

HTTP Security Headers and How They Work

A technical guide to HTTP security headers:
current recommendations, deprecated headers
to remove, and the configurations that actually
protect your applications in modern browsers.

Table of contents

Introduction	3
HTTP security headers at a glance	4
X-Frame-Options header	6
X-XSS-Protection header	8
X-Content-Type-Options header	10
X-Download-Options header	11
Content Security Policy (CSP) header	12
HTTP Strict Transport Security (HSTS)	17
HTTP Public Key Pinning (HPKP)	19
Expect-CT header	20
Referrer-Policy header	21
Permissions-Policy header	24
Cross-origin isolation headers: COOP, COEP, and CORP	25
Reporting header policy violations: Reporting-Endpoints and report-to	27
Common implementation pitfalls for security headers	28
Validating security headers in practice	28
Conclusion: Set the right headers for proactive application security	29

Introduction

Headers are part of the HTTP specification and are used to define the metadata of the message in both the HTTP request and response. While the HTTP message body is often meant to be read by the user, metadata is processed by clients, servers, proxies, and caches as part of the HTTP exchange, and has been included in the HTTP protocol since version 1.0.

In request messages, the header metadata can hold information that includes:

- Language of the request
- Cookies
- Credentials for the website
- Cache data

In response messages, the metadata can hold information that includes:

- Size and type of the content
- Cache storage preferences
- Server data
- Time and date
- Credentials to be set by the client

HTTP security headers are HTTP response headers that instruct browsers to enforce specific security behaviors. While some headers may vary by application or resource type, security-critical headers should be applied consistently wherever they are intended to provide protection.

Individual headers are usually straightforward to add, but maintaining correct and consistent configurations across modern applications – where environments, proxies, and third-party dependencies all interact – is far more challenging. Headers can be missing, misconfigured, or overridden as applications evolve, especially across multiple environments and APIs. Relying on the presence of security headers without verifying their real-world behavior can also lead to a false sense of security, especially when browser support has changed. In practice, this makes ongoing verification just as important as initial configuration.

This whitepaper explains how HTTP headers can be used in relation to web application security. It highlights the most commonly used HTTP security headers and explains how each of them works in technical detail – including several that are now deprecated, so you know what to remove as well as what to add.

HTTP security headers at a glance

Header	Status	Protects against	Recommended value
Content Security Policy	Current	XSS, clickjacking, injection attacks, unauthorized resource loading	<code>default-src 'self'; script-src 'nonce-{random}' 'strict-dynamic'; object-src 'none'; base-uri 'none'</code>
HTTP Strict Transport Security	Current	SSL stripping, protocol downgrade attacks, insecure connections	<code>max-age=31536000; includeSubDomains; preload</code>
X-Frame-Options	Current (legacy fallback)	Clickjacking, UI redressing	DENY — send alongside CSP frame-ancestors
X-Content-Type-Options	Current	MIME sniffing, content type confusion attacks	<code>nosniff</code>
Referrer-Policy	Current	Sensitive URL leakage to third parties	<code>strict-origin-when-cross-origin</code>
Permissions-Policy	Current	Unauthorized access to camera, microphone, geolocation, and other browser APIs	Disable unused features: <code>geolocation=(), camera=(), microphone=(), payment=(), usb=()</code>
Cross-Origin-Opener-Policy	Current	Cross-window reference attacks, Spectre-adjacent timing attacks	<code>same-origin</code>

Header	Status	Protects against	Recommended value
Cross-Origin-Embedder-Policy	Current	Side-channel attacks via shared memory; required for cross-origin isolation	require-corp (or credentialless for easier third-party compatibility)
Cross-Origin-Resource-Policy	Current	Unauthorized cross-origin loading of your resources	same-origin for sensitive resources
Reporting-Endpoints + report-to	Current	Not a security control but infrastructure for receiving violation reports from other headers	Reporting-Endpoints: csp-endpoint="https://example.com/reports"
X-XSS-Protection	Deprecated	Reflected XSS (browser filter) – Removed from browsers	Send 0 or omit; use CSP instead
X-Download-Options	Deprecated	IE file execution in site context – Internet Explorer only, retired 2022	Remove; use Content-Disposition: attachment instead
HTTP Public Key Pinning	Deprecated	Fraudulent certificate acceptance – Removed from browsers	Remove; Certificate Transparency + CAA DNS records now provide this protection
Expect-CT	Deprecated	Certificate Transparency opt-in — CT is now enforced by default	Remove; no replacement header needed

X-Frame-Options header

The X-Frame-Options header was introduced to prevent UI redressing attacks, beginning with clickjacking in 2009. It tells the browser whether a page is permitted to be loaded inside a frame, iframe, object, or embed element.

In general, UI redressing attacks work by loading a target page inside an iframe and overlaying it with other interface elements, tricking users into interacting with the hidden page beneath. Modern UI redressing attacks are not limited to transparent iframes and may also rely on overlays, visual deception, or interface manipulation techniques that trick users into interacting with embedded content.

How to prevent clickjacking attacks

A clickjacking attack is a classic variety of UI redressing that loads a malicious website inside a low-opacity iframe so the victim cannot see it. The victim only sees an innocuous-looking button, checkbox, or link in the foreground – but clicking it actually triggers an action on the embedded website beneath. Because the interaction happens in the victim's authenticated browser session, it carries their cookies and credentials, too.

The most reliable defense against clickjacking is to prevent your page from being framed by sites you don't trust. The modern way to do this is the Content Security Policy frame-ancestors directive ([see the CSP section below](#)). X-Frame-Options is its predecessor, but still worth sending as a fallback for legacy clients that may not support CSP.

Recommended X-Frame-Options configuration

```
Content-Security-Policy: frame-ancestors 'none';  
X-Frame-Options: DENY
```

Send both headers together and ensure they are consistently returned in responses across all application endpoints. Modern browsers generally prioritize the CSP frame-ancestors directive when both headers are present, while X-Frame-Options remains useful as a fallback for older clients.

X-Frame-Options directives

Directive	Description
DENY	The page must not be embedded in any frame or iframe on any site, including your own.
SAMEORIGIN	The page can only be framed by a page with the same scheme, hostname, and port.

Note on ALLOW-FROM: A third directive, ALLOW-FROM URL, was defined in early specifications to allow framing by a single named origin. It was never supported by Chromium-based browsers and is not supported by current browsers. Do not use it. The CSP frame-ancestors directive is its modern, widely supported replacement – and unlike ALLOW-FROM, it accepts multiple origins.

Why CSP frame-ancestors is the right long-term answer

With frame-ancestors you can express policies that X-Frame-Options cannot:

```
Content-Security-Policy: frame-ancestors 'self'  
https://trusted.example.com;
```

This allows your own origin and one specific external origin to embed your page, and nothing else. X-Frame-Options has no equivalent. For further details on same-origin concepts, see our [Same-Origin Policy whitepaper](#).

Verification tip: When both **X-Frame-Options** and **CSP frame-ancestors** are present, modern browsers prioritize the CSP directive. In practice, you should confirm that framing is blocked according to the CSP policy, not just the legacy header, especially when supporting a mix of client environments.

X-XSS-Protection header

Deprecated. The browser feature this header controlled no longer exists in any current browser. If you are currently sending X-XSS-Protection: 1, you should change it as described below.

X-XSS-Protection was designed to control the built-in reflected XSS filter – sometimes called the XSS Auditor – that some browsers once shipped. The idea was to detect dangerous HTML patterns appearing in both the request URL and the response body, and either strip or block them.

That filter is long gone because it proved ineffective in practice. Chrome removed it in 2019, Microsoft Edge dropped it when it moved to the Chromium engine, and Firefox never implemented it at all. No current browser provides meaningful protection through this header.

What attackers could do with a reflected XSS vulnerability

The underlying threat is still real, even if this header no longer addresses it. [Reflected XSS](#) arises when user-supplied input is evaluated as code in the page context.

Consider this PHP snippet:

```
<p>Welcome <?php echo $_GET["name"];?></p>
```

If you pass the following value of the name parameter, the application embeds it unfiltered in the page, triggering a JavaScript alert – or in a real attack, something far more damaging:

```
http://www.example.com?name=<script>alert(1);</script>
```

Attackers exploiting reflected XSS can steal cookies, capture keystrokes, issue requests as the victim, or hijack the session entirely.

Why the old header made things worse

The XSS Auditor was not only ineffective but also introduced its own vulnerabilities. Researchers demonstrated repeatedly that attackers could weaponize the filter to disable legitimate scripts, including frame-busting defenses. The idea was to inject a crafted URL that causes the filter to strip a legitimate script rather than a malicious one:

```
<iframe src="http://www.victim.com/?v=<script>if">
```

This causes the filter to remove a fragment of a legitimate script, thus breaking it – and in this case, potentially disabling a frame-busting guard. The protection mechanism became the attack surface, which is ultimately why browser vendors removed it rather than patching it.

Recommended X-XSS-Protection configuration

Send `X-XSS-Protection: 0` to explicitly disable any residual legacy behavior in the handful of older clients that might still act on this header. Omitting the header entirely is also acceptable.

Protection against reflected XSS now comes from:

- A properly configured Content Security Policy, particularly `script-src` with nonces or hashes
- Output encoding and escaping at the application layer
- Framework-level sanitization

Do not rely on the `X-XSS-Protection` header for any security guarantee.

X-Content-Type-Options header

This header controls MIME type sniffing – a content-evaluation behavior browsers use when the Content-Type of a response is missing or ambiguous. Without instruction, a browser will inspect the response body and attempt to render it in whatever format looks most appropriate. That automatic behavior can be exploited.

Why MIME sniffing is a risk

Suppose your application lets users upload text files. If a user uploads an HTML file containing JavaScript and your server returns it without specifying a Content-Type, the browser may sniff it as `text/html` and execute the embedded script – even if you intended it to be served as plain text.

In general, attackers may upload files that contain scripts or other potentially malicious content while masquerading as images or other safe file types. If the server returns the file with an incorrect or missing Content-Type, browser content sniffing may cause the file to be interpreted as executable rather than the content type intended by the developer.

Recommended X-Content-Type-Options configuration

```
X-Content-Type-Options: nosniff
```

This is the only available value. It instructs the browser to trust the Content-Type your server declares and refuse to sniff. Always pair it with correctly set Content-Type headers on your responses – `nosniff` makes the browser strict about what you declare, so what you declare needs to be right.

Verification tip: Ensure this header is returned consistently across all responses, including error pages and API endpoints.

X-Download-Options header

Deprecated – Internet Explorer only. This header only ever worked in Internet Explorer 8 and later versions of IE. Internet Explorer was retired in June 2022 and has no meaningful user base. If you are currently sending this header, it is safe to remove it.

`X-Download-Options: noopen` removed the “Open” button from IE’s file download dialog, forcing users to save a file before opening it. The intent was to prevent a downloaded file from running in the context of the serving website – a defense against attackers uploading malicious HTML that could then steal cookies from the same domain.

What to use instead

The file upload attack class this header targeted is still relevant. Current defenses include:

- `Content-Disposition: attachment` – instructs browsers to download the file rather than render it inline
- `X-Content-Type-Options: nosniff` – prevents browsers from second-guessing the declared content type
- Serving user-uploaded content from a separate, sandboxed origin – the strongest defense, because even if malicious code runs, it has no access to the main site’s cookies or storage

Content Security Policy (CSP) header

Content Security Policy is the most powerful client-side security header available. It gives you fine-grained control over every resource type your page is allowed to load – scripts, styles, images, fonts, frames, form targets, and more. It is the primary defense against [cross-site scripting \(XSS\)](#), clickjacking, protocol downgrading, and several injection attack classes.

Keep in mind that defining a CSP is only the first step – its effectiveness depends on how strictly it is enforced in actual browser responses.

Here is the basic CSP syntax:

```
Content-Security-Policy: <directive1> <value1 value2...>;  
<directive2>...
```

How the allowlist model works

The simplest CSP approach uses origin allowlists, where you declare which hosts are trusted sources for each resource type. For example:

```
Content-Security-Policy: script-src 'self' https://apis.google.com
```

This permits scripts only from your own origin and `https://apis.google.com`. Any inline scripts or scripts from other hosts are blocked.

Most fetch directives fall back to the value of `default-src` if they are not explicitly set. Setting `default-src 'self'` locks down everything to same-origin by default, after which you can open up individual resource types as needed.

```
Content-Security-Policy:  
  default-src 'self';  
  img-src 'self' https://cdn.example.com;  
  font-src https://fonts.googleapis.com
```

Note that `default-src` does not override `base-uri`, `form-action`, `frame-ancestors`, or `sandbox` – set those explicitly.

Large host allowlists often become difficult to maintain and can undermine CSP effectiveness. Modern CSP deployments should favor nonce-based policies with `strict-dynamic` wherever practical.

Writing a CSP that actually stops XSS: nonces, hashes, and strict-dynamic

An allowlist-based `script-src` is a good start but has a known weakness: any script hosted on an allowlisted CDN – even one with hundreds of unrelated JavaScript files – can become an XSS vector. Maintaining a host allowlist also becomes an operational burden as third-party dependencies change.

A more robust model uses nonces or hashes combined with `strict-dynamic`:

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

How this works:

- The server generates a fresh cryptographic random value (nonce) for every response and inserts it into every `<script>` tag, as in: `<script nonce="abc123">`.
- The browser only executes scripts whose nonce attribute matches the one in the CSP header.
- `strict-dynamic` propagates trust to scripts loaded by a nonced script, so dynamically injected scripts from trusted loaders don't need to be explicitly allowlisted.
- `object-src 'none'` and `base-uri 'none'` close common CSP bypass routes via plugins and base tag injection.

An attacker-injected `<script>` has no valid nonce and is blocked, regardless of which host it is served from.

Adding trusted types takes this further by defending against DOM-based XSS:

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

Using `require-trusted-types-for 'script'` forces dangerous DOM sinks – `innerHTML`, `eval()`, `document.write()`, and others – to only accept values that have passed through a Trusted Types policy. This makes DOM XSS exploitable only if an attacker can also subvert your sanitization logic, which is a much harder task.

Deploying safely with Report-Only mode

Never enforce a new CSP without testing it first, or you risk breaking legitimate functionality. Use `Content-Security-Policy-Report-Only` to receive violation reports without blocking anything:

```
Content-Security-Policy-Report-Only:
  default-src 'self';
  script-src 'nonce-{random}' 'strict-dynamic';
  report-to csp-endpoint;
```

Violations are reported to the endpoint you set but nothing breaks for users. Once the report stream is clean, you can switch the header to `Content-Security-Policy` to enforce. See the [reporting section below](#) for how to configure endpoints.

Verification tip: CSP is often present but ineffective. Policies that include overly permissive directives such as `*` or `'unsafe-inline'` can significantly weaken protection. Verification should focus not just on the presence of the header, but on whether it meaningfully restricts resource loading and blocks unauthorized content in practice.

CSP directive quick reference

Directive	Description
default-src	The fallback for all fetch directives that are not explicitly set. Set this first, then override individual types as needed.
script-src	Controls which scripts may execute. The most security-critical directive – the primary defense against XSS when set with nonces or hashes.
style-src	Controls which stylesheets may be applied. Relevant for data exfiltration via CSS injection .
img-src	Controls which image sources may be loaded.
font-src	Controls which font sources may be loaded via @font-face.
media-src	Controls sources for <video> and <audio>.
object-src	Controls sources for <object> and <embed>. Set to 'none' unless you have a specific requirement – objects such as browser plugins are a significant XSS risk.
frame-src	Controls which URLs may be loaded in frames and iframes on the page.
worker-src	Controls which URLs may be loaded as Worker, SharedWorker, or ServiceWorker scripts.
manifest-src	Controls which URLs may be loaded as application manifests.
connect-src	Controls the URLs that scripts can connect to via XMLHttpRequest, fetch, WebSockets, and similar. Prevents exfiltration of stolen data to attacker-controlled endpoints.

Directive	Description
form-action	Controls which URLs may be used as form submission targets. Defends against form tag hijacking . Not overridden by default-src.
frame-ancestors	Controls which sites may embed this page in a frame. The modern replacement for X-Frame-Options. Not overridden by default-src.
base-uri	Controls which URLs are permitted in the <base> element. Prevents base tag hijacking . Set to 'none' or 'self'. Not overridden by default-src.
upgrade-insecure-requests	Instructs the browser to rewrite HTTP resource requests to HTTPS before fetching. Useful when migrating a large site to HTTPS.
require-trusted-types-for	Requires that dangerous DOM sinks receive Trusted Types values, defending against DOM-based XSS. Use with 'script'.
trusted-types	Defines which Trusted Types policies are allowed to be created. Used alongside require-trusted-types-for.
report-to	Specifies the named endpoint (defined in Reporting-Endpoints) to which violation reports are sent. The modern replacement for the deprecated report-uri.
sandbox	Applies a sandbox to the page, restricting capabilities such as script execution, form submission, and popups (similar to the iframe sandbox attribute).

[Read more about configuring Content Security Policy headers](#)

HTTP Strict Transport Security (HSTS)

HSTS is the mechanism that closes the gap between a site that merely supports HTTPS and a site that truly cannot be accessed without encryption. Without HSTS, there are two categories of risk even on a fully HTTPS-capable site:

- **Partial HTTPS implementations:** Some applications only enforce HTTPS in places like login and checkout pages, leaving the rest of the session on HTTP. An attacker on the same network can intercept those unencrypted requests, modify responses, or **steal session cookies** without ever touching the HTTPS-protected pages.
- **SSL stripping attacks:** Even on all-HTTPS sites, a user's first request to `http://example.com` goes out unencrypted before the server has a chance to redirect to HTTPS. An attacker performing a **man-in-the-middle attack** can intercept that initial request, silently downgrade the connection to HTTP, and proxy the session without the user ever seeing a warning.

HSTS addresses both risks: once the browser has seen the HSTS header, it converts all requests to that host to HTTPS internally – before they leave the browser – and refuses to proceed if the certificate is invalid.

Recommended HSTS configuration

```
Strict-Transport-Security: max-age=31536000;  
includeSubDomains; preload
```

- `max-age` – how long (in seconds) the browser caches the HSTS policy. One year (31536000) is the standard minimum for preload eligibility.
- `includeSubDomains` – extends the policy to all subdomains. Only add this once you are confident all subdomains serve valid HTTPS.
- `preload` – opts the domain in to browser-maintained HSTS preload lists. Only add this when you intend to submit the domain to the HSTS preload list and have verified that all current and future subdomains can meet preload requirements (see below).

The HSTS header itself must be served over HTTPS – it is ignored over plain HTTP.

Solving the first-request problem with preloading

HSTS relies on Trust on First Use (TOFU): the browser caches the header after the first successful HTTPS response, so every subsequent request is automatically upgraded. But the very first request – before the browser has seen the header – can still be intercepted.

The HSTS preload list solves this. Browsers ship with a hardcoded list of domains that must always be loaded over HTTPS, so there is never an insecure first request. To be eligible for inclusion, your site must meet the following requirements:

1. Serve a valid certificate.
2. Redirect HTTP to HTTPS on the same hostname (not via a different domain).
3. Serve HTTPS on all subdomains, including `www`.
4. Return the HSTS header with `max-age` set to at least 31536000, and both `includeSubDomains` and `preload` directives set.

You can submit your domain at hstspreload.org. Note that preload is a long-term commitment – removal from the list is a slow process, so only add preload when you are confident your entire domain and all subdomains will remain on HTTPS indefinitely.

Verification tip: HSTS behavior cannot be fully validated from a single response. Because browsers cache HSTS policies, you should confirm that the header is consistently returned and that HTTP requests are automatically upgraded to HTTPS after the initial visit. Misconfigurations often arise when subdomains are excluded or **max-age** values are too low.

HTTP Public Key Pinning (HPKP)

Deprecated and removed from browsers. HPKP has not been supported in Chrome since version 72 (2019), and other browsers followed. Do not deploy it. Read on for context on what it was trying to solve and what now provides equivalent protection.

HPKP was designed to prevent fraudulently issued certificates from being trusted by browsers. To understand why that matters, it helps to know how certificate validation works.

When a user connects to your site over HTTPS, their browser validates your certificate against a trusted Certificate Authority (CA). The assumption is that CAs only issue certificates to the legitimate domain owner. In practice, several incidents have shown that assumption can break:

- In 2011, the Dutch CA DigiNotar was compromised and around 500 certificates were fraudulently signed, including certificates used to monitor roughly 300,000 Gmail accounts belonging to Iranian citizens.
- In 2008 and 2011, StartCom was compromised; certificates for PayPal and Verisign were issued. In the 2011 incident, attackers accessed the root key, giving them the ability to issue, invalidate, or renew any StartCom certificate.
- In 2011, the Turkish CA Turktrust mistakenly issued an intermediate CA certificate that was used to sign certificates for Google domains. Chrome caught and reported the mismatch – one of the earliest real-world demonstrations of key-pinning working as intended.

The HPKP solution was to let sites declare the expected public key fingerprints for their certificate chain. If a browser saw a certificate with an unexpected key – even one signed by a trusted CA – it would refuse the connection.

Why it was removed

The risks of operating HPKP outweighed its benefits. A misconfigured or expired pin could lock users out of a site for the entire max-age duration – potentially months – with no recovery path short of waiting for it to expire. Attackers discovered attacks such as “HPKP suicide” and ransom-pinning. Research found implementation errors even at large companies such as Facebook. Adoption was low and the error rate was high, and the mechanism was ultimately removed from Chrome in version 72 (2019).

What provides the same protection now

Certificate Transparency (CT) is the current answer – it is enforced by default for all publicly trusted certificates with no header configuration required (see the Expect-CT section below for an intermediate step in this transition). For an additional layer of control over which CAs are authorized to issue certificates for your domain, use CAA DNS records, which let you explicitly name trusted CAs and are checked during the certificate issuance process.

Expect-CT header

Deprecated and removed from browsers. Chromium deprecated Expect-CT in version 107 and has since removed it. Do not send this header – current auditing tools will flag it as a deprecated API. If you are currently sending it, remove it.

Expect-CT was a transitional header that allowed sites to opt in to Certificate Transparency enforcement and reporting before CT became mandatory.

Certificate Transparency (CT) requires CAs to submit every issued certificate to public, append-only logs. Anyone can monitor those logs for certificates issued for their domain without authorization. Since May 2018, all newly issued publicly trusted certificates are required to include Signed Certificate Timestamps (SCTs) – proof of submission to a CT log. The oldest certificates that predated CT enforcement have now all expired. Browsers enforce CT automatically for every public certificate, making the opt-in header pointless.

What to use instead

Certificate Transparency protection is now provided automatically by the browser and CA ecosystem with no configuration required on your part. If you want to actively monitor for unauthorized certificates issued for your domain:

- Use a CT log monitoring service to watch public logs for your domain name.
- Deploy CAA DNS records to restrict which CAs are authorized to issue certificates for your domain at all.

Referrer-Policy header

The `Referer` request header has an interesting history – the name with a single “r” is a misspelling that has been frozen into the HTTP specification since the 1990s. However, the `Referrer-Policy` response header that controls it is spelled correctly, which frequently creates confusion.

How the Referrer header works

When a user follows a link from your site to another, the browser automatically attaches a `Referer` header to the outgoing request, telling the destination site where the user came from:

```
GET / HTTP/1.1
Host: destination.com
Referer: https://yoursite.com/page?query=something
```

The same applies to requests triggered by loading images, stylesheets, scripts, and form submissions, not just explicit clicks.

This is useful for analytics and attribution, but it can expose more than intended: search queries, user IDs, session tokens, or internal path structures embedded in URLs can all leak to third-party servers through the `Referer` header. `Referrer-Policy` gives you control over what gets sent.

Default browser behavior

Modern browsers apply `strict-origin-when-cross-origin` by default when no policy is specified. You should still set `Referrer-Policy` explicitly – doing so makes your site’s behavior predictable across browsers and browser versions rather than relying on whatever default a given client applies.

Recommended configuration

For most sites, the browser default is also the right explicit choice:

```
Referrer-Policy: strict-origin-when-cross-origin
```

This sends the full URL on same-origin requests (useful for internal analytics), sends only the origin on cross-origin requests (so `destination.com` sees `https://yoursite.com` but not the full path), and sends nothing at all when downgrading from HTTPS to HTTP.

For applications handling sensitive data in URLs – healthcare, finance, internal tools – consider `same-origin` or `no-referrer` for stronger protection.

Verification tip: Ensure this header is returned consistently across all responses, including error pages and API endpoints.

Referrer-Policy directives

Directive	Description
no-referrer	The Referer header is never sent. Maximum privacy.
no-referrer-when-downgrade	Sends the full URL on same-origin and cross-origin requests, but nothing when downgrading from HTTPS to HTTP. This was the browser default before the spec was updated – treat it as a legacy value.
origin	Sends only the scheme and domain (e.g. <code>https://yoursite.com</code>) on all requests, never the path.
origin-when-cross-origin	Sends the full URL on same-origin requests; only the origin on cross-origin requests. Path information is also omitted when downgrading from HTTPS to HTTP.
same-origin	Sends the full URL on same-origin requests; no header at all on cross-origin requests.
strict-origin	Sends only the origin when the protocol level stays the same (HTTPS→HTTPS). Sends nothing when downgrading from HTTPS to HTTP.
strict-origin-when-cross-origin	Sends the full URL on same-origin requests; only the origin on cross-origin HTTPS→HTTPS requests; nothing on HTTPS→HTTP. This is the current browser default and the recommended explicit value for most sites.
unsafe-url	Sends the full URL on every request, including from HTTPS to HTTP. Sends sensitive URL data over unencrypted connections. Use only with full understanding of the data your URLs contain.

See the [Same-Origin Policy whitepaper](#) for more information about Same-Origin concepts.

Permissions-Policy header

Permissions-Policy lets you control which browser features and APIs your page – and any content it embeds – is allowed to use. Camera, microphone, geolocation, payment, fullscreen, autoplay, and many others can all be explicitly enabled or disabled on a per-origin basis.

This matters for two reasons. First, it limits the damage an injected script can do: even if an attacker achieves XSS on your page, they cannot silently activate the camera if Permissions-Policy has disabled it. Second, it restricts what third-party content loaded in frames on your page can do: an embedded ad or widget cannot request geolocation access unless you explicitly permit it.

Recommended starting point

Begin by disabling every feature your site does not actively use:

```
Permissions-Policy: geolocation=(), camera=(), microphone=(),  
payment=(), usb=(), fullscreen=(self)
```

An empty allowlist () disables a feature for all origins, including your own, while (self) allows it for your origin only. To allow a specific embedded origin, name it explicitly:

```
Permissions-Policy: camera=(self "https://trusted-widget.  
example.com")
```

Practical approach

Audit which features your site actually uses, disable everything else, and add Permissions-Policy to your default response headers. This is one of the highest-value, lowest-effort headers to add to a site that doesn't already have it – the risk of breaking something is low, and the surface area you remove from potential exploitation is real.

Verification tip: Ensure this header is returned consistently across all responses, including error pages and API endpoints.

Cross-origin isolation headers: COOP, COEP, and CORP

This family of headers was introduced in response to Spectre and related side-channel attacks that demonstrated JavaScript running in a browser could read memory it was not supposed to access by exploiting timing behavior in shared memory. The browser mitigations for these attacks – reduced timer resolution and disabled SharedArray Buffer – were effective but limiting.

The cross-origin isolation headers provide a structured way to re-enable those capabilities on pages that have genuinely isolated themselves from cross-origin interference. Even if you do not need SharedArrayBuffer, deploying COOP and CORP is worthwhile as defense-in-depth against cross-origin data leakage and window-reference attacks.

Cross-Origin-Opener-Policy (COOP) header

COOP controls whether your page shares a browsing context group with pages that open it or that it opens. When a cross-origin page can hold a reference to your window object, it can probe your page's URL, interact with popups, or mount Spectre-adjacent timing attacks.

Recommended COOP configuration

```
Cross-Origin-Opener-Policy: same-origin
```

This gives your page its own isolated browsing context group. Cross-origin windows can no longer hold a reference to your window object, and vice versa. Use `same-origin-allow-popups` if your site relies on cross-origin popups (e.g. OAuth flows) that need to communicate back.

Cross-Origin-Embedder-Policy (COEP) header

COEP requires that every cross-origin resource your page loads either explicitly opts in to being loaded (via CORS or CORP) or is blocked. It prevents your page from silently pulling in resources from origins that have not agreed to be embedded.

Recommended COEP configuration

```
Cross-Origin-Embedder-Policy: require-corp
```

Setting COEP alongside COOP: same-origin makes the page cross-origin isolated, which re-enables SharedArrayBuffer and high-resolution timers. Test carefully before deploying COEP – any third-party resource that does not send a permissive CORP or CORS header will be blocked.

If your page loads third-party resources that you don't control and that don't set CORP headers, require-corp will block them. An alternative value, credentialless, achieves cross-origin isolation by stripping credentials from no-cors cross-origin requests instead of requiring explicit CORP opt-in:

```
Cross-Origin-Embedder-Policy: credentialless
```

Note that cross-origin iframes remain restricted under both values and still require their own COEP headers.

Cross-Origin-Resource-Policy (CORP) header

CORP is set on your resources (images, scripts, API responses) to declare which origins are allowed to load them. Without CORP, browsers treat your resource as implicitly permitting cross-origin loads – the effective default is equivalent to Cross-Origin-Resource-Policy: cross-origin. Setting CORP explicitly to same-origin or same-site overrides that permissive default.

Recommended CORP configuration for sensitive resources

```
Cross-Origin-Resource-Policy: same-origin
```

Use same-site if resources need to be loaded by other subdomains. Use cross-origin only for resources explicitly intended to be publicly embeddable – this is the permissive value that allows pages enforcing COEP to load your resource.

Reporting header policy violations: Reporting-Endpoints and report-to

Several headers presented in this paper – most importantly CSP – can send violation reports when a policy is breached. These reports are invaluable both during rollout (to catch legitimate resources you forgot to allowlist) and in production (to detect active injection attempts).

The modern approach: Reporting-Endpoints

The most current mechanism uses the Reporting-Endpoints response header to declare named endpoint URLs, which individual policies then reference by name:

```
Reporting-Endpoints: csp-endpoint="https://example.com/csp-reports"
Content-Security-Policy: default-src 'self'; report-to csp-endpoint
```

One Reporting-Endpoints header can serve multiple policies and report types – CSP violations, COOP reports, and others can all route to the same or different endpoints:

```
Reporting-Endpoints: csp-endpoint="https://example.com/csp-reports",
coop-endpoint="https://example.com/coop-reports"
Content-Security-Policy: default-src 'self'; report-to csp-endpoint
Cross-Origin-Opener-Policy: same-origin; report-to="coop-endpoint"
```

Compatibility note

The CSP report-to directive paired with Reporting-Endpoints is the modern standards-based approach. Browser support continues to evolve, so organizations commonly deploy both report-to and report-uri during migration:

```
Content-Security-Policy: default-src 'self'; report-to
csp-endpoint; report-uri https://example.com/csp-reports
```

Browsers that support `report-to` to use it and ignore `report-uri`. Older browsers fall back to `report-uri`.

`Reporting-Endpoints` replaces the earlier `Report-To` header, which used a more complex JSON format and has since been deprecated. Use `Reporting-Endpoints` in new deployments.

Common implementation pitfalls for security headers

Even when HTTP security headers are correctly understood, real-world implementations often fall short. Common issues include:

- Headers configured in one environment but missing in production
- Security controls overridden by reverse proxies, load balancers, or CDNs
- Overly permissive policies that negate the intended protection
- Deprecated headers left in place, creating a false sense of security

Because these configuration issues are rarely visible during development, they can go unnoticed without testing the running application.

Validating security headers in practice

While individual headers can be checked manually, modern applications often expose many endpoints across multiple environments, which makes consistent validation difficult to maintain. What ultimately matters is how headers behave in the running application. Testing should confirm that headers are present, correctly configured, and consistently applied across all accessible surfaces.

Automated [dynamic application security testing \(DAST\)](#) is particularly valuable because it evaluates the application as an attacker sees it, verifying that headers are actually present and effective in production rather than merely configured in source code, infrastructure templates, or documentation.

Conclusion: Set the right headers for proactive application security

The headers covered in this paper represent a meaningful layer of browser-enforced protection – but only when they are correctly configured, consistently applied, and kept current as browser behavior evolves. As this paper shows, that last condition is not trivial: some of the most widely deployed headers are now deprecated, and others require more careful construction than a single line of configuration suggests.

Getting headers right is a start. Keeping them right across multiple environments, through frequent deployments, and as your application changes, requires treating header configuration as something that needs to be verified continuously, not just set once. Automated DAST scanning verifies headers as they actually behave in production, catching misconfiguration and drift before they become exploitable gaps.

Invicti provides a [modern AppSec platform](#) makes that verification part of your regular development and release cycle across application frontends and APIs – [request a demo](#) to see it at work.



Invicti Security provides a centralized application security platform that helps organizations prove and reduce real application risk with zero noise. Combining application security posture management (ASPM) with discovery and scanning, Invicti gives security and development teams a single, correlated view of exploitable vulnerabilities across their application frontends and APIs. With its best-of-breed dynamic application security testing (DAST) acting as the fact-checker, Invicti integrates into CI/CD pipelines to deliver proof-based results supported by AI-powered prioritization and remediation.

Built on more than 20 years of DAST innovation through Acunetix and Netsparker and further strengthened by the acquisition of Kondukto ASPM, Invicti operates globally across more than 11 countries and serves over 4,000 customer organizations.

 **Acunetix** &  **Netsparker** &  **kondukto**

ARE NOW

invicti

Invicti's solutions automate application vulnerability identification, confirmation, and management to help keep sensitive information and critical infrastructure secure.

[Find Us](#)

[LinkedIn](#)

[X \(Twitter\)](#)

[Facebook](#)

[Instagram](#)

[invicti.com](#)

[Contact Us](#)