CSP defence mechanisms against XSS

Oussama KADDAMI - Lukas BONAUER

June 2022

1 Introduction

Isolation is one of the critical aspects in web applications security. Same-origin policy is by default the mechanism ensuring isolation between different frame. It in particular prevents malicious website from executing third party scripts. However SOP cannot help against XSS attacks. An attacker can exploit a vulnerable web application to steal for example sensitive data which is consistent with the SOP. CSP added a layer of security that helps to detect and mitigate certain types of attacks. In this project, we explore the defence mechanisms of CSP, the security ensured by the policy versus the usability and we perform an analysis on the most visited web sites and their inclusion of CSP. [More on the same origin policy]

2 Technical foundations & Threat model

The content security policy is a declarative mechaniqm that allows websites administrators to specify a number of security rules on there applications that should be enforced by web browsers on the user's side. CSP has known lot of (evolving- mofications) since its creation and comprends actuellement 3 versions. On the user side, browsers have different implementation problems which can lead to compatibility problems However, browsers that don't support CSP ignore it, functioning as usual, defaulting to the standard same-origin policy for web content.

The administrator serves Content Security Policy via *Content-Security-Policity* HTTP headers or via a ¡meta¿ HTML object. A policy is described using a series of directives separated by semicolons, each describing the policy for a certain resource type.

illustration

The most important directive is **default-src** which establishes the default policy for the resources that don't have a policy of their own. The consent security policy includes a list of directives that permits to control permission to

different types of resources in a fine-grained manner allowing different types of resources.

CSP provides different types of functionalities. It particularly ensures by different mechanisms **resource loading restrictions**. It ensures Auxilary URL-basd restrictions to prevent click jacking and other hardening options.

Resource loading restrictions can be ensured using whitelists/Allowlists, in which case the browser only execute scripts loaded in source files received from allow listed domains, ignoring 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 mis-configuration, mostly due to compatibility problems, allowing attackers to bypass CSP. The other way that proposes CSP in order ensure this property is to use Nonce based CSP. 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, application controlled scripts. On the user side, the browser only allows the execution of scripts whose nonce matches the specified value in the policy. Since the nonce are pseudo-random, the attacker cannot guess and is thus unable to execute malicious code. However, because of compatibility and usability problems the nonce based CSP should be relaxed using *strict-dynamic*.

While the two detailed approaches are effective in combating server-side XSS, they are not well-suited to address DOM XSS. Trusted types comes to be the approach aimed at preventing DOM XSS. I makes risky injection to sinks secure by processing the data that passes to the sink with specific security rules. 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's safe.

3 CSP Levels: Usability Versus security

3.1 Whitelist CSP:

Whitelist CSP ensures resource loading restrictions by allowing only the execution of scripts loaded in source files received from allow list domains. Additionally, the whitelist CSP blocks inline scripts and event-handling attributes. This aspect, in particular, breaks many functionalities in existing applications and requires a considerable effort from developers to upgrade the code and make it compatible with the CSP policy. But 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. **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(). Using these keywords in the *script-src* policy, due to compatibility

problems with the implementation, makes CSP ineffective or more explicitly removes any CSP protection against XSS.

3.2 Nonce based CSP:

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, application controlled scripts. The non-guessability of the nonce prevents the injected code by the attacker from being executed by the browser. However this approach poses compatibility problems with dynamically loaded or generated scripts. CSP three provides a source expression for *'script-src'* to allow these latter to be correctly executed and circumvent the compatibility problems. This relaxation that 'strict dynamic" brings, opens other doors for new potential attacks. If the attacker succeeds to leak its input in a trusted ¡script¿ context, it will circumvent the security policy of the nonce based 'strict-dynamic' CSP.

3.3 Trusted types:

Trusted types are aimed at preventing DOM XSS by making risky injection to sinks secure by processing the data that passes to the sink with specific security rules. The default policy implemented by trusted types blocks any data flow to the considered risky sinks. Even though this approach is secure, it poses many usability problems and disables many important functionalities in web application. The other option 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's safe. This opens once again other doors for new potential attacks. Using for example a sanitizer function before data injection into innerHTML can be exploited by an attacker if this latter finds how to bypass the sanitizer.

4 CSP miss-configurations & bypasses

CSP adds a layer of security that helps to detect and mitigate certain types of attacks, in particular XSS. However, even if CSP is misconfigured, the attacker still need to find an XSS attack in the application in order to exploit the induced security breach.

There exist many common misconfigurations of CSP that leads to the creation of breaches that could be exploited by an attacker. Those configurations are mostly due to compatibility problems with the application features.

- **Whitelist CSP:**
  As discussed earlier, the main functionality broken by the whitelist CSP is inline scripting and event-handling. Administrator often try to add 'unsafe-inline' and 'unsafe-eval' for usability reasons that outweigh security. Unfortunately, using these keywords in the *script-src* policy, due to compatibility problems with the implementation, makes CSP ineffective or more explicitly removes any CSP protection against XSS. another common misconfiguration is using Wildcards. This enables the attacker to have more ground to find a vulnerability and exploit it. Additionally, the admin should make sure that the attacker cannot load any script from a sub domain of the specified domains in the allowlist, or use a vulnerable old version for a library that enables him to execute malicious scripts.

- **Nonce-Based CSP:**
  Nonce-based CSP provides a strong security defence while preserving most of the functionalities broken by whitelist CSP. However this approach poses compatibility problems with dynamically loaded or generated scripts. CSP 3 provides a source expression for *'script-src'* to allow these latter to be correctly executed and circumvent the compatibility problems. This relaxation that 'strict dynamic" brings, opens other doors for new potential attacks. If the attacker succeeds to leak its input in a trusted ¡script¿ context, it will circumvent the security policy of the nonce based 'strict-dynamic' CSP. for example, If it is possible to pass a user-controlled string to an eval-expression - it is possible to bypass the CSP and execute any inline script. If the page uses vulnerable frameworks like jQuery 2.x (and jQuery 1.x), or symbolic script execution, it is possible to insert and execute any inline script.

- **Trusted types:**
  Trusted types represents an efficient solution to prevent DOM XSS. However, for usability reasons, the administrator should relax the default policy, that block any data flow to risky sinks, by defining what data should be trust. This can create new vulnerabilities. Usually the data passed to sinks is considered safe after being sanitized. However, if an attacker bypasses the sanitizer, he can exploit an XSS vulnerability.

4.1 Our demonstration:

- **CSP 0:** We start our demonstration with a vulnerable application to reflected and DOM XSS with no CSP protection.
  Exploit: include any ¡script¿ tag
- **CSP 1:** whitelist CSP {object-src 'none'; script-src 'self' https://cdnjs.cloudflare.com/ 'unsafe-eval' } **Exploit:** load an old version of Angular from cdnjs.cloudflare.com and execute code through an Angular expression
- **CSP2:** Insecure nonce CSP {object-src 'none'; script-src 'nonce-jM59jjHfbhDWj4je5Z0Omw==' 'strict-dynamic' }
  Exploit: include a ¡base¿ tag to point to a malicious movies.js file
- **CSP3:** Secure nonce CSP {object-src 'none'; script-src 'nonce-6YreS2vB2F1emBtSeN4tEg==' 'strict-dynamic'; base-uri 'none' }
  Exploit: None for server-side XSS, but exploitation of DOM-based XSS is still possible through a gadget in jQuery Mobile
- **CSP4:** Trusted types with secure nonce CSP {require-trusted-types-for 'script'; object-src 'none'; script-src 'nonce-BExZbzu7R38f4ibKChczdw==' 'strict-dynamic'; base-uri 'none' }
  Exploit: Hopefully none.

5 CSP scanner

We developed a csp scanner for the 1000 most visited websites. We are still working on analysis of the results and the conclusion we can withdraw from the study.