Content Security Policy

Extensions developed with WebExtension APIs have a Content Security Policy (CSP) applied to them by default. This restricts the sources from which they can load code such as <script> and disallows potentially unsafe practices such as using `eval()`. This article briefly explains what a CSP is, what the default policy is and what it means for an extension, and how an extension can change the default CSP.

Content Security Policy (CSP) is a mechanism to help prevent websites from inadvertently executing malicious content. A website specifies a CSP using an HTTP header sent from the server. The CSP is mostly concerned with specifying legitimate sources of various types of content, such as scripts or embedded plugins. For example, a website can use it to specify that the browser should only execute JavaScript served from the website itself, and not from any other sources. A CSP can also instruct the browser to disallow potentially unsafe practices, such as the use of `eval()`.

Like websites, extensions can load content from different sources. For example, a browser action's popup is specified as an HTML document, and it can include JavaScript and CSS from different sources, just like a normal web page:

HTML

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
  </head>
  <body>
    <!--Some HTML content here-->
    <!--
      Include a third-party script.
      See also https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.
    -->
    <script
      src="https://code.jquery.com/jquery-2.2.4.js"
      integrity="sha256-iT6Q9iMJYuQiMWNd9lDyBUStIq/8PuOW33aOqmvFpqI="
      crossorigin="anonymous"></script>

    <!-- Include my popup's own script-->
    <script src="popup.js"></script>
  </body>
</html>
```

Compared to a website, extensions have access to additional privileged APIs, so if they are compromised by malicious code, the risks are greater. For this reason:

- a fairly strict content security policy is applied to extensions by default. See default content security policy.
- the extension's author can change the default policy using the `content_security_policy` manifest.json key, but there are restrictions on the policies that are allowed. See `content_security_policy`.

Default content security policy

The default content security policy for extensions using Manifest V2 is:

```
"script-src 'self'; object-src 'self';"
```

While for extensions using Manifest V3, the default content security policy is:

```
"script-src 'self'; upgrade-insecure-requests;"
```

Filter

These policies are applied to any extension that has not explicitly set its own content security policy using the `content_security_policy` manifest.json key. It has the following consequences:

In this article

- You can only load <script> and <object> resources that are local to the extension.
- The extension is not allowed to evaluate strings as JavaScript.
- Inline JavaScript is not executed.
- WebAssembly cannot be used by default.
- Insecure network requests are upgraded in Manifest V3.

Browser extensions

▶ Getting started

▼ Concepts

JavaScript APIs

Content scripts

Background scripts

Match patterns

Work with files

Internationalization

Content Security Policy

Native messaging

Differences between API implementations

Chrome incompatibilities

▶ User interface

▶ How to

▶ JavaScript APIs

▶ Manifest keys

▶ Extension Workshop

Contact us

▶ Channels

Location of script and object resources

Under the default CSP, you can only load code that is local to the extension. The CSP limits `script-src` to secure sources only, which covers `<script>` resources, ES6 modules and web workers. In browsers that support obsolete plugins, the `object-src` directive is also restricted. For more information on object-src in extensions, see the WECG issue Remove object-src from the CSP (at least in MV3)).

For example, consider a line like this in an extension's document:

HTML

```
<script src="https://code.jquery.com/jquery-2.2.4.js"></script>
```

This doesn't load the requested resource: it fails silently, and any object that you expect to be present from the resource is not found. There are two main solutions to this:

- download the resource, package it in your extension, and refer to this version of the resource.
- allow the remote origin you need using the `content_security_policy` key or, in Manifest V3, the `content_scripts` property.

> **Note:** If your modified CSP allows remote script injection, your extension will get rejected from addons.mozilla.org (AMO) during the review. For more information, see details about security best practices .

eval() and friends

Under the default CSP, extensions cannot evaluate strings as JavaScript. This means that the following are not permitted:

JS

```
eval("console.log('some output');");
```

JS

```
setTimeout("alert('Hello World!');", 500);
```

JS

```
const f = new Function("console.log('foo');");
```

Inline JavaScript

Under the default CSP, inline JavaScript is not executed. This disallows both JavaScript placed directly in `<script>` tags and inline event handlers, meaning that the following are not permitted:

HTML

```
<script>
  console.log("foo");
</script>
```

HTML

```
<div onclick="console.log('click')">Click me!</div>
```

If you are currently using code like `<body onload="main()">` to run your script when the page has loaded, listen for DOMContentLoaded or load instead.

WebAssembly

Extensions wishing to use WebAssembly require `'wasm-unsafe-eval'` to be specified in the `script-src` directive.

From Firefox 102 and Chrome 103, `'wasm-unsafe-eval'` can be included in the `content_security_policy` manifest.json key to enable the use of WebAssembly in extensions.

Manifest V2 extensions in Firefox can use WebAssembly without `'wasm-unsafe-eval'` in their CSP for backward compatibility. However, this behavior isn't guaranteed, see Firefox bug 1770909 . Extensions using WebAssembly are therefore encouraged to declare `'wasm-unsafe-eval'` in their CSP.

For Chrome, extensions cannot use WebAssembly in version 101 or earlier. In 102, extensions can use WebAssembly (the same behavior as Firefox 101 and earlier). From version 103, extensions can use WebAssembly if they include `'wasm-unsafe-eval'` in the `content_security_policy` in the manifest key.

Upgrade insecure network requests in Manifest V3

Extensions should use `https:` and `wss:` when communicating with external servers. To encourage this as the standard behavior, the default Manifest V3 CSP includes the `upgrade-insecure-requests` directive. This directive automatically upgrades network requests to `http:` to use `https:` .

While requests are automatically upgraded, it is still recommended to use `https:` -URLs in the extension's source code where possible. In particular, entries in the `host_permissions` section of manifest.json should start with `https://` or `*://` instead of only `http://` .

Manifest V3 Extensions that need to make `http:` or `ws:` requests can opt out of this behavior by overriding the default CSP using the `content_security_policy` manifest.json key with a policy that excludes the `upgrade-insecure-requests` directive. However, to comply with the security requirements of the Add-on Policies, all user data must be transmitted securely.

CSP for content scripts

In Manifest V2, content scripts have no CSP. As of Manifest V3, content scripts share the default CSP as extensions. It is currently not possible to specify a separate CSP for content scripts (source).

The extent to which the CSP controls loads from content scripts varies by browser. In Firefox, JavaScript features such as eval are restricted by the extension CSP. Generally, most DOM-based APIs are subjected to the CSP of the web page. In Chrome, many DOM APIs are covered by the extension CSP instead of the web page's CSP (crbug 896041).

- Edit the page on GitHub.
- Report the content issue.
- View the source on GitHub.

Want to get more involved? Learn how to contribute.

This page was last modified on Dec 8, 2023 by MDN contributors.