CSP Defence Mechanisms Against XSS

Lukas Bonauer, Oussama Kaddami

June 2022

1 Introduction

Isolation is one of the critical aspects in web application security. The same-origin policy (SOP) is a browser mechanism ensuring isolation between different pages. It in particular prevents malicious websites from accessing data related to websites of a different origin. However SOP cannot help against XSS attacks. An attacker can exploit a vulnerable web application to steal sensitive data without violating the SOP. A modern mechanism that can help to detect and mitigate XSS vulnerabilities is the Content Security Policy (CSP)[1]. In this project, we explore the defence mechanisms of CSP against XSS, the security ensured by the policy versus its usability for web developers and we perform an analysis on the most visited websites in regards to their adoption of CSP.

2 Technical foundations & threat model

The Content Security Policy is a declarative mechanism that allows websites administrators to specify a number of security rules on their applications that should be enforced by web browsers on the user's side. CSP has considerably evolved since its creation and currently supports 3 major versions [2]. Browsers support different subsets of its features, which can lead to compatibility problems. However, browsers that do not support some parts of the policy can ignore them, functioning as usual, defaulting to the standard same-origin policy for web content.

The administrator can enable CSP via the HTTP header *Content-Security-Policity* or via an equivalent HTML meta tag. A policy is described using a series of directives separated by semicolons, each describing the policy for a certain resource type [3]. CSP also provides different types of directives unrelated to XSS, for example to prevent click jacking [2]. In this work we focus only on the features that are relevant for XSS prevention. The **default-src** directive establishes the default policy for the resources that do not have a policy of their own [4]. The **script-src** directive restricts the most important type of resource: the execution of JavaScript.

Resource loading restrictions can be specified using whitelists/allow-lists, in which case the browser only executes scripts loaded in source files received from allowed domains, blocking all other scripts including inline scripts and event-handling attributes. It turns out that this approach is hard to set up correctly leading to different misconfigurations, mostly due to the often high number of sources that need to be whitelisted, often allowing attackers to bypass CSP.

An alternative way to restrict script loading is to use nonces. In a nonce-based policy, instead of whitelisting hosts and domains for script execution, the application defines a single use, non guessable token (nonce) delivered both in the CSP policy and as an HTML attribute of legitimate, trusted scripts controlled by the application. The browser only allows the execution of scripts whose nonce matches the specified value in the policy. Since the nonce is randomly changes at each request, the attacker cannot guess it and is thus unable to execute malicious code.

While the two mentioned approaches help to prevent server-side XSS, they are not well-suited to address DOM XSS. Trusted Types is a new feature of CSP that is aimed at preventing DOM XSS. It attempts to make risky HTML injection into DOM sinks secure by forcing the data to be processed according to specific security rules [5]. Implementing Trusted Types requires significant effort of the web developer to consider at each location of the code injection how a given value should reach a sink and how it must be filtered to make sure it is safe.

3 CSP usability versus security

3.1 Whitelist CSP

Restricting script execution to only a fixed set of sources breaks many functionalities in existing applications and it requires a considerable effort from developers to create a whitelist that covers all scripting needs of their website. To make this more usable, the *script-src* directive provides several keywords to allow more granular control over script execution: **unsafe-inline** allows the execution of inline script blocks and JavaScript event handlers and **unsafe-eval** allows the use of JavaScript APIs that execute string data as code, such as eval(). However, while using these keywords makes it easier to make a website compatible with CSP, it makes the policy very ineffective. In particular the use of "unsafe-inline" removes the XSS protection completely as the easiest attack is usually to inject inline code.

Another common misconfiguration is using wildcards or URLs that are not specific enough, such as allowing access to an entire CDN hosting many different scripts. This means the attacker only has to find a vulnerability in any of the whitelisted sources and exploit it. Commonly, this includes abusing a JSONP endpoint [6] or loading a vulnerable old version of a popular library that contains insecure code that can be triggered without inline scripts.

3.2 Nonce-based CSP

In a nonce-based policy, the server must dynamically generate a fresh nonce for every page request and every script must be annotated with this nonce in order to be allowed to execute. This can sometimes be hard to implement, for example for statically served websites.

This approach also poses compatibility problems with dynamically loaded or generated scripts – a common pattern in modern JavaScript code. As a solution, CSP provides a relaxation for *script-src*: The keyword **strict-dynamic** allows scripts that are dynamically inserted by already trusted scripts to be executed even without a nonce, circumventing this compatibility problem. But this relaxation opens other doors for new potential attacks. If the attacker succeeds to leak its input into a trusted script context or trick trusted code into using their input as script code, it will circumvent the CSP.

3.3 Trusted Types

Trusted Types is a feature aimed at preventing DOM XSS. The default policy implemented by trusted types blocks any data flow to the considered risky sinks. Even though this restriction vastly improves security, it poses many usability problems and disables many important functionalities in web applications. The other option is to allow the developer to specify the security policy in a fine grained manner. In this case, implementing Trusted Types requires the effort of the developer to consider why a given value should reach a sink and how to process the data and assert that it is safe. A good practice is to use a sanitizer function before data injection into *innerHTML*. However, this is only secure as long as the attacker does not manage to find a vulnerability within the sanitizer.

4 CSP bypasses due to misconfigurations

We implemented a web application that is intentionally vulnerable to reflected XSS: An attacker can inject arbitrary HTML through a parameter in the URL query. The application then applies various examples of misconfigured CSP to try and prevent its exploitation. In this section we discuss the properties of these insecure policies and show exploits that circumvent their protection.

4.1 No CSP

Without any CSP protection, the attacker can inject any script tag directly without any extra steps:

```
<script>alert("XSS")</script>
```

4.2 Whitelist CSP

```
object-src 'none'; script-src 'self' https://cdnjs.cloudflare.com/
    'unsafe-eval'
```

This configuration disallows inline scripts and requires scripts to be hosted on either the same origin (”self”) or on a particular Content Delivery Network. This is a common practice because JavaScript libraries are often loaded from CDNs for convenience, so often the entire domain of the CDN is whitelisted in the CSP. ”unsafe-eval” is very often added because a lot of JavaScript libraries require the use of eval for their functionality – in this case jQuery.

While the previous attack is no longer possible, this policy can still be exploited by adding a script tag that loads a vulnerable outdated version of a JavaScript library that is hosted on the same CDN:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js
    /1.8.2/angular.min.js"></script>
<div ng-app ng-csp>{{$eval.constructor('alert(\'XSS\')')()}}</div>
```

For this demonstration, the attacker first loads a very old version of AngularJS, and then makes it evaluate an Angular expression that contains a sandbox escape to run the exploiting code. This does not violate the CSP and the XSS succeeds.

4.3 Insecure nonce-based CSP

```
object-src 'none'; script-src 'nonce-{random}' 'strict-dynamic'
```

This configuration applies the more secure nonce-based CSP, which means the attacker cannot include any script tag without knowing the random nonce, no matter whether inline or from an external URL.

However, an attacker can try to abuse the ”base-uri” feature of HTML. This feature allows setting a prefix that should be used to resolve all relative URLs that are used on the page. If the website loads JavaScript files from such relative URLs, an attacker can redirect this to point to a server under their control, where they serve a malicious version of the script:

```
<base href="https://cdn.jsdelivr.net/gh/Bone008/csp-demo/evil/">
```

Our vulnerable example application loads its script from ”./movies.js”, with the server setting the nonce for it. After the attacker injects this base tag, the relative path resolves to the ”evil” directory provided by the attacker.

4.4 Strict nonce-based CSP

```
object-src 'none'; script-src 'nonce-{random}' 'strict-dynamic';
    base-uri 'none'
```

This configuration is the same as the previous one, but now it also applies the ”base-uri” directive to prevent the injection of <base> tags.

This policy is currently the best practice for strict CSP as recommended by Google [7] (we left out some parts that only serve backwards compatibility or reporting purposes). All previously described attacks are mitigated by this policy.

Nevertheless, attackers might still be able to exploit DOM-based injection vulnerabilities, where existing (trusted) JavaScript code can be tricked into indirectly injecting a script from the attacker [8]. One example of such a gadget can be found in the "popup" plugin of the jQuery Mobile library, which our example application makes use of. It can be exploited by providing the exploit payload in the "id" attribute of a popup element:

```
<div data-role=popup id='--!><script>alert("XSS")</script>'></div>
```

This works because the library copies the *id* attribute into an HTML comment upon initialization of the popup. The value is assigned to *innerHTML* without sanitization. The payload is then executed despite having no nonce because it is created by an already trusted script and the "strict-dynamic" directive is present in the CSP.

4.5 Trusted types with strict nonce-based CSP

```
object-src 'none'; script-src 'nonce-{random}' 'strict-dynamic';
    base-uri 'none'; require-trusted-types-for 'script'
```

The final version of our policy additionally enables enforcement of Trusted Types. To implement this, libraries that manipulate DOM must be made compatible by never assigning raw strings to DOM sinks. However, the Trusted Types API also provides a way to set a "default policy", which is executed whenever code assigns to a DOM sink directly. The policy is then called automatically and it can filter the HTML code and pass it through a sanitizer (here we use DOMPurify).

With Trusted Types enabled, the vulnerability in jQuery Mobile from the previous section can no longer be exploited, raising an error in the console instead. Note that currently only Chromium-based browsers support this feature and other browsers still execute the exploit.

5 Code documentation

In order to demonstrate the different levels of CSP and their resilience to XSS attacks, we built an example vulnerable website that could be exploited using server-side reflected XSS or DOM XSS. For the implementation of the web application, we used an Express JS backend server with mustache dynamic templates. The code is available on GitHub[1].

[1] `https://github.com/Bone008/csp-demo`

5.1 Running the server

Using this command, the npm packet manager installs the dependencies specified in the *package.json* in the main directory:

```
npm install
```

The server can be launched from the main directory using:

```
npm run start
```

5.2 Code structure

The code structure is as follows:

- **public:** Contains the public files which include the scripts, images and style files used for rendering on the client side.
- **routes:** Contains the handlers of the different urls, it is the main component of our backend server.
- **views:** Contains the mustache templates.
- **evil:** Contains XSS payloads used for the base-uri exploit demonstration.
- **scanner:** Contains the different scripts used for the CSP analysis of popular websites.

6 CSP scanner

We developed a CSP scanner for the 1000 most visited websites according to DataForSEO [9]. We analyzed the presence and type of CSP that the websites use on their landing page. 27 websites failed to load over HTTPS with a connection error, which leaves 973 results that we evaluated. For each website, we analyzed the value of the Content-Security-Policy header, the Content-Security-Policy-Report-Only header, or an equivalent <meta> tag in the HTML body, and categorized them based on the security mitigations that were applied. The results are shown in Table 1.

CSP category	**count**	**percentage**
(A) none	642	66.0 %
(B) not restricting scripts	178	18.3 %
(C) allowing 'unsafe-inline'	124	12.7 %
(D) whitelist-based	4	0.4 %
(E) nonce-based	25	2.6 %
total:	**973**	**100 %**

Table 1: Types of CSP present in the top 1000 most visited websites

The numbers show that the vast majority of websites do not apply any CSP (category A), or only apply it for purposes other than blocking unwanted script execution (category B). Examples of other purposes include restricting framing using the 'frame-ancestors' directive or to prevent loading content over unencrypted HTTP using the 'upgrade-insecure-requests' directive.

Of those websites that use CSP to restrict script execution in some way (categories C, D and E), 81 % continue to allow 'unsafe-inline' script execution, rendering their XSS defense mostly useless as attackers can inject code using inline script tags or event handlers – the easiest type of attack. 4 websites use an approach based on whitelists without allowing inline scripts.

Only 25 websites in the dataset (2.6 %) use the recommended nonce-based CSP. 19 of these also restrict the 'base-uri' to prevent attackers from redirecting trusted scripts with relative paths by injecting a <base> tag, while 6 of them fail to do so.

Regarding trusted types, only a single website – a marketing site hosted by Google behind the domain doubleclick.net – currently enforces it. This can be explained by the feature being very new and currently only supported by Chromium browsers.

7 Conclusion

In this project we performed a detailed analysis of several ways to apply CSP as a defense-in-depth technique against XSS. We highlighted numerous ways in which inadequate configurations of CSP make it ineffective and leave open attack vectors that can still be exploited under the right circumstances.

However, we also demonstrated how the bypasses that are still possible despite CSP being applied are increasingly complex and rely on additional vulnerabilities in addition to the initial HTML injection point. So applying CSP on a website can still be heavily recommended.

In particular, applying a strict nonce-based CSP mitigates a large percentage of possible attacks and it takes relatively little effort to implement for web developers – especially when used with the *strict-dynamic* relaxation.

Unfortunately, we found that the usage of strict CSP in popular websites is still extremely limited, with only 2 % applying it according to best practices, and a shockingly high percentage explicitly allowing the most dangerous inline scripting.

It is important to keep in mind that CSP must only be considered a last line of defense against XSS, and great focus should be put on preventing unsanitized input from making it into the document in the first place – fixing the root cause in an effective and browser-independent manner. But since it is easy to do this incorrectly and a single code injection point in a complex application can lead to significant data breaches of the entire domain, enforcing a strict CSP across the entire application should be a high priority for web developers.

References

[1] S. Stamm, B. Sterne, and G. Markham, “Reining in the web with content security policy,” in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 921–930. [Online]. Available: https://doi.org/10.1145/1772690.1772784

[2] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1376–1387. [Online]. Available: https://doi.org/10.1145/2976749.2978363

[3] F. Inc. (2012-2021) Content security policy reference. [Online]. Available: https://content-security-policy.com/

[4] M. Foundation. (2022) Content security policy (csp). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[5] K. Kotowicz, “Trusted types-mid 2021 report,” 2021.

[6] S. Benson, “Bypassing csp with jsonp endpoints,” Mar 2022. [Online]. Available: https://hurricanelabs.com/blog/bypassing-csp-with-jsonp-endpoints/

[7] Google, “Strict csp - content security policy.” [Online]. Available: https://csp.withgoogle.com/docs/strict-csp.html

[8] C. Polop. Content security policy (csp) bypass. [Online]. Available: https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass

[9] The dataforseo top 1000 websites. [Online]. Available: https://dataforseo.com/top-1000-websites

Appendix: Contributions

The work on the majority of this project was done by both people. Oussama worked on setting up the demo application and styling it, finding the exploit against the whitelist CSP and setting up the infrastructure for the CSP scanner. Lukas found and implemented the exploits against nonce-based CSP and implemented serving the same page with different CSP versions. Lukas also finished the CSP scanner and evaluated the results.