securepubads.g.doubleclick.net` being whitelisted.

The fact that we were able to automatically generate and validate bypasses for the artificially created CSPs of more than 3,441 sites that are part of the Tranco Top 10,000 indicates the severity of our bypass. We also note the severe impact of open redirects, especially on a popular site such as Doubleclick. One potential solution to sideloading is the usage of hashes and nonces in CSP. The problem of nonce-based policies, though, is that especially ad providers tend to add additional scripts to the page, which would either necessitate them to explicitly add nonces to the newly introduced scripts or, more likely, force the first party to deploy `strict-dynamic` [27]. However, if `strict-dynamic` is present, nearly all libraries with script gadgets found by Lekies et al. [14] enable an attacker to sideload other scripts. This, in turn, would allow an attacker to sideload, e.g., AngularJS, opening the site up for our outlined attacks.

6.3 Hypothetical case study: reddit.com

As a sanity check and to understand why our secure CSP script source directives are vulnerable against our bypass, we manually investigated the generated policies as well as our generated exploits.

The social news aggregation site `reddit.com` does not deploy a CSP in its real-world application, thus is was targeted by our hypothetical analysis. The generated script-src directive is depicted in Figure 9. Due to the nature of our generation, the policy is secure, according to our definition from Section 3. It uses jQuery loaded using the googleapis.com, but no other script from this source, thus it is whitelisted as full URL. A script source which is present in the whitelist as a full domain is `securepubads.g.doubleclick.net`, which is due to the fact that scripts loaded from this domain have random identifiers or timestamps in their *path*. Given that this site is fully whitelisted, but importantly also contains open redirects. As discussed before, CSP relaxes the path matching if a resource is loaded as the result of a redirect. Thus we are able to load the AngularJS library from `ajax.googleapis.com` although it is not explicitly present in the whitelist.

This representative example is only one of the 1,654 sites that are attackable due to open redirect vulnerabilities. In order to secure their site from this vulnerability, Reddit would need to move from host-based whitelisting to hash and nonce based whitelisting.

```
script-src
  'self'
  https://ajax.googleapis.com/ajax/.../jquery.min.js
  https://adservice.google.com/adsid/integrator.js
  securepubads.g.doubleclick.net
  www.googletagservices.com
  www.redditstatic.com
  ...
```

Figure 9: Generated (abbreviated) CSP for `www.reddit.com`.

However, due to several scripts loads being triggered programmatically, they would need to use the strict dynamic mode. However, the `strict-dynamic` expression is not universally supported by all browsers, as mentioned in Section 2.2. Thus, Reddit would either need to pass along nonces programmatically or would have a CSP that is incompatible with some browsers. Furthermore, script-dynamic has shown to be bypassable by several script gadgets. Thus, removing the vulnerability would only be possible by massively changing reddit.com as a web application.

6.4 Limitations and Potential Modifications

For the validation of the CSP bypasses, we assume that a markup injection vulnerability is present on every Web site in our dataset. Notably, the goal of our work is not to show that Web sites are vulnerable, but we want to show how hard the generation of a secure host-based whitelist is on the modern Web. CSP was initially designed to mitigate the effect of XSS attacks and to investigate the effectiveness of host-based whitelists in the modern Web. We only investigate CSP in isolation. In addition to that, research has shown that a non-negligible fraction of sites suffers from markup injection vulnerabilities [13, 15, 30].

Furthermore, even if we could find XSS vulnerabilities in the wild, automatically verifying exploitability is infeasible due to external effects caused by, e.g., Web application firewalls. Irrespective of this drawback, we find that there is no reasonably ethical way to confirm the problem at scale, and hence refer to our discussed simulation of an injection (fully on the client-side).

One cornerstone in our whitelist generation is the threshold for adding an origin instead of multiple URLs from the same origin. Naturally, the higher this threshold, the lower is the chance of whitelisting an entire origin and thereby a script gadget library. We experimented with changing the number of URLs that are necessary to whitelist a full origin. The results of this analysis are shown in Figure 10. The x-axis shows the threshold, i.e., the value 2 implies that at least URLs must be loaded from the same origin before we add the origin. This is the baseline we used for our measurements. Naturally, the higher the threshold, the more secure the site is. However, it must be noted that we did not achieve any meaningful coverage of the applications in our crawl, i.e., the amount of included JavaScript we observed is likely a lower bound.

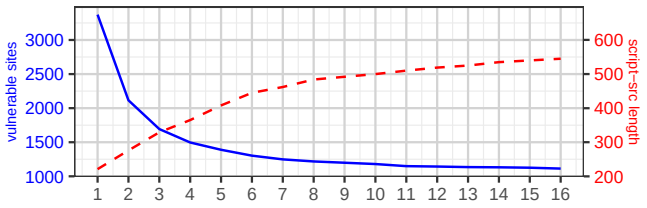

Figure 10: Impact of the URL to domain threshold.

7 DISCUSSION

Analyzing the root causes of the issues first described by Weichselbaum et al. [27], we find that several factors contribute to the danger of script gadgets. First, these snippets bypass one of the fundamental assumptions of CSP: that, given the absence of `unsafe-eval`, only code originating from the developer's whitelisted sources can be executed. In particular, script gadgets enable the transformation from string data to code (similar to what `eval` enables) and cannot be controlled via CSP. This opens any site making use of a library containing a script gadget to attacks. Importantly, though, another major reason comes into play when considering our results of one-third of sites being attackable even in the presence of a sane CSP, namely hosting libraries for diverse purposes on the same origin.

In contrast, one major security benefit originates from the separation of content. As an example, social networks like Facebook or Twitter do not allow users to upload any content to their own origins, but instead to twimg.com and fbcdn.net, respectively. The reason behind this is simple: even though both sites likely employ mechanisms to ensure that uploaded images are not HTML markup or active objects like Flash (which, importantly retains its origin if included from other sites), they employ a defense-in-depth approach. Hence, if an attacker manages to bypass upload filters to upload HTML, this does not reside on the main page, and hence cannot be leveraged to attack the sites.

Arguably, many of the sites we found to be vulnerable would easily benefit if CDNs approached their different content in a similar fashion. In particular, if Google decided to split up their hosts containing the reCaptcha snippets and AngularJS, sites leveraging the reCaptcha APIs would not be susceptible to script gadget attacks. We note that our crawler, in fact, did not detect AngularJS hosted by gstatic.com, and we instead had to rely on a list of known AngularJS sources. This highlights the absence of the necessity to even host AngularJS on that particular site. While the same level of protection as separating libraries could theoretically be achieved by whitelisting the entire path to the reCaptcha API, our findings regarding open redirects indicate that this does not suffice. Hence, even if the full reCaptcha URL is whitelisted, as long as a single other entry points to an open redirect, any scripts on the hosts of the reCaptcha script can be loaded.

Another potential improvement for the Web's susceptibility to script gadgets is the removal of open redirects. As our analysis has shown, sites that contain such redirect flaws are often related to advertisement and analytics, which are used frequently and are thus trusted by a large body of sites. Arguably, such open redirects may well serve a purpose, e.g., an ad provider can leverage this URL first to collect statistics about clicked links and subsequently redirect users to the target site. Notably, though, this suffers from the same problem as hosting all sorts of libraries on the same site: the sites are mixing up different use cases under the same site. An easy way of addressing this problem is to have specific (sub)domains, which are only meant to be used for redirects. In this way, a site could still whitelist an ad network's entire domain for script content, but given that this domain would no longer have open redirects would no longer be susceptible to redirect-enabled script gadget attacks.

While prior work has already indicated the problem in real-world policies, our hypothetical analysis adds another order of magnitude to the problem: even in a perfect world, in which CSP is widely deployed, the reliance of sites on third parties for their code enables bypasses through the combination of script gadgets and open redirects. Given that the CSP is unlikely to be changed due to the perceived grave threat of leaking sensitive path information after

redirects, we instead call on CDNs to resort to meaningful separation. In particular, Lekies et al. [14] already contains an extensive list of script gadget libraries, which could be easily separated from more meaningful content. This comes with the drawback that, assuming a site wants to use both AngularJS and the reCaptcha API regularly, CSPs would become longer, it does nevertheless reduce the attack surface for sideloading script gadgets.

Based on the result from Section 6, we argue that our former definition of a meaningfully secure policy needs to shift when we consider script gadgets. By trusting a domain, the developer does not only trust those resources that they are directly loading, but also every resource that is hosted by this domain. In order to create a secure policy, the developer needs to be sure that *none of the trusted domains* have any open redirects and/or are not hosting a script-gadget library. In practice, such a priori knowledge is infeasible to come by, as even thorough crawls cannot find libraries that are only included behind, e.g., logins. Hence, a developer may spend significant efforts in hardening they CSP, which is bypassable through an open redirect and/or gadget library on a whitelisted sites. Furthermore, restricting a whitelist to sites that do not host any gadget library blocks the site from using a CDN. Assuming that no other whitelisted source suffers from an open redirect vulnerability, a developer needs to only whitelist full URLs. This may, however, be brittle when considering that third parties often add more scripts [10, 11], for which URLs can change, which would violate the CSP. Hence, we overall find that the prevalence of script gadgets on many widely used hosts severely impairs CSP's ability to mitigate XSS attacks.

8 RELATED WORK

In this section, we discuss how our work relates to prior research. In particular, this covers works that investigate the (in)security of CSP in the wild as well as mechanisms that can be used to automatically generate CSPs or make applications compliant with CSP. One of the earliest works on CSP was done in 2013 by Doupé et al. [5]. The authors proposed a tool, which externalized inline scripts, allowing for deployment of CSP without `'unsafe-inline'`. The first measurement study on the adoption of CSP in the wild was conducted by Weissbacher et al. [28] in 2014. They found that only 1 of the Top 100 Alexa sites enforced a CSP. To explain this low adoption of the security mechanism, they tried to create policies for three different hosts on their own and found that the creation of an initial CSP requires massive engineering efforts. This is not only due to finding and whitelisting all sources, which can be error-prone, but also because the then-current CSP spec did not support nonces, making inline scripts a severe roadblock for CSP deployment.

The insecurity of CSPs deployed in the wild was shown by two works in 2016. Next to showing that well above 90% of unique policies discovered in the wild were trivially bypassable, Weichselbaum et al. [27] also first indicated the danger of having whitelists with AngularJS-hosting sites. Notably, though, their paper assumed that after detection of what seemed to be AngularJS (the paper claims to do so by checking certain bytes of AngularJS, without going into further detail) on a given site, a whitelist entry pointing to that site would automatically be a bypass for CSP. In contrast, our analysis has shown that this is not always the case, given the inaccuracies of detection libraries. Furthermore, their paper proposed the `strict-dynamic` expression, which is also part of the CSP Level 3 standard in order to not require extensive application changes. In contrast to Weichselbaum et al., we not only highlight *potential* issues but validate our findings with PoCs, showing that merely relying on library detection tools may yield false positive. We further consider open redirects for the sideloading of script-gadget libraries and created a hypothetical risk evaluation in a scenario where every Web site is using a CSP that is not trivially bypassable. Calzavara et al. [3] investigated similar problems in CSP deployment. During their longitudinal analysis over four months, the authors found that CSPs changed less frequently than necessary. Later on, they extended their work to a six months analysis. In doing so, the authors discovered that the overall quality of CSP slightly improved, essentially increasing the mitigation potential against XSS attacks. They attribute this trend to the grown usage of nonces in policies, although the authors also point out that the majority of policies in the wild do not seem to be related solely to script content restriction [4]. Our work has found a similar trend in the usage of CSP, where only a minuscule fraction of sites deploy a meaningful CSP (in terms of script code restriction).

Web applications' struggle to deploy CSP was also investigated by Kerschbaumer et al. [9]. They showed that many of the CSP deploying Web sites use the `unsafe-inline` expression to avoid the rewriting of their applications. In order to solve this problem, they created a crowd-sourced learning system that is able to generate CSP policies for a given application automatically. In addition, Pan et al. [18] built a tool capable of generating CSP by rewriting sites, showing that such an automated system works for the Alexa top 50 sites. While our work does not tackle the problem of rewriting applications to be CSP-compliant, we leverage ideas from them for the hypothetical analysis of what could be.

In recent years, attacks on CSP have become a prominent research topic. In 2015 Hausknecht et al. [7] discovered that browser extensions frequently perform invasive modification on both the page content and the CSP. Van Acker et al. [23] showed that CSP is not able to prevent data leakage, in particular, when DNS and resource prefetching are used as channels. In addition, a more recent paper from Some et al. [21] showed that even if CSP is deployed, given its inconsistent deployment throughout an application as well as concepts like domain relaxation, it may be bypassable.

Most influential for our work naturally is the paper from Lekies et al. [14], which first discussed the concept of script gadgets and also mentioned the possible bypass of CSPs using AngularJS from whitelisted sources. While Lekies et al. were rather unspecific about which Web sites are exploitable, we generated and validated exploits for real-world Web sites from the Tranco Top 10,000. Furthermore, we do not only consider cases were the script-gadget containing library is actively used on a Web site, but also cases where we are able to load this library from one of the whitelisted parties. In addition to that, we try to probe the cause of why certain sites whitelist dangerous script sources by analyzing the specific use cases of multiple Web sites. Furthermore, we extend the analysis by hypothetically evaluating the bypassability in case of CSP being deployed on every Web site that we investigated. The first academic work that tackles the problem of unvalidated or open redirects was Shue et al. [20].

They created several heuristics in order to identify dynamic URL redirections by searching for the http prefix. Using these heuristics, they found out that around 80% of all redirects that they found were actually open redirects. Our work has shown that now, more than ten years later, the issue of open redirects is still present with 114 domains providing open redirects.

9 CONCLUSION

In this work, we aimed to understand how shaky CSP's foundation is when we consider script gadgets and open redirects. To answer our main research question, we first showed how sideloading script gadgets can increase the attack surface of a CSP-secured Web application. To that end, we analyzed the Tranco top 10,000, looking for both sources hosting script gadgets as well as sites with open redirect issues. Based on the gathered insights, we then conducted an analysis of real-world CSP so as to understand the susceptibility of the sites deploying CSP against the threat of sideloaded gadgets. In doing so, we found that of the few sites that even securely use CSP to restrict content, 10% are susceptible to bypasses through whitelisted script gadgets. Our results hint at the fact that being overly permissive in whitelisting the entire `www.gstatic.com` host is a major contributor. Notably, though, we found evidence that even when resources from the said origin are explicitly whitelisted by their full URL, whitelisted open redirects allow an attacker to include AngularJS nevertheless. In addition to the real-world analysis, which draws a skewed picture given the minuscule deployment of CSP in the wild, we ran a hypothetical experiment, curating host-based whitelists based on the scripts used by the Top 10,000 sites. Overall, we found that which a conservative approach to automatically generating CSPs, 3,441/8,330 sites for which a CSP was auto-generated could still fall victim to a script gadget attack. In particular, of those bypassable sites, 1,654 were susceptible to due whitelisted open redirects, highlighting again the threat this insecure practice may cause to *other* Web sites.

To mitigate the presented CSP bypass, a practice that is already widely used throughout the Web can be adopted. The best practice for hosting uploaded passive content like images is to host them under a different domain. If CDN providers leverage this practice by only hosting the known-dangerous JavaScript libraries on domains separate from their remaining code, this would greatly reduce the attack surface for sideloading script-gadgets. Similarly, while analytics and ad providers often leverage open redirects for different purposes, the design choice to have such redirect URLs on the domain which needs to be whitelisted by CSP for the main functionality of the provider widens the attack surface as well. Here, we propose the same concept of separation, i.e., hosting the redirects on a separate domain, which does not require whitelisting in CSP.

Summarizing, we find that CSP's ability to protect sites from XSS is even further impaired by script gadgets and open redirects. Importantly, this means that a third party — without any ill intentions — may put sites which trust it to host benign functionality may accidentally undermine the first party's CSP. In particular, an operator trying to deploy a secure CSP must be well-aware of *any* site hosting such a library, as well as any of their whitelisted hosts with open redirects. We hope that our paper ensures that this threat gets its well-deserved attention, such that CSP can be used to help secure the Web even in the presence of script gadgets; i.e., by ensuring that libraries which can be abused as script gadgets are hosted separately from utility functionality.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their insightful feedback on how to better frame our paper and improve its contributions. In particular, we also thank our shepherd Tobias Lauinger for his guidance in the shepherding process.

REFERENCES

[1] A. Barth. RFC 6454. *Online at https://www.ietf.org/rfc/rfc6454.txt*, 2011.
[2] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0. *Online at https://www.ietf.org/rfc/rfc1945.txt*, 1996.
[3] S. Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *CCS*, 2016.
[4] S. Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-Based Analysis of Content Security Policy Deployment. *TWEB*, 2018.
[5] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In *CCS*, 2013.
[6] GitHub - GoogleChrome. Puppeteer. *Online at https://github.com/GoogleChrome/puppeteer*, 2019.
[7] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May I?-Content Security Policy Endorsement for Browser Extensions. In *DIMVA*, 2015.
[8] E. Homakov. Using Content Security Policy for Evil. *Online at http://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html*, 2014.
[9] C. Kerschbaumer, S. Stamm, and S. Brunthaler. Injecting CSP for Fun and Security. In *ICISSP*, 2016.
[10] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, A. J. Halderman, and M. Bailey. Security challenges in an increasingly tangled web. In *WWW*, 2017.
[11] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *NDSS*, 2017.
[12] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *NDSS*, 2019.
[13] S. Lekies, B. Stock, and M. Johns. 25 million flows later: Large-scale detection of dom-based xss. In *CCS*, 2013.
[14] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *CCS*, 2017.
[15] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In *NDSS*, 2018.
[16] Microsoft Developer. EdgeHTML Platform Status. *Online at https://tinyurl.com/skxpgsy*, 2019.
[17] E. Oftedal. Retire.js: What you require you must also retire. *Online at https://retirejs.github.io/retire.js/*, 2018.
[18] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box enforcement of Content Security Policy upon Real-World Websites. In *CCS*, 2016.
[19] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *NDSS*, 2020.
[20] C. A. Shue, A. J. Kalafut, and M. Gupta. Exploitable Redirects on the Web: Identification, Prevalence, and Defense. In *WOOT*, 2008.
[21] D. F. Some, N. Bielova, and T. Rezk. On the Content Security Policy Violations due to the Same-Origin Policy. In *WWW*, 2017.
[22] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *WWW*, 2010.
[23] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Data Exfiltration in the Face of CSP. In *AsiaCCS*, 2016.
[24] W3C. CSP 1.0. *Online at https://www.w3.org/TR/CSP1/*, 2015.
[25] W3C. CSP Level 2. *Online at https://www.w3.org/TR/CSP2/*, 2016.
[26] W3C. CSP Level 3. *Online at https://www.w3.org/TR/CSP3/*, 2016.
[27] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In *CCS*, 2016.
[28] M. Weissbacher, T. Lauinger, and W. Robertson. Why is CSP failing? Trends and challenges in CSP adoption. In *RAID*, 2014.
[29] WHATWG HTML Standard. The Script Element. *Online at https://html.spec.whatwg.org/multipage/scripting.html#the-script-element*, 2019.
[30] WhiteHat Security. 2018 Whitehat Application Security Statistics. *Online at https://www.whitehatsec.com/blog/2018-whitehat-app-sec-statistics-report/*.