Engineer

Content Security Policy

a list of where a page may load things from

What it is

Content Security Policy is a set of rules a website hands to the browser about what its pages may run, fetch and submit, which the browser then enforces on the site's behalf. It is almost always shortened to CSP. The site sends the rules in a `Content-Security-Policy` response header, or writes them into the page in a `meta` element, and from that moment any script, stylesheet, image, font, frame or connection the rules do not allow is refused before it loads.

A policy is a short string of directives separated by semicolons. Each directive names one kind of resource or behaviour and the sources it accepts. Shown one directive per line for reading, though on the wire it is a single header value, a strict policy looks like this:

```
default-src 'self';
script-src 'nonce-q8Zt3VwR1mKx7NpL2cYfJA==' 'strict-dynamic';
object-src 'none';
base-uri 'none';
form-action 'self';
frame-ancestors 'none'
```

The vocabulary is small. `'self'` means the page's own origin. A host such as `https://static.example.com` or a scheme such as `https:` widens that. `'none'` allows nothing. A nonce is a random value the server generates for one response and writes both into the policy and onto each script element it vouches for. A hash is the digest of an inline script's exact text, so that one script and no other may run. Two keywords, `'unsafe-inline'` and `'unsafe-eval'`, switch protections back off, and their names are a fair warning.

`default-src` is the fallback for the fetch directives a policy leaves out, such as `script-src`, `style-src`, `img-src` and `connect-src`. It is a fallback for those alone. Directives that govern behaviour rather than fetching, among them `base-uri`, `form-action` and `frame-ancestors`, stay unrestricted unless they are written out.

A policy can also run in a watching mode. Sent as `Content-Security-Policy-Report-Only`, it blocks nothing and instead reports each violation to an endpoint named with `report-to`, which uses the browser's Reporting API. The older `report-uri` directive is deprecated in its favour. The current text of the standard is Content Security Policy Level 3, a W3C Working Draft from the Web Application Security Working Group.

Use the concept in a decision

Put "What is Content Security Policy?" to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

The attack CSP was built against is cross-site scripting. Somebody gets text of their choosing into a page, through a comment field, a search term echoed back into the HTML or a compromised dependency, and the browser runs it as script with all the authority the site's own code has: it can read the page, act as the logged-in user and send what it finds anywhere. The first defence is escaping output correctly, and that has to be right in every template, on every line, every time. CSP is the second defence. When an escaping mistake does slip through, the injected script still has to get past the policy, and a policy that only runs scripts carrying this response's nonce turns it away.

It also limits what a script that does run can reach. `connect-src` decides which hosts script may talk to, which narrows where a skimmer can post what it collects. `frame-ancestors` decides which sites may show the page inside a frame, the standard defence against clickjacking, where a page is overlaid on another so

that a click lands somewhere the user never meant. `form-action` decides where forms may submit, and `base-uri` stops an injected `base` element from quietly redirecting every relative address on the page.

It is worth being exact about who CSP protects. It runs in the browser, so it guards the person reading the page from content an attacker planted there. It does nothing for the server against that same person, who can edit or strip the header in their own browser in seconds. A policy is a reason to trust less of the page, never a reason to trust more of the request.

The cost is mostly inventory. A policy forces a team to know every origin its pages load from, every inline script and every event handler written as an attribute. A new site can be built to a strict policy from its first commit for very little. An older site with inline handlers, a tag manager and several widgets pays for the clean-up first, and that clean-up is usually the real project.

What it looks like in practice

This site holds itself to one. The security page lists a strict Content Security Policy, declared in the page itself, with `form-action 'none'`, because the site posts no forms anywhere. Every HTML page states `script-src 'none'`: no executable JavaScript or service worker runs. The site makes no third-party requests; everything a page needs comes from its own host, so there is very little to allow.

An application with logged-in users needs more than a short allowlist. On a platform holding professional and organisational data, the Next.js proxy middleware sets a Content-Security-Policy with a per-request nonce and `strict-dynamic`, alongside HSTS and SameSite cookies. A fresh nonce on every response means that only the script elements the server stamped on that response will run. With `strict-dynamic` in the policy, a script the nonce trusts may load further scripts itself, so the policy does not have to name every host a bundle fetches its chunks from. That platform works from the assumption that the client is hostile. Each request's entitlements are therefore checked again by the server, and the gates in the frontend count for user experience only. CSP is one layer of several there, protecting users from injected code, and authorisation is decided elsewhere.

Where it goes wrong

The policy goes live and half the site stops working. A strict policy sent straight to enforcement blocks the inline script nobody remembered and the tracking tag someone added two years ago. Ship it as `Content-Security-Policy-Report-Only` first, read the reports until they settle, allow or remove each source they name, and only then enforce. The report-only header has to be a real header: the specification does not support it in a `meta` element.

The allowlist trusts a whole host. `script-src https://cdn.example.com` allows every file on that host, including old library versions and endpoints that echo a caller's input back as script. The specification itself says that deploying a policy that works against cross-site scripting is a challenge, and recommends what it names a Strict CSP: `script-src` built from nonces or hashes with `'strict-dynamic'`, plus `base-uri` set to `'self'` or `'none'`.

`'unsafe-inline'` **is assumed to allow every inline script.** Once a nonce or a hash appears in the same source list, browsers that understand them ignore `'unsafe-inline'`. The rule lets a policy keep `'unsafe-inline'` as a fallback for browsers too old to know nonces while current ones apply the strict list, and it surprises a team that adds a nonce to one script and finds its other inline scripts blocked.

The nonce stops being a secret. The specification requires a new value every time a policy is sent and asks for at least 128 bits from a cryptographically secure generator. A nonce hard-coded into a template, or a whole page with its nonce stored in a shared cache and served to everyone, gives an attacker the value they need to stamp their own script.

A policy in a `meta` **element does less than the same policy in a header.** The browser ignores `frame-ancestors`, `report-uri` and `sandbox` when they arrive that way, and the policy only governs what comes after the element in the document, so anything above it in the head loads unchecked. A site with no say over its response headers still gets real protection from a `meta` policy, placed as early in the head as it will go, but anti-framing needs the header.

A second policy tightens instead of loosening. A response can carry several policies, from a framework, a proxy and a CDN, and a resource has to pass every one of them. Adding a policy can only restrict, so a permissive header added downstream to fix a blocked request does nothing, and the fix has to go into the policy that blocked it.

Further reading

- Content Security Policy Level 3 — the standard itself, with the Strict CSP recommendation and a section on how nonces get stolen.
- HTML Standard: the meta element — the pragma that applies a policy declared in the page, the directives it drops, and why resources fetched before it may slip past.
- Reporting API — the endpoint mechanism behind `report-to`, for reading violations before enforcing.
- Trusted Types — the companion standard that a `require-trusted-types-for` directive switches on, aimed at script injected through the DOM.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is Content Security Policy?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/csp/