Magdeburger Journal zur Sicherheitsforschung

Gegründet 2011 | ISSN: 2192-4260
Herausgegeben von Stefan Schumacher
Erschienen im Magdeburger Institut für Sicherheitsforschung
http://www.sicherheitsforschung-magdeburg.de/publikationen/journal.html

This article appears in the special edition »In Depth Security – Proceedings of the DeepSec Conferences«.
Edited by Stefan Schumacher and René Pfeiffer

CSP Is Dead, Long Live CSP!

On the Insecurity of Whitelists and the Future of Content Security Policy

Lukas Weichselbaum and Michele Spagnuolo and Sebastian Lekies and Artur Janc

Content Security Policy is a web platform mechanism designed to mitigate cross-site scripting (XSS), the top security vulnerability in modern web applications MITRE, 2014. In this paper, we take a closer look at the practical benefits of adopting CSP and identify significant flaws in real-world deployments that result in bypasses in 94.72% of all distinct policies.

We base our Internet-wide analysis on a search engine corpus of approximately 100 billion pages from over 1 billion hostnames; the result covers CSP deployments on 1,680,867 hosts with 26,011 unique CSP policies – the most comprehensive study to date. We introduce the security-relevant aspects of the CSP specification and provide an in-depth analysis of its threat model, focusing on XSS protections. We identify three common classes of *CSP bypasses* and explain how they subvert the security of a policy.

We then turn to a quantitative analysis of policies deployed on the Internet in order to understand their security benefits. We observe that 14 out of the 15 domains most commonly whitelisted for loading scripts contain *unsafe endpoints*; as a consequence, 75.81% of distinct policies use script whitelists that allow attackers to bypass CSP. In total, we find that 94.68% of policies that attempt to limit script execution are ineffective, and that 99.34% of hosts with CSP use policies that offer no benefit against XSS.

Finally, we propose the `'strict-dynamic'` keyword, an addition to the specification that facilitates the creation of policies based on cryptographic nonces, without relying on domain whitelists. We discuss our experience deploying such a *nonce-based* policy in a complex application and provide guidance to web authors for improving their policies.

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

Editor's note: This article has already been published in: Weichselbaum, Spagnuolo, Lekies, and Janc (2016)

Citation: Weichselbaum, L., Spagnuolo, M., Lekies, S., & Janc, A. (2017). CSP Is Dead, Long Live CSP!: On the Insecurity of Whitelists and the Future of Content Security Policy. *Magdeburger Journal zur Sicherheitsforschung*, *14*, 779–796. Retrieved August 10, 2017, from http://www.sicherheitsforschung-magdeburg.de/uploads/journal/MJS_053_Weichselbaum_CSP.pdf

Version 2017/08/09 14:59

1 Introduction

Cross-site scripting – the ability to inject attacker-controlled scripts into the context of a web application – is arguably the most notorious web vulnerability. Since the first formal reference to XSS in a CERT advisory in 2000 CERT, 2000, generations of researchers and practitioners have investigated ways to detect Jovanovic, Kruegel, and Kirda, 2006; Lekies, Stock, and Johns, 2013; Saxena, Hanna, Poosankam, and Song, 2010; Wassermann and Su, 2008, prevent Louw and Venkatakrishnan, 2009; Nadji, Saxena, and Song, 2009; Vogt et al., 2007 and mitigate Bates, Barth, and Jackson, 2010; Maone, 2017; Ross, 2008; Stock, Lekies, Mueller, Spiegel, and Johns, 2014 the issue. Despite these efforts, XSS is still one of the most prevalent security issues on the web MITRE, 2014; Security, 2013; Wichers, 2013, and new variations are constantly being discovered as the web evolves Bojinov, Bursztein, and Boneh, 2009; Heiderich, Niemietz, Schuster, Holz, and Schwenk, 2012; Heiderich, Schwenk, Frosch, Magazinius, and Yang, 2013; Klein, 2005.

Today, Content Security Policy Stamm, Sterne, and Markham, 2010 is one of the most promising countermeasures against XSS. CSP is a declarative policy mechanism that allows web application developers to define which client-side resources can be loaded and executed by the browser. By disallowing inline scripts and allowing only trusted domains as a source of external scripts, CSP aims to restrict a site's capability to execute malicious client-side code. Hence, even when an attacker is capable of finding an XSS vulnerability, CSP aims to keep the application safe by preventing the exploitation of the bug – the attacker should not be capable of loading malicious code without controlling a trusted host.

In this paper, we present the results of the first in-depth analysis of the security of CSP deployments across the web. In order to do so, we first investigate the protective capabilities of CSP by reviewing its threat model, analyzing possible configuration pitfalls and enumerating little-known techniques that allow attackers to bypass its protections.

We follow with a large-scale empirical study using real-world CSP policies extracted from the Google search index. Based on this data set, we find that currently at least 1,680,000 Internet hosts deploy a CSP policy. After normalizing and deduplicating our data set, we identify 26,011 unique CSP policies, out of which 94.72% are *trivially bypassable* – an attacker can use automated methods to find endpoints that allow the subversion of CSP protections. Even though in many cases considerable effort was spent in deploying CSP, 90.63% of current policies contain configurations that immediately remove any XSS protection, by allowing the execution of inline scripts or the loading of scripts from arbitrary external hosts. Only 9.37% of the policies in our data set have stricter configurations and can potentially protect against XSS. However, we find that at least 51.05% of such policies are still bypassable, due the presence of subtle policy misconfigurations or origins with unsafe endpoints in the `script-src` whitelist.

Based on the results of our study, we conclude that maintaining a secure whitelist for a complex application is infeasible in practice; hence, we propose changes to the way CSP is used. We suggest that the model of designating trust by specifying URL whitelists from which scripts can execute should be replaced with an approach based on *nonces and hashes* A Barth, Veditz, and West, 2014, already defined by the CSP specification and available in major browser implementations.

In a *nonce-based* policy, instead of whitelisting hosts and domains for script execution, the application defines a single-use, unguessable token (nonce) delivered both in the CSP policy and as an HTML attribute of legitimate, application-controlled scripts. The user agent allows the execution only of those scripts whose nonce matches the value specified in the policy; an attacker who can inject markup into a vulnerable page does not know the nonce value, and is thus not able to execute malicious scripts. In order to ease the adoption process of this nonce-based approach, we present a new CSP source expression for `'script-src'`, provisionally called `'strict-dynamic'`. With `'strict-dynamic'`, dynamically generated scripts implicitly inherit the nonce from the trusted script that created them. This way, already-executing, legitimate scripts can easily add new scripts to the DOM without extensive application changes. However, an attacker who finds an XSS bug, not knowing the correct nonce, is not able to abuse this functionality because they are prevented from executing scripts in the first place.

In order to prove the feasibility of this approach, we present a real-world case study of adopting a nonce-based policy in a popular application.

Our contributions can be summarized as follows:

- We present the results of the first in-depth analysis of the CSP security model, analyzing the protections against web bugs provided by the standard. We identify common policy misconfigurations and present three classes of CSP bypasses that disable the protective capabilities of a policy.
- We conduct a large-scale empirical study of the benefits of real-world CSP deployments by extracting policies from the Google search index. Based on a corpus of approximately 106 billion pages, of which 3.9 billion are protected with CSP, we identify 26,011 unique policies. We find that at least 94.72% of these policies are ineffective at mitigating XSS, due to policy misconfigurations and insecure whitelist entries.
- Based on our findings, we propose a change to how Content Security Policy is deployed in practice: instead of whitelisting, we advocate for a nonce-based approach. To further this approach, we present `'strict-dynamic'`, a new

feature of the CSP3 specification currently implemented in the Chromium browser. We discuss the benefits of this approach and present a case study of deploying a policy based on nonces and `strict-dynamic` in a popular web application.

The rest of this paper has the following structure: in Section 2, we provide an in-depth introduction to CSP. Thereby, we cover the technical foundations in 2.1, the CSP threat model and common security pitfalls when designing a policy in 2.2 and 2.3. Subsequently, we present the result of our empirical study in Section 3. In order to do so, we first outline our research questions in 3.1, introduce our data set in 3.2, and explain our methodology in 3.3, before we present the results and our analysis in 3.4. Based on the results of this study, we then propose a way to improve CSP in Section 4. Finally, we present related work in Section 5, before we conclude in Section 6.

2 Content Security Policy

2.1 Overview

The Content Security Policy (CSP) is a declarative mechanism that allows web authors to specify a number of security restrictions on their applications, to be enforced by supporting user agents.

CSP is intended as *"a tool which developers can use to lock down their applications in various ways, mitigating the risk of content-injection vulnerabilities (...) and reducing the privilege with which their applications execute."* A Barth et al., 2014

CSP is evolving quickly: the version currently undergoing specification is CSP3, and the standard is unevenly implemented by user agents. For example, Chromium has full CSP2 support and implements most of the working draft of CSP3, in some cases behind experimental runtime flags, while Mozilla Firefox and WebKit-based browsers just recently obtained full CSP2 support Foundation, 2016. When discussing the details of CSP, we do not focus on any particular revision of the standard, but instead attempt to provide a broad overview across implementations and versions Stamm et al., 2010.

A CSP policy is delivered in the `Content-Security-Policy` HTTP response header or in a `<meta>` element. The functionality of CSP can be divided into three categories:

Resource loading restrictions. The most well-known and commonly used aspect of CSP is limiting the ability to load various subresources to a set of origins allowed by the developer, known as a *source list*. Commonly used directives are `script-src`, `style-src`, `img-src`, and the catch-all `default-src`; a full list of directives regulating resources is shown in Table 1. As a special case, several additional configuration options are available for the `script-src` and `style-src` directives; these allow more fine-grained control over scripts and stylesheets and are discussed below.

Auxiliary URL-based restrictions. Certain classes of attacks cannot be prevented by policing fetched subresources, but similarly require a concept of trusted origins with which the document can interact. A common example is the `frame-ancestors` directive, which defines the origins that are allowed to frame a document in order to prevent clickjacking Hansen and Grossman, 2008. Similarly, `base-uri` and `form-action` define which URLs can be targets of `<base\#href>` and `<form\#action>` elements in order to prevent some *post-XSS* attacks Zalewski, 2011a.

Miscellaneous confinement and hardening options. Due to the lack of other common mechanisms for enabling security restrictions in web applications, CSP has become the home for several loosely fitting security features. This includes the `block-all-mixed-content` and `upgrade-insecure-requests` keywords, which prevent mixed content bugs and improve HTTPS support; `plugin-types`, which restricts allowed plugin formats; and `sandbox`, which mirrors the security features of HTML5 sandbox frames.

In order to make web applications compatible with a Content Security Policy useful against XSS, web authors often have to refactor the HTML markup generated by the application logic, as well as by frameworks and templating systems. In particular, inline scripts, the usage of `eval` and equivalent constructs, inline event handlers and `javascript:` URIs must be avoided or refactored with CSP-friendly alternatives.

In addition to the default behavior of enforcing policy restrictions, CSP can be configured in `Report-Only` mode, in which violations are recorded but not enforced. In both cases, the `report-uri` directive can be used to send violation reports to inform the application's owner of incompatible markup.

Listing 1: Example of a traditional CSP policy

```
Content-Security-Policy: script-src 'self'; ↙
    ↳ style-src
cdn.example.org third-party.org; child-src https:
```

2.1.1 Source lists

CSP source lists (commonly known as *whitelists*) have been a core part of CSP and are the traditional way to specify trust relationships. For example, as shown in Listing 1, an application might choose to trust only its hosting domain for loading scripts, but allow fonts or images from `cdn.example.org` and `third-party.org`, and require frames to be loaded over HTTPS, while enforcing no restrictions on other resource types.

For any directive, the whitelist can be composed of hostnames (`example.org`, `example.com`), possibly including the `*` wildcard to extend trust to all subdomains (`*.example.org`); schemes (`https:`,

Directive	Controlled resource type
`default-src`	All resources (fallback)
`script-src`	Scripts
`style-src`	Stylesheets
`img-src`	Images
`media-src`	Media (audio, video)
`font-src`	Fonts
`frame-src`	Documents (frames)
`object-src`	Plug-in formats (object, embed)
`child-src`	Documents (frames), [Shared]Workers
`worker-src`	[Shared]Workers
`manifest-src`	Manifests

Table 1: CSP directives and controlled resources

`data:`); and the special keywords `’self’`, denoting the origin of the current document, and `’none’`, enforcing an empty source list and prohibiting the loading of any resources whatsoever.

Starting with CSP2, authors also have the option to specify paths in their whitelists (`example.org/resources/js/`). Interestingly, path-based restrictions cannot be relied on to limit the location from which resources can be loaded; a broader discussion of this issue is provided in Section 2.3.4.

2.1.2 Restrictions on script execution

Because of the significance of scripting in modern web applications, the `script-src` directive provides several keywords to allow more granular control over script execution:

1. `unsafe-inline` allows the execution of inline `<script>` blocks and JavaScript event handlers (effectively removing any CSP protection against XSS).
2. `unsafe-eval` allows the use of JavaScript APIs that execute string data as code, such as `eval()`, `setTimeout()`, `setInterval()`, and the `Function` constructor. Otherwise, these APIs are blocked by a policy with a `script-src` directive.
3. A *CSP nonce* allows the policy to specify a one-time value that acts as an authorization token for scripts (`script-src’nonce-{random-value}’`). Any script on the page with the correct `nonce="{random-value}"` attribute will be allowed to execute.
4. A *CSP hash* allows the developer to list cryptographic hashes of expected scripts within the page (`script-src’sha256-nGA...’`). Any inline script whose digest matches the value supplied in the policy will be allowed to execute.

Nonces and hashes can similarly be used with the `style-src` directive to allow the loading of inline stylesheets and external CSS whitelisted via a nonce value.

Listing 2: Locked down policy using a nonce and a hash

```
Content-Security-Policy: script-src ↙
    ↳ ’nonce-BPNLMA4’
’sha256-OPc+f+ieuYDM...’ object-src ’none’;
```

2.2 The threat model of CSP

In order for CSP to offer a security benefit, it must prevent attackers from exploiting flaws that would otherwise enable malicious actions against the application's users. In its current form, CSP offers protections from three types of vulnerabilities A Barth et al., 2014:

- XSS: the ability to inject and execute untrusted scripts in a vulnerable application (protected with the `script-src` and `object-src` directives)
- Clickjacking: forcing users to take unwanted actions in an affected application by overlaying hidden frames on attacker-controlled pages (protected by restricting framing with `frame-ancestors`)
- Mixed content: Accidentally loading resources from insecure protocols on pages delivered over HTTPS (protected with the `upgrade-insecure-requests` and `block-all-mixed-content` keywords and by restricting the loading of scripts and sensitive resources to `https:`).

It follows that only a small subset of CSP directives are useful for XSS protection. Furthermore, the ability to execute malicious scripts in the context of an application subverts the protections offered by all other directives, as discussed in Section 2.2.2.

2.2.1 Benefits of adopting CSP

Since some popular user agents do not yet support CSP or offer only partial supportFoundation, 2016, CSP should only be used as a defense-in-depth to hinder attack attempts in case the primary security mechanism has failed. Accordingly, applications using CSP must also employ traditional protection mechanisms; e.g., employ frameworks with strict contextual escaping for generating markup, use the `X-Frame-Options` header to protect against clickjacking, and ensure that resources on secure pages are fetched over HTTPS.

The actual benefit of setting a Content Security Policy arises only when the *primary* security mechanism has turned out to be insufficient – CSP can help protect users when developers introduce programming mistakes that would otherwise lead to XSS, clickjacking, or mixed content bugs.

In practice, however, clickjacking protection with `X-Frame-Options` is rarely subverted, and active mixed content (scripts and other active content loaded over HTTP from a HTTPS web page) is already blocked by default in modern user agents. Thus, the primary value of CSP – and indeed, the main motivation for the creation of the standard A Barth et al., 2014 – lies in preventing the exploitation of XSS, as it is the only class of vulnerabilities which both can be mitigated by CSP and is commonly inadvertently introduced by developers.

2.2.2 Defending against XSS

The security benefit of CSP is overwhelmingly concentrated in two directives that prevent script execution: `script-src` and `object-src` (plugins such as Adobe Flash can execute JavaScript in the context of their embedding page), or `default-src` in their absence.

An attacker who can inject and execute scripts is able to bypass the restrictions of all other directives. As a result, applications that use a policy without safe `script-src` and `object-src` source lists gain very limited benefit from CSP. For additional directives to provide a meaningful security benefit, the site must first use a safe policy that successfully prevents script execution. In general, non-script directives might serve as a defense against some post-XSS Zalewski, 2011a or "scriptless" Heiderich et al., 2012 attacks, such as exfiltrating data by hijacking form URIs, or phishing by spoofing the page UI using attacker-controlled styles, but they improve security only if CSP is already effective as a protection against XSS.

To achieve the primary goal of preventing unwanted script execution, a policy must meet three requirements:

- The policy must define both the `script-src` and `object-src` directives (or `default-src` in their absence)

Listing 3: CSP bypass due to missing directives

```
<script src="//evil.com"></script>

<object data="//evil.com/evil.swf">
  <param name="allowscriptaccess" ↙
      ↳ value="always">
</object>
```

- The `script-src` source list cannot contain the `unsafe-inline` keyword (unless accompanied by a nonce) or allow `data:` URIs.

Listing 4: Bypass for `unsafe-inline` and `data:` URIs

```
<img src="x" onerror="evil()">

<script ↙
    ↳ src="data:text/javascript,evil()"></script>
```

- The `script-src` and `object-src` source lists cannot contain any endpoints that allow an attacker to control security-relevant parts of the response or contain unsafe libraries.

Listing 5: XSS CSP whitelist bypasses

```
<script ↙
    ↳ src="/api/jsonp?callback=evil"></script>

<script src="angular.js"></script> <div ↙
    ↳ ng-app>
{{ executeEvilCodeInUnsafeSandbox() }} ↙
    ↳ </div>
```

If any of these conditions is not met, the policy is not effective at preventing script execution and consequently offers no protection from content-injection attacks.

We now turn to an analysis of the types of endpoints that, when hosted on a whitelisted origin, allow an attacker to bypass CSP protections against script execution.

2.3 Script execution bypasses

One of the underlying assumptions of CSP is that domains whitelisted in the policy only serve safe content. Hence, an attacker should not be able to inject valid JavaScript in the responses of such whitelisted origins.

In the following subsections, we demonstrate that in practice, modern web applications tend to utilize several patterns that violate this assumption.

2.3.1 JavaScript with user-controlled callbacks

Although many JavaScript resources are static, in some situations a developer may want to dynamically generate parts of a script by allowing a request parameter to set a function to execute when the script is loaded. For example, JSONP interfaces that wrap a JavaScript object in a *callback* function are typically used to allow the loading of API data, by sourcing it as a script from a third-party domain:

Listing 6: Loading JSONP data

```
<script
src="/path/jsonp?callback=alert(document.domain)//">
</script>

/* API response */
alert(document.domain);//{"var": "data", ...});
```

Unfortunately, if a domain whitelisted in the policy contains a JSONP interface, an attacker can use it to execute arbitrary JavaScript functions in the context of a vulnerable page by loading the endpoint as a

`<script>` with an attacker-controlled callback Zalewski, 2011b. If attackers can control the entire beginning of the JSONP response, they gain unconstrained script execution. If the character set is restricted and thus only the function name is controllable, they can use techniques such as SOME Hayak, 2014 which are often qualitatively equivalent to full, unconstrained XSS.

2.3.2 Reflection or symbolic execution

Restrictions on CSP script execution can be (often accidentally) circumvented by a cooperating script in a whitelisted origin. For example, a script can use reflection to look up and invoke a function in the global scope, as depicted in Listing 7.

Listing 7: JavaScript reflection gadget

```
// Can be used to invoke window.* functions with
// arbitrary arguments via markup such as:
// <input id="cmd" value="alert,safe string">
var array =
  document.getElementById(’cmd’).value.split(’,’);
window[array[0]].apply(this, array.slice(1));
```

Such JavaScript gadgets would normally not compromise security, because their arguments are under the control of the developer whose page loads the script. A problem arises when such scripts obtain data by inspecting the DOM, which can be partly attacker-controlled if the application has a markup-injection bug – an attacker can then execute arbitrary functions, possibly with unconstrained arguments, bypassing CSP.

A practical example is the behavior of the popular AngularJS library, which allows the creation of single-page applications with powerful templating syntax and client-side template evaluation (Listing 8).

Listing 8: Bypassing CSP by loading AngularJS

```
<script ↙
    ↳ src="whitelisted.com/angular.js"></script>
<div ng-app>{{ 1000 - 1 }}</div>
```

To achieve its goal, AngularJS parses templates in designated parts of the page and executes them. The ability to control templates parsed by Angular can be considered equivalent to executing arbitrary JavaScript. By default, Angular uses the `eval()` function to evalute sandbox expressions, which is prohibited by CSP policies without the `unsafe-eval` keyword. However, Angular also ships with a “CSP compatibility mode” (`ng-csp`), in which expressions are evaluated by performing symbolic execution, making it possible to call arbitrary JavaScript code despite CSP.

As a consequence, an attacker who can load the Angular library from a domain whitelisted in the CSP can use it as a *JS gadget* to bypass script execution protections. This is possible even if the attacked application doesn’t make use of Angular itself – the only requirement is for the Angular library to be hosted on one of the domains whitelisted in `script-src`. Thus, the mere presence of any Angular library in a trusted domain subverts the protections offered by CSP.

2.3.3 Unexpected JavaScript-parseable responses

For compatibility reasons, web browsers are generally lenient about checking whether the MIME type of a response matches the page context from which the response is used. Any response that can be parsed as JavaScript without syntax errors – and in which attacker-controlled data appears before the first runtime error – can lead to script execution. CSP can thus be bypassed with the following types of responses:

- Comma-separated value (CSV) data with partially attacker-controlled contents:

```
Name,Value
alert(1),234
```

- Error messages echoing request parameters:

```
Error: alert(1)// not found.
```

- User file uploads, even if their contents are properly HTML-escaped or sanitized

Thus, if a whitelisted domain hosts any endpoints with such properties, an attacker can “forge” script responses and execute arbitrary JavaScript. Similar concerns apply to the `object-src` whitelist: if an attacker can upload a resource that will be interpreted as a Flash object to a domain whitelisted for `object-src`, script execution will be possible.

It is important to note that none of the above bypass patterns pose a direct security risk, so developers typically have no reason to change them. However, when an application adopts CSP such endpoints become a security problem because they allow a policy to be bypassed.

More problematically, this issue affects not only the application’s origin, but also all other domains whitelisted in `script-src`. These domains often include trusted third parties and CDNs that might not be aware of CSP – and thus have no reason to identify and fix behaviors that allow CSP bypasses.

2.3.4 Path restrictions as a security mechanism

To address issues about insufficient granularity of domain-based source lists, CSP2 introduced the ability to constrain whitelists to specific paths on a given domain (e.g. `example.org/foo/bar`). Developers now have the option to designate specific directories on a trusted domain for loading scripts and other resources.

Unfortunately, as a result of a privacy concern related to the handling of cross-origin redirects Homakov, 2014, this restriction has been relaxed. If a source list entry contains a redirector (an endpoint returning a `30x` response that points to another location), that redirector can be used to successfully load resources from whitelisted origins even if they do *not* match the path allowed in the policy.

Listing 9: Bypassing CSP path restrictions

```
Content-Security-Policy: script-src example.org
    partially-trusted.org/foo/bar.js

// Allows loading of untrusted resources via:
<script src="//example.org?
    redirect=partially-trusted.org/evil/script.js">
```

Because of this behavior and the prevalence of redirectors in complex web applications (often used in security contexts such as OAuth and to prevent *referer* leaks), path restrictions cannot be relied upon as a security mechanism in CSP.

We have shown how some seemingly benign programming patterns allow a content-injection attacker to bypass script execution protections offered by CSP, and in turn remove any anti-XSS benefit of a policy – its primary focus. We now turn to analyzing the consequences of such bypasses for real-world policies.

3 Empirical Study on CSP

The goal of our work is to investigate the prevalence and protection capabilities offered by CSP in practice. In order to do so, we conducted a large-scale empirical study to collect and analyze real-world CSP policies. In this section, we describe the methodology and the results of this study.

3.1 Research Questions

Our study is divided into two major parts. First, we aim to understand how CSP is currently used; second, we want to analyze the security properties of the deployed policies.

3.1.1 How is CSP used on the web?

As previous research Weissbacher, Lauinger, and Robertson, 2014; Patil and Frederik, 2016 has shown, the CSP adoption rate lags behind the expectations of the security community. Hence, in the first part of our study we aim to shed light on the current state of CSP, in order to understand how widely CSP is used. Furthermore, we'd like to understand whether CSP is used exclusively for XSS protection or whether other prevalent use cases exist. Since many major web applications need to be changed to be compatible with CSP, it is unclear whether CSP policies in the wild are already used for XSS protection, or are in a rather experimental state in which enforcement is still disabled. As such, we are interested in the ratio between policies in enforcing mode and policies in report-only mode. In the second part of this study, we will use the enforced policies to conduct our security analysis.

3.1.2 How secure are real-world CSP policies?

As described in Section 2, there are quite a few pitfalls that might render a policy's protection capabilities ineffective. Avoiding such mistakes in policy creation requires extensive knowledge. In the second part of our study, we aim to identify how many policies are affected by mistakes and thus can be bypassed. We also investigate which kinds of mistakes are the most prevalent.

Additionally, we aim to analyze the security of strict policies, and of whitelists in particular.

3.2 Data set

In order to answer the questions posed above, we used a data set that is representative of the web as a whole: a search index consisting of about 6.5 petabytes of data. The index contains the response headers and bodies of pages on the public Internet crawled within the past 20 days by the Google search crawling infrastructure.

3.3 Methodology

In the following subsections, we outline the methodology used to extract and analyze Content Security Policies from the given data set.

3.3.1 Detecting Content Security Policies

In order to extract CSP policies from the data set, we wrote a MapReduce job. For each URL in the index with a CSP policy, we extracted the following tuple:

$$(URL, CSP, isCSPReportOnly)$$

Based on this list of tuples, we then extracted a set of unique policies for each host, effectively removing duplicate policies on a per-host basis.

3.3.2 Normalizing CSP policies

Several websites automatically generate CSP policies that include random nonces, hashes, or report URIs. In this process, some generation routines randomly switch the order of certain directives or directive values. In order to make the policies in our data set comparable, we first normalized the policies. We implemented a CSP parser as described in the specification[1] and stored a parsed copy of every CSP for later in-depth evaluation. For each of the policies, we applied the following normalization steps:

- First, we removed superfluous white spaces and replaced all variable values, such as nonces and report URIs, with fixed placeholders.
- Second, we ordered and deduplicated all directives and directive values.

3.3.3 Deduplicating CSPs

During our analysis, we noticed that off-the-shelf web applications like message boards and e-commerce

[1] https://www.w3.org/TR/CSP2/#policy-parsing

platforms are spread across many different hosts, while deploying the exact same CSP policy. To address this, we decided to deduplicate the CSPs, based on the normalized policy string. Thus, our final data set contains a single entry for each unique policy that we found across the web.

3.3.4 Identifying XSS-protection policies

As described earlier, CSP supports many directives that are not primarily meant for defending against XSS, such as `img-src` and `frame-ancestors`. Since our study aims to assess the security of a policy in terms of its XSS mitigation capabilities, we needed a way to distinguish policies that attempt to defend against XSS from all other policies. According to our definition, an *XSS-protection* policy must be in enforcing mode and must contain at least one of the following two directives: `script-src` or `default-src`.

3.3.5 Assessing the security of policies

In order to assess whether a CSP policy can be bypassed to execute attacker-controlled scripts, we conduct the following checks:

1. **Usage of `'unsafe-inline'`:** A policy with the `'unsafe-inline'` keyword is inherently insecure if it doesn't also specify a script nonce. Such policies are flagged as bypassable.
2. **Missing `object-src`:** A policy that specifies `script-src` but lacks the `object-src` directive (and does not set `default-src`) allows script execution by injecting plugin resources, as shown in Listing 3.
3. **Use of wildcards in whitelists:** A policy is also insecure if a security-relevant whitelist contains a general wildcard or a URI scheme[2], allowing the inclusion of content from arbitrary hosts.
4. **Unsafe origin in whitelists:** When a domain hosting an endpoint with a CSP bypass is whitelisted, the protective capabilities of CSP are rendered void, as discussed in Section 2.3. In order to assess the security of policies, we compiled a list of hosts with such bypassable endpoints. If a whitelist entry of a given policy appears in this list, we flag the policy as bypassable. In the next section, we outline how we created this list.

3.3.6 Identifying domains with endpoints allowing CSP byasses

In order to identify domains that are insecure for whitelisting in CSP, we extracted pages from the search index that employ one of the practices described in Section 2.2. As noted previously, hosting the AngularJS library and exposing JSONP endpoints are two of many ways to create CSP bypasses.

JSONP endpoints: In order to identify JSONP endpoints, we extracted all URLs from the search index that contain a GET parameter with one of the following names: *callback*, *cb*, *json*, *jsonp*. Subsequently, we verified the resulting data set by changing the value of the corresponding parameter, requesting the resource, and checking whether the changed value was reflected in the beginning of the response. We checked that all endpoints allow full XSS or a SOME attack by verifying the allowed characters in the response. According to our data 39 % of the JSONP bypasses allow arbitrary JS execution while the rest allows arbitrary calls to existing functions via the SOME attack, which in real world applications is considered equally as harmful as full XSS Hayak, 2014.

AngularJS: For the AngularJS library, we created a small signature that matches a specific part of the source code (both minified and non-minified). For each match, we then extracted the version of the file by matching the included version string.

3.4 Results and Analysis

3.4.1 The state of CSP on the web

We used one of Google's indices as our data set for detecting CSP policies. At the time of this analysis, this particular index contained approximately 106 billion unique URLs, spanning 1 billion hostnames and 175 million top private domains.[3] We believe this index is representative of the current state of the web, since all URLs were visited by the Google crawler within a time frame of about 20 days prior to our analysis.

In this data set, we found that 3,913,578,446 (3.7 %) URLs carried a CSP policy. This number, however, is not a good approximation of the CSP adoption rate because applications with large numbers of URLs might be overrepresented within the overall data set. When considering the distribution across domains, the overall picture looks different: only 1,664,019 (0.16 %) of all hostnames across 274,214 top private domains deploy a CSP policy. Out of this list, 1 million hostnames were mapped to one of five e-commerce[4] applications, using only a few distinct policies. To account for this, we deduplicated the data set using the normalized policy. By doing so, we identified 26,011 unique policies.

3.4.2 How CSP is used

CSP's main goal is to protect against XSS attacks. However, it has many other use cases. Hence, as a first step, we sought to determine whether CSP is used for its intended purpose. Figure 1 shows a list

[2] http:, https: or data:

[3] See https://github.com/google/guava/wiki/InternetDomainNameExplained for an explanation of the term "top private domain."

[4] For example, Alibaba mini shops had the same CSP deployed across more than 600,000 hostnames.

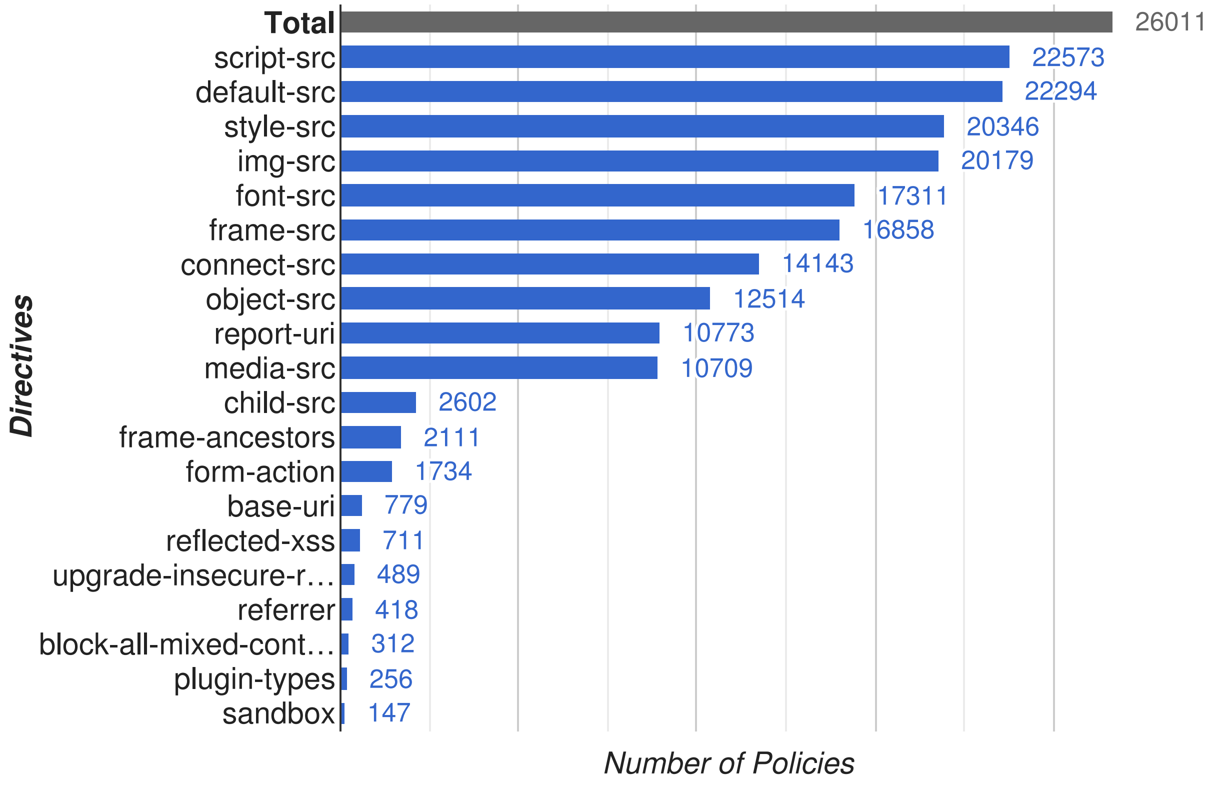

Figure 1: Distribution of CSP directives among unique CSPs

Data Set	Total	Report Only	Bypassable					
			Unsafe Inline	Missing `object -src`	Wildcard in White list	Unsafe Domain	Trivially Bypass-able Total
Unique CSPs	26,011	2,591 9.96%	21,947 84.38%	3,131 12.04%	5,753 22.12%	19,719 75.81%	24,637 94.72%
XSS Policies	22,425	0 0%	19,652 87.63%	2,109 9.4%	4,816 21.48%	17,754 79.17%	21,232 94.68%
Strict XSS Policies	2,437	0 0%	0 0%	348 14.28%	0 0%	1,015 41.65%	1,244 51.05%

Table 2: Security analysis of all CSP data sets, broken down by bypass categories

of all CSP directives, ordered by the number of occurrences. The list clearly shows that the `script-src` and/or `default-src` directive are used in the majority of policies. In contrast, the `frame-ancestors` directive, which can be used to control the framing behavior of a page, is used in only 8.1 % of policies. Furthermore, out of the 26,011 unique policies, only 9.96 % are in report-only mode, whereas the other 90.04 % are switched to enforcing mode. In these numbers, we see clear evidence that CSP is meant as an XSS protection.

3.4.3 Security analysis: overview

The goal of our analysis was to find out whether CSP in its current form can be used to effectively protect from XSS flaws. In order to do so, we compiled three distinct data sets:

1. **All policies:** This data set contains all unique CSP policies, both in report-only and enforcing mode.
2. **XSS-protection policies:** This data set contains all enforcing policies that contain at least one directive for protecting against XSS (`script-src`, `object-src` or `default-src`). This data set excludes all policies for non-XSS-protection use cases.
3. **Strict XSS-protection policies:** Finally, we compiled a set of the strongest CSP policies in the overall data set. These policies are strict in the sense that they do not include any inherently unsafe directive values such as `'unsafe-inline'`, a URI scheme or the general `*` wildcard for whitelisting all hosts.

Table 2 presents the final results. The results for each data set are presented in a single row of the table. In the following sections, we discuss these results in de-

`script-src` value	Usage
`self`	90.95%
`unsafe-inline`	87.26%
`unsafe-eval`	81.65%
Nonce	0.92%
`https:`	3.64%
`http:`	0.85%
`data:`	4.04%
General wildcard	1.18%
Host w. wildcard	69.59%
Host w. path	6.92%
SHA-256 Hash	1.65%
SHA-384 Hash	0.04%
SHA-512 Hash	0.01%

Table 3: Most common features used in `script-src`

tail.

3.4.4 Security of CSP in general

In order to assess the security properties of the detected CSP policies, we automatically applied the checks described in Section 2.2. Based on the analysis of the configuration and whitelist bypassability, we observed that 94.72 % of policies in the overall data set do not offer any protection from XSS. It's important to note that some of these policies are not in enforcing mode or are not used to protect against XSS; however, even for the XSS-protection policies, the percentage of bypassable policies is very similar: 94.68 %.

Unfortunately, most of the policies are inherently insecure. Of the XSS protection policies, 87.63 % employed the `'unsafe-inline'` keyword without specifying a nonce, which essentially disables the protective capabilities of CSP. This surprisingly high number might be explained by the fact that many web applications need to rewrite large parts of their code in order to be compatible with CSP. Some of these pages might still be in a transitional phase, in which they require the `'unsafe-inline'` keyword. Although this problem might be fixed in the long run, many policies contain other obvious problems. For example, we determined that 9.4 % of the policies contain neither the `default-src` nor the `object-src` directive. Hence attackers are able to exploit an XSS vulnerability by injecting a malicious Flash object capable of executing JavaScript. Furthermore, 21.48 % of the policies utilize a general wildcard or a URI scheme (`http:` or `https:`) within the `script-src` or `default-src` directives and thus allow the inclusion of scripts from arbitrary, potentially attacker-controlled hosts.

Given these numbers, it seems that the vast majority of the policies are not capable of effectively protecting against XSS exploits. However, because CSP might be immature, the numbers could be inflated by early adoption issues. In order to account for this fact, we compiled a set of policies that do not contain trivial problems, such as the `'unsafe-inline'` keyword or a general wildcard in the whitelist. In total, we found 2,437 policies that match these criteria. We observed that with our automatic policy analysis tool, we were still able to bypass 51.05 % of these strict policies. Although some of these bypasses were caused by missing `object-src` and `default-src` directives, the majority of bypasses were caused by unsafe origins within the `script-src` whitelist. In the following section, we discuss our analysis of whitelists in detail.

3.4.5 Security of whitelists

For each host within the whitelist the maintainer needs to ensure that an attacker is not capable of injecting malicious content, which could be included via a `<script>` or an `<object>` tag. As described in Section 2.3.1, JSONP endpoints and AngularJS libraries are two of many ways to achieve this. If even just one domain exposes such endpoints, the anti-XSS capabilities of CSP are rendered useless. Hence, the bigger a whitelist gets, the more difficult it is to maintain the security of the corresponding policy.

Figure 2 depicts the number of CSP policies with a specific number of whitelisted domains. At the median, a policy has 12 distinct whitelisted hosts. Also, there is a long tail of policies with a large number of entries. The policy with the longest whitelist, for example, contained 512 hosts.

By querying the index we found 194,908 domains with JSONP endpoints and 101,330 domains hosting the AngularJS library. For each policy within our data set, we then checked whether one of the whitelisted domains was contained in this list. Via this fully automated approach, we found that 41.65 % of all strict policies and 79.17 % of all XSS-protecting policies have insecure whitelists (see Table 4). While these numbers are surprisingly high, they represent only the lower bound. Since many CSP bypasses are application-dependent, it is difficult to fully automate the discovery process. Hence, we believe that the actual number of insecure policies is even higher.

Figure 3 shows that maintaining long whitelists is infeasible in practice. The graph shows how bypassability correlates with the length of a whitelist. While very short whitelists are still quite safe, longer whitelists are much less secure. For example, at the median of 12 entries, we managed to bypass 94.8 % of all policies.

Table 5 lists the top 15 whitelist host entries, ordered by the number of occurrences. The results clearly underline the fact that maintaining whitelists is difficult. Of the top 15 domains, 12 introduce full CSP bypasses, 2 introduce bypasses if combined with `unsafe-eval`, and for only 1 we were not able to find a bypass automatically.

Figure 4 demonstrates that the top 10 domains for whitelist bypasses are sufficient to bypass 68% of all unique CSPs. However, even if JSONP and AngularJS endpoints were removed from the top 10 domains, the

Data Set	Total	*Unsafe domain*	*JSONP Bypass*	*AngularJS Bypass*	*`object-src` Bypass*
XSS Policies	22,425	17,754	17,381	12,617	2,915
Strict XSS Policies	2,437	1,015	968	576	77

Table 4: Number of CSPs that could be bypassed due to JSONP, AngularJS or vulnerable Flash files

remaining hosts would still allow bypassing 66% of observed policies.

As a result of our analysis we conclude that deploying CSP in the traditional whitelist-based model to prevent XSS is not feasible, because in practice the script-execution restrictions can commonly be subverted. In Section 4 we propose a way to solve this problem by crafting CSP policies which replace domain whitelists with script nonces.

4 Improving CSP

In practice, the vast majority of websites currently using CSP deploy a policy that offers no security protections against XSS. Aside from obvious configuration issues (policies with `'unsafe-inline'` and those that do not specify `object-src`), the primary reason for the insecurity of policies is the bypassability of `script-src` whitelists. On the modern web, an approach based on whitelisting domains (even if accompanied with paths) appears to be too inflexible to offer developers security gains and prevent XSS.

At the same time, CSP already offers more granular methods of granting trust to scripts: cryptographic nonces and hashes. In particular, nonces allow the developer to explicitly annotate every trusted script (both inline and external), while prohibiting attacker-injected scripts from executing.

In order to improve the overall security of CSPs in the wild, we thus propose a slightly different way of writing policies. Instead of relying on whitelists, application maintainers should apply a nonce-based protection approach. The following listing depicts a whitelist-based CSP policy and a script satisfying this policy:

```
Content-Security-Policy: script-src example.org

<script ↙
    ↳ src="//example.org/script.js?callback=foo">
</script>
```

Unfortunately, the whitelist of this policy contains an unsafe host and thus the depicted policy is insecure. The attacker could abuse the JSONP endpoint by injecting a script with the following URL: `https://example.org/script?\\callback=malicious\_code`.

In order to avoid this problem, we propose rewriting such policies in the following way:

```
Content-Security-Policy:
    script-src 'nonce-random123'
    default-src 'none'

<script nonce="random123"
  src="https://example.org/script.js?callback=foo">
</script>
```

By using a nonce, scripts can be whitelisted individually. Even if an attacker is capable of finding an XSS, the nonce value is unpredictable, so it is not possible for the attacker to inject a valid script pointing to the JSONP endpoint.

One useful feature of CSP is that it allows for the central enforcement of security decisions. A security team might, for example, use CSP for enforcing a set of trusted hosts from which scripts are allowed to be loaded, instead of relying on the goodwill of developers to not include scripts from untrusted sites. In a single nonce-based policy, however, this is not possible; a resource is only required to adhere to either the whitelist or the nonce. Hence, adding a whitelist to a nonce-based policy removes the benefits of a nonce. Interestingly, browsers allow the enforcement of multiple policies. If two policies are specified for a page, the browser ensures that a resource adheres to both policies. Hence, this feature can be used to get the benefits of both worlds: one nonce-based policy can be used to whitelist individual scripts, while a second whitelist-based policy can be used to centrally enforce security decisions. Two policies can be transferred to the client in the same HTTP response header by separating them with a comma:

```
Content-Security-Policy:
    <!-- whitelist-based CSP -->
    script-src https://example.org
    default-src  https://foobar.org,
    <!-- nonce-based CSP -->
    script-src 'nonce-random123'
```

Another problem with nonce-based policies arises, however, when new scripts are added to the page by JavaScript: because JS libraries might not be aware of CSP and do not know the correct CSP nonce, dynamically inserted scripts would be blocked from executing by CSP, and parts of the application would fail.

To address this problem and to facilitate safe policies without relying on source lists, we propose a new

Count	Percentage	`script-src` value	JSONP Bypass	AngularJS Bypass	Bypassable
8825	33.93%	www.google-analytics.com	yes, if `unsafe-eval`	no	yes, if `unsafe-eval`
7201	27.68%	*.googleapis.com	yes	yes	yes
6307	24.25%	*.google-analytics.com	yes, if `unsafe-eval`	no	yes, if `unsafe-eval`
5817	22.36%	*.google.com	yes	no	yes
5475	21.05%	*.yandex.ru	yes	no	yes
5146	19.78%	*.gstatic.com	no	yes	yes
5076	19.51%	vk.com	yes	no	yes
4728	18.18%	mc.yandex.ru	yes	no	yes
4423	17.00%	yandex.st	no	yes	yes
4189	16.10%	ajax.googleapis.com	yes	yes	yes
3829	14.72%	*.googlesyndication.com	yes	no	yes
3621	13.92%	*.doubleclick.net	yes	no	yes
3617	13.91%	yastatic.net	no	yes	yes
2959	11.38%	connect.facebook.net	no	no	no
2809	10.80%	www.google.com	yes	no	yes

Table 5: Bypassability of the 15 most common whitelisted hosts in `script-src`

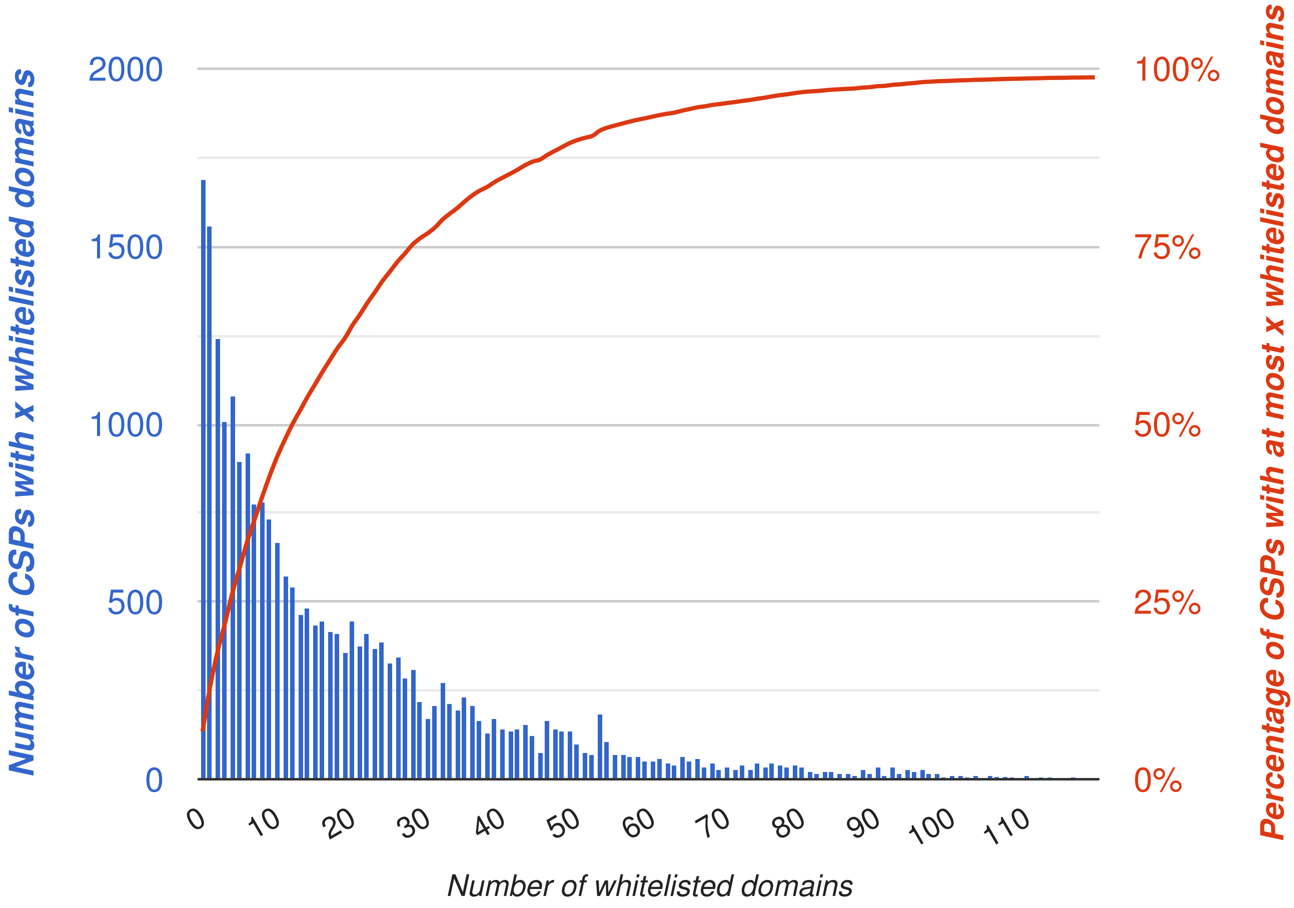

Figure 2: Number of CSPs with a given number of whitelisted domains

source expression for `script-src`: `'strict-dynamic'`. `'strict-dynamic'` is a draft CSP3 specification[5] and is implemented in Chrome and Opera. We describe the adoption process and results in a popular production application in 4.2.

4.1 Propagating trust to dynamic scripts

The addition of the proposed `'strict-dynamic'` keyword to a `script-src` source list has the following consequences:

- Dynamically added scripts are allowed to execute. In practice, this means that script nodes created by `document.createElement('script')` will be allowed by the policy, regardless of whether the URL from which they are loaded is in the `script-src` whitelist.
- Other `script-src` whitelist entries are ignored. The browser will not execute a static or parser-inserted script unless it is accompanied by a valid nonce.

The core observation behind this approach is that scripts added by calling `createElement()` are already trusted by the application – the developer has explicitly chosen to load and execute them. On the other hand, an attacker who has found a markup-injection bug will not be able to directly call `createElement()` without *first* being able to execute JavaScript; and the attacker cannot inject a malicious script and execute JavaScript without knowing the proper nonce defined in the policy.

This mode of using CSP offers the promise of enabling *nonce-based* policies, in which the capability to execute scripts is controlled by the developer by setting nonces on trusted scripts, and allowing trust to propagate to subscripts by setting `'strict-dynamic'`.

As an example, a developer could set a policy similar to the following:

```
Content-Security-Policy:
  script-src 'nonce-random123' 'strict-dynamic';
  object-src 'none';
```

With such a policy, the owner would need to add nonces to static `<script>` elements, but would be assured that only these trusted scripts and their descendants would execute. This mode of deploying CSP can significantly improve the security of a policy and facilitate adoption.

4.2 Case study for 'strict-dynamic'

In February 2015, we adopted a whitelist-based enforcing Content Security Policy in Google Maps Activities, a complex and JavaScript-heavy web application used by 4 million monthly active users. We started

[5] https://www.w3.org/TR/CSP3/#strict-dynamic-usage

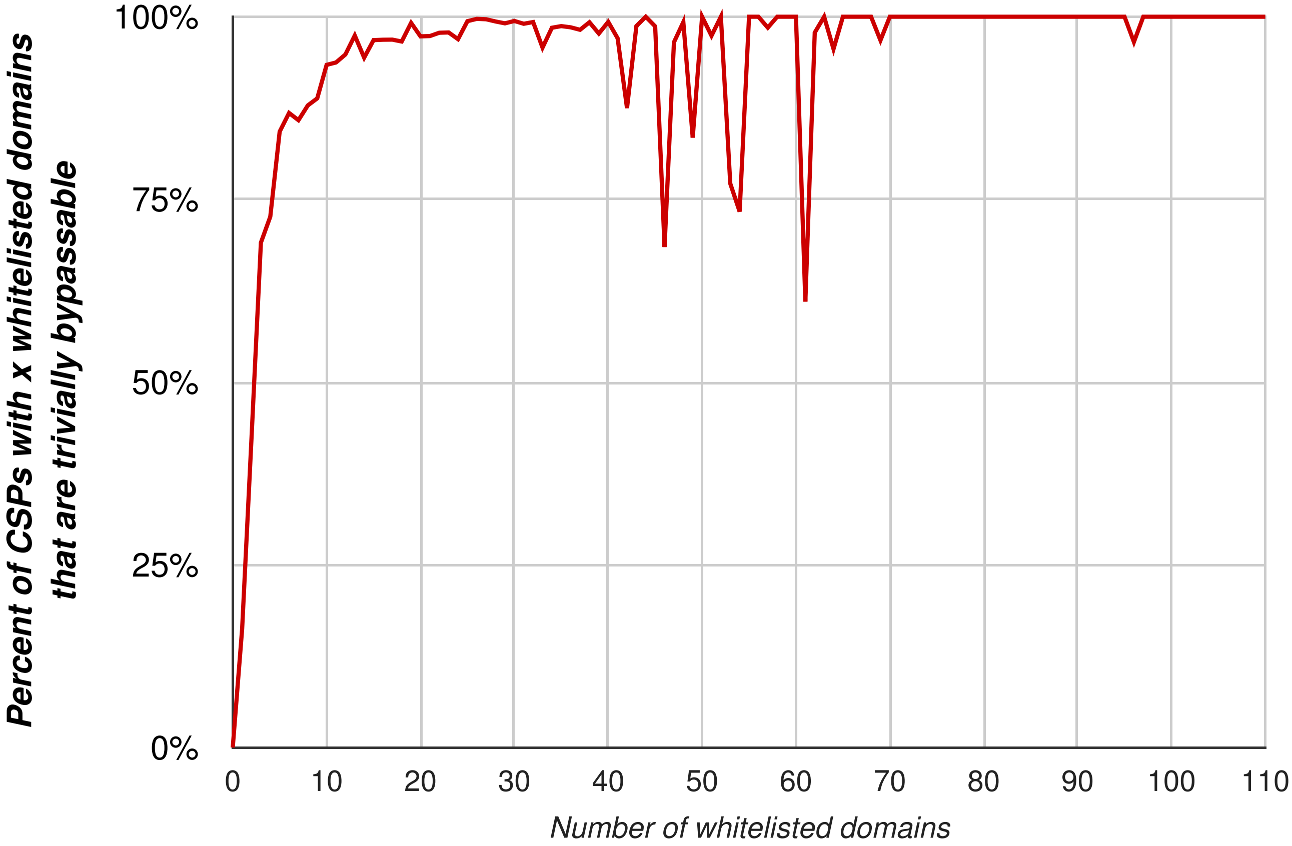

Figure 3: Correlation of whitelist bypasses and number of whitelisted domains

with a simple policy, including a nonce and whole origins, but had to extend it progressively – making 5 major changes throughout 2015 – to cope with changes in the application, APIs and libraries, while keeping the whitelisted paths as secure and restrictive as possible. In order to avoid breakages in production, we had to periodically update origins to reflect changes to the API and the content-serving infrastructure. This led to an explosion in size of the `script-src` whitelist: it grew to 15 long paths, which unfortunately still had to include at least one JSONP endpoint, compromising the effectiveness of the policy in terms of XSS protection.

Since noncing of scripts in the markup was already in place, switching from a whitelist-based approach to a nonce-only policy with `'strict-dynamic'` required no refactoring effort. The switch also allowed us to drastically simplify the policy, avoiding breakages, while at the same time making it more secure and much easier to maintain – in fact, we have not had to make changes to the policy since then.

So far, we have deployed a nonce-only policy with `'strict-dynamic'` with very little effort on Google Photos, Cloud Console, History, Cultural Institute and others.

4.3 Limitations

Nonce-based policies that use `'strict-dynamic'` offer the promise of a more secure and simple-to-deploy CSP, but they are not a panaceum for XSS. Authors will still need to pay attention to both security and compatibility considerations:

4.3.1 Security

- Injections into the `src`-attribute of dynamically created scripts: With `'strict-dynamic'`, if the root cause of an XSS bug is the injection of untrusted data into a URL passed to the `src`-attribute of a script created via the `createElement()` API, the bug will become exploitable, whereas with a whitelist-based policy, the location of the script would be restricted to sources allowed in the policy.
- Injections into a nonced `<script>`: If the injection point is inside a `<script>` trusted by the developer with a nonce, an attacker will be able to execute their malicious script without restrictions. This, however, is still possible with traditional policies.
- Post-XSS/scriptless attacks: Even if a policy prevents an attacker from executing arbitrary scripts in the context of the application, other limited, but also damaging attacks might still be possibleZalewski, 2011a; Heiderich et al., 2012.

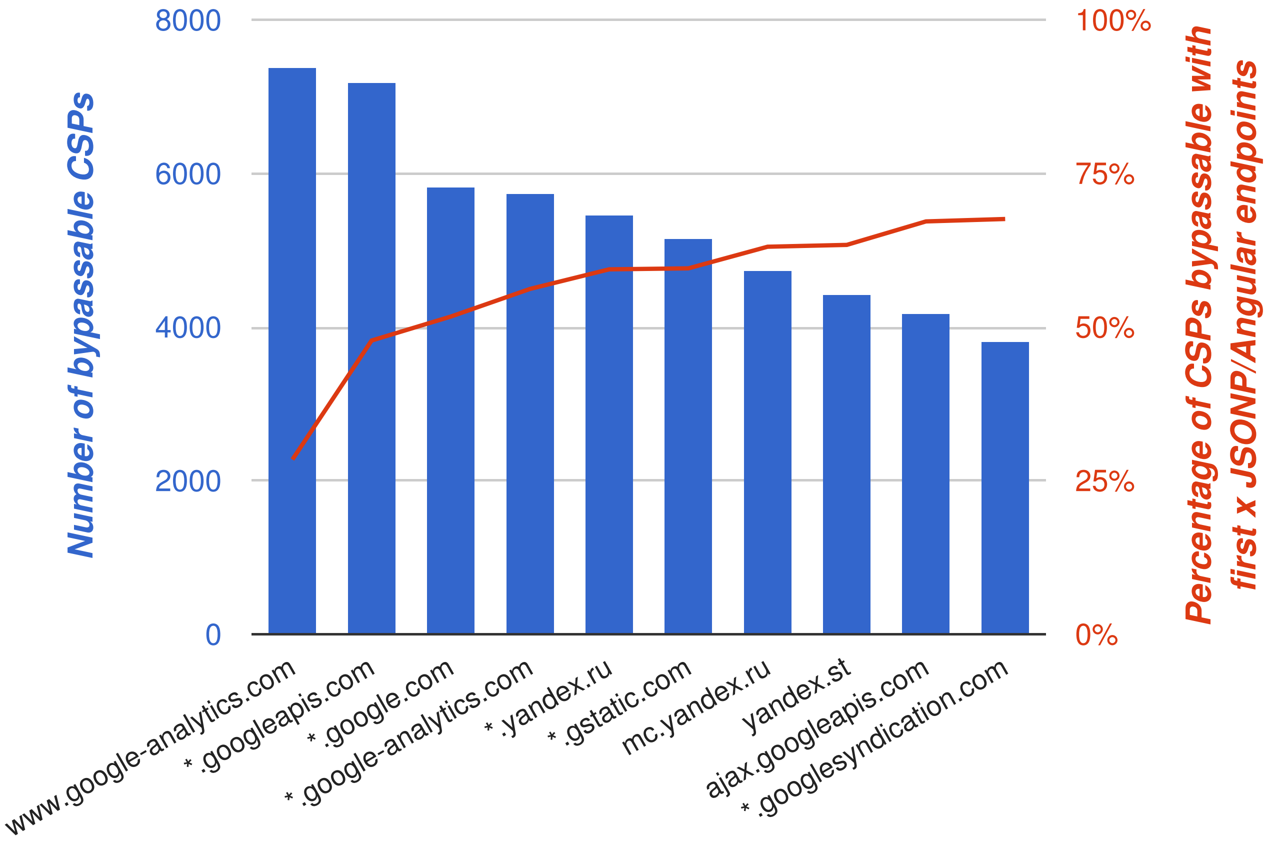

Figure 4: Top 10 script-src host whitelist bypasses + accummulated total bypasses.

4.3.2 Compatibility

- Parser-inserted scripts: If an application uses APIs such as `document.write()` to dynamically add scripts, they will be blocked by `'strict-dynamic'` even if they point to a whitelisted resource. Adopters will have to refactor such code to use another API such as `createElement()`, or explicitly pass a nonce to the `<script>` element created with `document.write()`.
- Inline event handlers: `'strict-dynamic'` does not eliminate the time-consuming process of removing markup incompatible with CSP, such as `javascript:` URIs or inline event handlers. Developers will still need to refactor such patterns before adopting CSP.

Despite these caveats, based on an analysis of hundreds of XSS bugs in a Google-internal data set, we expect that a large majority of XSS will be mitigated using nonce-based policies, and that adopting such policies is significantly easier for developers than the traditional approach based on whitelists.

5 Related work

One of the first papers that proposed whitelisting of scripts to thwart injection attacks was published in 2007 Jim, Swamy, and Hicks, 2007. The system called Browser-Enforced Embedded Policies (BEEP), aims to restrict script inclusion at the browser level based on a policy provided by the application owner. Similar to BEEP, Oda et al. proposed SOMA Oda, Wurster, van Oorschot, and Somayaji, 2008, which extends the idea of BEEP from scripts to other web resources. These ideas were picked up by Stamm et al. who published the initial CSP paper called “Reining in the Web with Content Security Policy” Stamm et al., 2010. Afterward, CSP was picked up by several browser vendors and standardization committees. In 2011, Firefox Sterne, 2011 as well as Chromium Adam Barth, 2011 shipped first experimental prototypes. Subsequently, several iterations of CSP have been standardized and shipped.

Initially, CSP got a lot of attention and many sites started experimenting with it. However, since CSP requires large-scale changes the adoption rate is still small. In 2014, Weissbacher et al. published the first study on the adoption of CSP Weissbacher et al., 2014. In their study, they found that only 1 % out of the top 100 web pages utilized CSP. In order to explore the reasons behind this low adoption rate, they conducted experiments by deploying CSP policies to three distinct sites. Thereby, they found that creating an initial policy is very difficult, because secure policies require extensive changes to existing applications. This problem was investigated by Doupé et al. Their system, named deDacota Doupé et al., 2013, employs automatic code rewriting in order to externalize inline scripts. This in turn enables their system to automatically deploy a CSP policy to the given application.

Kerschbaumer et al. aimed to solve a similar problem. They observed that many pages utilize the insecure `'unsafe-inline'` keyword in order to avoid the rewriting of their applications. Hence, Kerschbaumer et al. created a system to automatically generate CSP policies via a crowd-sourced learning approach Kerschbaumer, Stamm, and Brunthaler, 2016. Over time, their system learned the legitimate scripts observed by multiple users and ensures that only these legitimate scripts are whitelisted within the policy, via script hashes.

Another problem in CSP was investigated by Johns. In his paper Johns, 2014, he addressed the security issues caused by dynamically generated scripts. To counter the threat imposed by JSONP-like endpoints, he proposed not to whitelist scripts based on their origin, but to whitelist scripts based on their checksum; i.e., the script's hash. However, this approach only works for static files, not for dynamic ones such as JSONP. Hence, he proposed a script templating mechanism that allows developers to separate dynamic data values from static code. In this way, a script's hash can be calculated for its static parts, while it is still capable of containing dynamic data values.

Another paper by Hausknecht et al. investigates the tension between browser extensions and CSP Hausknecht, Magazinius, and Sabelfeld, 2015. The authors conducted a large-scale study of browser extensions from the Chrome web store and found that many extensions tamper with the CSP of a page. Hence, they propose an endorsement mechanism that allowed an extension to ask the web page for permission before changing the security policy.

In Section 4, we present a new way of writing CSP policies. Instead of whitelists, we recommend the use of script nonces. The idea of using nonces to prevent XSS has been proposed before. The first paper to do so presented a system called Noncespaces Gundy and Chen, 2009. Noncespaces automatically prepends legitimate HTML tags with a random XML namespace. If an injection vulnerability occurs in the application, the attacker is not capable of predicting this random namespace and thus is not able to inject a valid script tag.

Another system that picked up the idea of instruction set randomization is `xJs` Athanasopoulos et al., 2010. xJS XORs all the legitimate JavaScript code with a secret key that is shared between the server and the browser and is refreshed for each request. Since the browser decrypts the scripts at runtime and the attacker cannot know the secret key, it is impossible to create a valid exploit payload.

6 Conclusion

In this paper, we presented an assessment of the practical security benefits of adopting CSP in real-world applications, based on a large-scale empirical study.

We performed an in-depth analysis of the security model of CSP and identified several cases where seemingly safe policies provided no security improvement. We investigated the adoption of CSP on over 1 billion hostnames, and identified 1.6 million hosts using 26,011 unique policies in the Google search index.

Unfortunately, the majority of these policies are inherently insecure. Via automated checks, we were able to demonstrate that 94.72 % of all policies can be trivially bypassed by an attacker with a markup-injection bug. Furthermore, we analyzed the security properties of whitelists. Thereby, we found that 75.81 % of all policies and 41.65 % of all strict policies contain at least one insecure host within their whitelists. These numbers lead us to the believe that whitelists are impractical for use within CSP policies.

Hence, we proposed a new way of writing policies. Instead of whitelisting entire hosts, we recommend enabling individual scripts via an approach based on CSP nonces.

In order to ease the adoption of *nonce-based* CSP, we furthermore proposed the `'strict-dynamic'` keyword. Once specified within a CSP policy, this keyword enables a mode inside the browser to inherit nonces to dynamic scripts. Hence, if a script trusted with a nonce creates a new script at runtime, this new script will also be considered legitimate. Although this technique departs from the traditional host whitelisting approach of CSP, we consider the usability improvements significant enough to justify its broad adoption. Since this is designed to be an opt-in mechanism, it does not reduce the protective capabilities of CSP by default.

We expect that that the combination of a nonce-based approach and the `'strict-dynamic'` keyword will allow developers and organizations to finally enjoy real security benefits offered by the Content Security Policy.

7 About the Authors

Lukas Weichselbaum
Lukas Weichselbaum is a Senior Information Security Engineer at Google focusing on security enhancements and mitigations for web applications. He coauthored the specification for 'strict-dynamic' in CSP3 and launched CSP-Evaluator, a small tool for developers and security experts to check if a Content Security Policy serves as a strong mitigation against cross-site scripting attacks. Lukas graduated from Vienna University of Technology in Austria where he worked on dynamic analysis of Android malware. He also founded Andrubis - one of the very first large

scale malware analysis platforms for Android applications.

Michele Spagnuolo
Senior Information Security Engineer at Google Switzerland, Michele is a security researcher focused on web application security, and the Rosetta Flash guy. He is also author of BitIodine, a tool for extracting intelligence from the Bitcoin network.

Sebastian Lekies
Sebastian Lekies is a Senior Software Engineer at Google and a PhD Student at the Ruhr-University Bochum. His research interests include client-side Web application security and Web application security scanning. At Google, Sebastian is part of the Security Test Engineering team that develops Google's internal Web application security scanner. Before joining Google, Sebastian was part of SAP's Security Research team, where he conducted academic research in the area of client-side Web application security. Sebastian is regularly speaking at academic and non-academic security conferences such as BlackHat US/EU/Asia, DeepSec, OWASP AppSec EU, Usenix Security, CCS, and many more...

Artur Janc
Artur Janc is an Information Security Engineer at Google, focusing on building and adopting security mechanisms for the web platform. He holds undergraduate degrees in Computer Science and Electrical and Computer Engineering, as well an M.Sc. in Computer Science from Worcester Polytechnic Institute.

References

Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E. P., & Karagiannis, T. (2010). xJS: practical XSS prevention for web application development. In *USENIX conference on Web application development*.

Barth, A. [A], Veditz, D., & West, M. (2014). Content security policy level 2. *W3C Working Draft*.

Barth, A. [Adam]. (2011). Bug 54379 - Add basic parser for Content Security Policy. Retrieved from https://bugs.webkit.org/show_bug.cgi?id=54379

Bates, D., Barth, A., & Jackson, C. (2010). Regular expressions considered harmful in client-side XSS filters. WWW '10. doi:10.1145/1772690.1772701

Bojinov, H., Bursztein, E., & Boneh, D. (2009). XCS: cross channel scripting and its impact on web applications. CCS '09. Chicago, Illinois, USA. doi:10.1145/1653662.1653713

CERT. (2000, February). Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. Retrieved from http://www.cert.org/advisories/CA-2000-02.html

Doupé, A., Cui, W., Jakubowski, M., Peinado, M., Kruegel, C., & Vigna, G. (2013). deDacota: toward preventing server-side XSS via automatic code and data separation. In *CCS'13*.

Foundation, M. (2016). CSP Policy Directives. Retrieved from https://developer.mozilla.org/en-US/docs/Web/Security/CSP/CSP_policy_directives#Browser_compatibility

Gundy, M. V. & Chen, H. (2009). Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *NDSS*.

Hansen, R. & Grossman, J. (2008). Clickjacking.

Hausknecht, D., Magazinius, J., & Sabelfeld, A. (2015). May I?-Content Security Policy Endorsement for Browser Extensions. In *DIMVA'15*.

Hayak, B. (2014). Same Origin Method Execution (SOME): Exploiting A Callback for Same Origin Policy Bypass.

Heiderich, M., Niemietz, M., Schuster, F., Holz, T., & Schwenk, J. (2012). Scriptless attacks: stealing the pie without touching the sill. In *CCS'12*.

Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., & Yang, E. Z. (2013). mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *CCS'13*.

Homakov, E. (2014). Using Content-Security-Policy for Evil. Retrieved from http://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html

Jim, T., Swamy, N., & Hicks, M. (2007). Defeating script injection attacks with browser-enforced embedded policies. In *WWW'07*.

Johns, M. (2014). Script-templates for the Content Security Policy. *Journal of Information Security and Applications*.

Jovanovic, N., Kruegel, C., & Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *S&P'06*.

Kerschbaumer, C., Stamm, S., & Brunthaler, S. (2016). Injecting CSP for Fun and Security.

Klein, A. (2005). DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium Articles 4*.

Lekies, S., Stock, B., & Johns, M. (2013). 25 million flows later: large-scale detection of DOM-based XSS. In *CCS'13*.

Louw, M. T. & Venkatakrishnan, V. (2009). Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy, 2009*. IEEE.

Maone, G. (2017). NoScript. Retrieved from http://noscript.net/

MITRE. (2014, October 1). Common Vulnerabilities and Exposures - The Standard for Information Security Vulnerability Names. Retrieved October 29, 2014, from https://cve.mitre.org/

Nadji, Y., Saxena, P., & Song, D. (2009). Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS*.

Oda, T., Wurster, G., van Oorschot, P. C., & Somayaji, A. (2008). SOMA: Mutual approval for included content in web pages. In *CCS'08*.

Patil, K. & Frederik, B. (2016). A Measurement Study of the Content Security Policy on Real-World

Applications. *International Journal of Network Security*.

Ross, D. (2008). IE 8 XSS filter architecture/implementation. *Blog: http://goo.gl/eOiPsI*.

Saxena, P., Hanna, S., Poosankam, P., & Song, D. (2010). FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*.

Security, W. (2013, May). Website Security Statistics Report. Retrieved from https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf

Stamm, S., Sterne, B., & Markham, G. (2010). Reining in the web with content security policy. In *WWW'10*.

Sterne, B. (2011). Creating a Safer Web with Content Security Policy. Retrieved from https://blog.mozilla.org/security/2011/03/22/creating-a-safer-web-with-content-security-policy/

Stock, B., Lekies, S., Mueller, T., Spiegel, P., & Johns, M. (2014). Precise client-side protection against dom-based cross-site scripting. In *USENIX Security*.

Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., & Vigna, G. (2007). Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*.

Wassermann, G. & Su, Z. (2008). Static detection of cross-site scripting vulnerabilities. In *ICSE'08*.

Weichselbaum, L., Spagnuolo, M., Lekies, S., & Janc, A. (2016). CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Pages 1376–1387). CCS '16. Vienna, Austria: ACM. doi:10.1145/2976749.2978363

Weichselbaum, L., Spagnuolo, M., Lekies, S., & Janc, A. (2017). CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. *Magdeburger Journal zur Sicherheitsforschung*, *14*, 779–796. Retrieved August 10, 2017, from http://www.sicherheitsforschung-magdeburg.de/uploads/journal/MJS_053_Weichselbaum_CSP.pdf

Weissbacher, M., Lauinger, T., & Robertson, W. (2014). Why is csp failing? trends and challenges in csp adoption. In *RAID'14*.

Wichers, D. (2013). OWASP Top-10 2013. *OWASP Foundation, February*.

Zalewski, M. (2011a). Postcards from the post-xss world. *Online at http://lcamtuf.coredump.cx/postxss*.

Zalewski, M. (2011b). The subtle / deadly problem with CSP. *Online at http://goo.gl/sK4w7q*.