

HTTP headers for web security

Giacomo Debidda

Freelance full stack developer / web performance consultant

I write TypeScript / Clojure / Zig

I like:

- 📖 - add me on goodreads 🔗
- 🛹 - surfskating 🔗 actually
- 🛼 - it's time for a rollerblading emoji 🍻

🐙 jackdbd

🐦 jackdbd

👤 giacomodebidda.com



Why this talk?





Important

- "Insufficient technical and organisational measures to ensure information security" is the 3rd cause of GDPR fines (EUR ~390 million as of 2024/04/09).
- The Minimum Viable Secure Product (MVSP) checklist mentions the importance of security headers.



Not talked about enough

- HTTP Headers - The State of the Web (Chrome for Developers, 2018) 
- HTTP headers for the responsible developer (Stefan Judis, 2019) 



Browsers: recent changes, new security features

- new Content-Security-Policy directives, Reporting API v1, Permissions-Policy, etc

(web) security can be **overwhelming**. Where to start?

The  [Web Security Cheat Sheet @ infosec.mozilla.org](https://infosec.mozilla.org)  lists a few recommendations.

For each recommendation, the Web Security Cheat Sheet explains:

1. Security benefit
2. Implementation difficulty
3. Suggested order for implementation
4. A few notes/suggestions

Can you spot the problem?

```
curl --head http://nginx.org
```

```
HTTP/1.1 200 OK
Server: nginx/1.25.3
Date: Wed, 03 Apr 2024 19:40:40 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 6985
Last-Modified: Thu, 28 Mar 2024 08:52:04 GMT
Connection: keep-alive
Keep-Alive: timeout=15
ETag: "66052fb4-1b49"
Accept-Ranges: bytes
```

This web page:

1. is loaded insecurely (there is no redirect to HTTPS)
2. tells us which server was served by

Why No HTTPS?



Not secure

nginx.org

Can you spot the problem?

Assume the web page is served over HTTPS.

```
<script src="http://example.com/foo.js"></script>
```

Attempts to load a script (active content) over HTTP will be blocked and will generate mixed content ⚠ warnings.

```

```

Attempts to load an image (passive/display content) over HTTP might* be allowed, but it will still generate mixed content warnings.


*Most browsers prevent mixed active content from loading, and some also block mixed display content.

Should **we** redirect to HTTPS?

Should we configure our **web server** (e.g. nginx, Caddy) to redirect HTTP to HTTPS?


No, because this leaves us vulnerable to SSL stripping attacks (a type of man-in-the-middle attack).

If a website accepts a connection through HTTP and redirects to HTTPS, visitors may initially communicate with the non-encrypted version of the site before being redirected, if, for example, the visitor types `http://www.foo.com/` or even just `foo.com`. This creates an opportunity for a man-in-the-middle attack. The redirect could be exploited to direct visitors to a malicious site instead of the secure version of the original site.

Source: Strict-Transport-Security 

Insted, we should let **the browser** redirect to HTTPS for us.

HSTS exists to remove the need for the common, insecure practice of redirecting users from `http://` to `https://` URLs.

Source: The HTTPS-Only Standard 

HTTP Strict Transport Security (HSTS)

```
Strict-Transport-Security: max-age=63072000; includeSubDomains; preload
```

This configuration tells the browser to:

1. **Connect** to the site over HTTPS, even if the scheme chosen was HTTP.
2. **Upgrade** all requests to HTTPS.
3. Treat TLS and certificate-related errors more strictly: **users will no longer be able to bypass the error page.**
4. **Preload** the HSTS configuration automatically*.
5. Do all of the above for **two years**, on **all subdomains**.

*You must first submit the form on hstspreload.org to ask Chrome to include your domain in the HSTS preload list. The approval typically takes two months 🍪.

Once HSTS is enabled, it cannot be disabled until the period specified in the header elapses. It is advisable to make sure HTTPS is working for all content before enabling it for your site. Removing a domain from the HSTS Preload List will take even longer. The decision to add your website to the Preload List is not one that should be taken lightly. ”

Source: [The Basics of Web Application Security](#) 📖

Example: setting HSTS on Cloudflare

Caution: If misconfigured, HTTP Strict Transport Security (HSTS) can make your website inaccessible to users for an extended period of time.

Enable HSTS (Strict-Transport-Security)

Serve HSTS headers with all HTTPS requests



Max Age Header (max-age)

Specify the duration HSTS headers are cached in browsers

12 months



Apply HSTS policy to subdomains (includeSubDomains)

Every domain below this will inherit the same HSTS headers

Caution: If any of your subdomains do not support HTTPS, they will become inaccessible.



Preload

Permit browsers to preload HSTS configuration automatically

Caution: Preload can make a website without HTTPS support completely inaccessible.



No-Sniff Header

Send the "X-Content-Type-Options: nosniff" header to prevent Internet Explorer and Google Chrome from MIME-sniffing away from the declared Content-Type.



HSTS on Cloudflare Docs

Hijacking user's clicks

```
<!DOCTYPE html>
<html lang="en-US">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Decoy web page</title>
</head>

<body>
  <div style="position: absolute; left: 25%; top: 5%;">
    You won! Click TOTALLY LEGIT BUTTON to get your prize!
  </div>

  <div style="position: absolute; left: 40%; top: 10%;">
    <button type="button" style="border-color: red; border-width: 2px; color: red;">
      ⚡— never gets clicked since the iframe covers the entire page →
      TOTALLY LEGIT BUTTON
    </button>
  </div>

  ⚡— this iframe captures all clicks, since it covers the entire page →
  <iframe style="opacity: 0;" width="680" height="480" scrolling="no"
    src="https://mybank/Transfer.aspx">
  </iframe>
</body>
</html>
```

X-Frame-Options and frame-ancestors

The decoy web page "re-dresses" the UI and hijacks the user's clicks.

This attack is called clickjacking, aka UI redressing.

Let's say we are the owners of

`https://mybank/Transfer.aspx` .

How do we prevent other websites from embedding our website's content into their web pages?

Yesterday

```
# prevents any domain from framing our content
X-Frame-Options: DENY
```

```
# allows our site to frame its content
X-Frame-Options: SAMEORIGIN
```

Today

```
# prevents any domain from framing our content
Content-Security-Policy: frame-ancestors 'none';
```

```
# allows our site to frame its content
Content-Security-Policy: frame-ancestors 'self';
```

The `frame-ancestors` CSP directive offers more flexibility than the `X-Frame-Options` header:

```
# allows our content to be framed by these domains
Content-Security-Policy: frame-ancestors
  https://legit-site.org
  https://another-legit-site.com
  https://*.legit-site.it
```

The above configuration allows `legit-site.org`, `another-legit-site.com`, `*.legit-site.it` to embed our content in their `<iframe>`, `<embed>`, and `<object>` .

Cross-Site Request Forgery

```
<!DOCTYPE html>
<html lang="en-US">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Mario Rossi's totally legit website</title>
</head>

<body>
  <h1>Welcome to Mario Rossi's totally legit website</h1>
  <p>Lorem Ipsum... </p>

  <form action="https://vulnerable-bank.com/transfer.html" id="send-money-to-mario-rossi" method="POST">
    <input type="hidden" name="to" value="Mario Rossi">
    <input type="hidden" name="iban" value="IT81F0300203280886251833317">
    <input type="hidden" name="amount" value="€100">
  </form>

  <script>
    document.addEventListener('DOMContentLoaded', (event) => {
      document.getElementById('send-money-to-mario-rossi').submit()
    })
  </script>
</body>
</html>
```

CSRF: **when** it works?

For a CSRF attack to be possible, three key conditions must be in place:

1. A relevant **action**. E.g. a form submission.
2. An **automatic way to submit user's credentials**. E.g. session cookies, HTTP Basic authentication, certificate-based authentication.
3. **No unpredictable request parameters**. The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

CSRF: **how** it works?

If a victim user visits the attacker's web page, the following will happen:

1. The attacker's page will **trigger an HTTP request** to the vulnerable website.
2. **If the user is logged in** to the vulnerable website, their browser will **automatically include their session cookie** in the request*.
3. The vulnerable website will process the request in the normal way, as it had been made by the victim user on the vulnerable website.

*Browsers will **not** send the cookie if it has the `SameSite=Strict` or `SameSite=Lax` attribute.

Anti-forgery tokens

The use of an anti-forgery tokens (aka request verification tokens) is the recommended, most widespread solution to mitigate CSRF attacks.




It can be achieved either with state (synchronizer token pattern) or stateless (encrypted or hashed based token pattern).

Anti-forgery tokens prevent CSRF because without a valid token, the attacker cannot create a valid request to the backend server.

The standard frequency of token generation is per-session, so make sure your sessions have a reasonable/configurable time-out. It is possible to issue new tokens on a per-request basis. However, the added protection may be insignificant.

Source: Anti CSRF Tokens ASP.NET 

Examples

- crumb (Hapi) 
- csrf-protection (Fastify) 
- ring-anti-forgery (Ring) 

Backend generates an anti-forgery token per-session.

```
<input class="csrf-token" type="hidden"
  name="__anti-forgery-token"
  value="gI4w1EuorXBhF/D3tcwk0hZtzePHqu+vjsyPv46G4ngds6HEYTp0"
```

Different session → different token.

```
<input class="csrf-token" type="hidden"
  name="__anti-forgery-token"
  value="tPr7VCcPIMixfQQsQfjSSzMLgjr3p6wALIYKRhgq6Dw7c/3BTV3o"
```

SameSite cookies

`SameSite=Strict` and `SameSite=Lax` are an excellent defense against CSRF attacks.


```
Set-Cookie: sid=session-ID-here; path=/; SameSite=Strict
Set-Cookie: sid=session-ID-here; path=/; SameSite=Lax
```


Read [this article](#) to understand which CSRF attacks are mitigated by `Strict` but not by `Lax`.

`SameSite=None` does not mitigate CSRF attacks and must always be used with Secure.

```
Set-Cookie: widget_session=abc123; SameSite=None; Secure
```

Browser support for `SameSite` cookies is very good.

This attribute `SameSite` should not replace a CSRF token. 
Instead, it should co-exist with that token to protect the user in a more robust way.

Source: [Cross-Site Request Forgery Prevention Cheat Sheet](#) 

Examples

- [@hapi/cookie](#) 
- [@hapi/yar](#) 
- [@fastify/cookie](#) 

The default name for the cookie created by `@hapi/cookie` or `@fastify/cookie` is `sid`.

The default name for the cookie created by `@hapi/yar` is `session`.

Maybe give you application's session cookie a more descriptive name than `sid` or `session`. For example:

- `sessionid` (Instagram)
- `li_at` (LinkedIn)
- `d` (Slack)
- `_twitter_sess` (Twitter)

Defense-in-depth

Even if we use anti-forgery tokens **and** `SameSite` cookies to mitigate CSRF...

Cross-Site Scripting (XSS) can defeat all CSRF mitigation techniques!

”

Source: [Cross-Site Request Forgery Prevention Cheat Sheet](#)

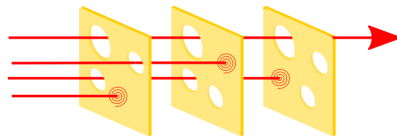
...we still need to mitigate XSS to avoid CSRF attacks.

Since no single technique will solve XSS, using the right combination of defensive techniques will be necessary to prevent XSS.

”


Source: [Cross Site Scripting Prevention Cheat Sheet](#)

This approach of **layered security** is called **defense-in-depth**. The principle behind it is the so-called **Swiss cheese model**.



How dangerous is XSS?

XSS attacks can:

- bypass CSRF protection;
- capture the user's login credentials;
- read any data that the user is able to access;
- carry out any action that the user is able to perform;
- inject malicious code the web site;
- perform a plethora of other attacks. 

Likelihood and Impact of XSS vulnerabilities

In risk analysis: $Risk = Likelihood \times Impact$

Likelihood

If there is no way to enter untrusted data on the website (i.e. no `input`, no `textarea`, etc), XSS attacks will be unlikely but not impossible (e.g. a malicious Chrome extension).

The more `input`, `textarea`, etc there are, the easier will be to forget to perform HTML sanitization on at least one of them.

If JS is disabled, client-side XSS attacks are not possible. However, attacks that exploit server-side vulnerabilities or manipulate HTML and CSS in a way that doesn't rely on JS execution are still possible.

The OWASP Risk Rating Methodology page [🔗](#) offers guidelines for assessing likelihood and impact, and for estimating the resulting risk.

Impact


The **actual** impact of an XSS attack generally depends on the nature of the application, its functionality and data, and the status of the compromised user. ”

Source: Impact of XSS vulnerabilities [🔗](#)

In a brochureware website, where all users are anonymous and all information is public, the impact will often be low.

In a website holding sensitive data, such as banking transactions, credit card numbers, or healthcare records, the impact will usually be high.

Is our site vulnerable to XSS?

We can check most kinds of XSS vulnerabilities by injecting a payload that causes the browser to execute some arbitrary JavaScript. This is called XSS proof of concept .

1. Enter this snippet into an `<input>` (e.g. search bar, email field, etc). If the alert shows up **immediately**, the website is vulnerable to **reflected XSS**.

```
<script>alert('hi')</script>
```

2. Enter this snippet into a comment box (e.g. a `<textarea>`). If this snippet is stored as it is, the alert will show up **every time the page is visited**. This means the website is vulnerable to **stored XSS (aka persistent or second-order XSS)**.

```
<script>alert('hi')</script>
```

3. Open Chrome DevTools and execute this snippet in the console. If the alert shows up, the website is vulnerable to **DOM-based XSS**.

```
const script = document.createElement("script")
script.innerText = "alert('hi')"
document.head.appendChild(script)
```

In some cases we need a complex XSS proof of concept to spot a XSS vulnerability.

How do we mitigate XSS?

1. Use a **modern web framework** that has templating, auto-escaping, etc.

When you use a modern web framework, you need to know how your framework prevents XSS and where it has gaps. There will be times where you need to do something outside the protection provided by your framework, which means that Output Encoding and HTML Sanitization can be critical.

Source: Cross Site Scripting Prevention Cheat Sheet (Framework Security) [🔗](#)

2. Leave **output encoding / escaping** to your framework or use a popular library.
3. **Sanitize HTML** with a library like DOMPurify [🔗](#).
4. Define a strict **Content-Security-Policy** tailored for your site / web app.

In the context of XSS defense, CSP works best when it is:

- Used as a defense-in-depth technique.
- Customized for each individual application rather than being deployed as a one-size-fits-all solution.

Source: Cross Site Scripting Prevention Cheat Sheet (Common Anti-patterns) [🔗](#)

CSP: how hard can it be?

How it started:

```
Content-Security-Policy: default-src 'self'; img-src 'self' cdn.example.com;
```

How it's going (GitHub):

```
default-src 'none';
base-uri 'self';
child-src github.com/assets-cdn/worker/ gist.github.com/assets-cdn/worker/;
connect-src 'self' uploads.github.com www.githubstatus.com collector.github.com raw.githubusercontent.com api.github.com gi
font-src github.githubassets.com;
form-action 'self' github.com gist.github.com copilot-workspace.githubnext.com objects-origin.githubusercontent.com;
frame-ancestors 'none';
frame-src viewscreen.githubusercontent.com notebooks.githubusercontent.com;
img-src 'self' data: github.githubassets.com media.githubusercontent.com camo.githubusercontent.com identicons.github.com a
manifest-src 'self';
media-src github.com user-images.githubusercontent.com/ secured-user-images.githubusercontent.com/ private-user-images.gith
script-src github.githubassets.com;
style-src 'unsafe-inline' github.githubassets.com;
upgrade-insecure-requests;
worker-src github.com/assets-cdn/worker/ gist.github.com/assets-cdn/worker/
```

CSP: Reddit

```
child-src 'self' blob: accounts.google.com;
connect-src 'self' events.redditmedia.com o418887.ingest.sent
default-src 'self';
font-src 'self' data;;
form-action 'none';
frame-ancestors 'self' *.reddit.com *.snooguts.net;
frame-src 'self' www.reddit.com www.youtube-nocookie.com play
img-src 'self' data: blob: https;;
manifest-src 'self' www.redditstatic.com;
media-src 'self' blob: data: *.redd.it www.redditstatic.com;
object-src 'none';
script-src 'self' 'unsafe-inline' 'unsafe-eval' www.redditsta
style-src 'self' 'unsafe-inline' www.redditstatic.com *.reddi
style-src-attr 'unsafe-inline';
worker-src 'self' blob;;
report-to csp;
report-uri https://w3-reporting-csp.reddit.com/reports
```

There are 30+ CSP directives for a variety of resources, including fonts, frames, images, audio and video media, scripts, and workers.

CSP: X / Twitter

```
connect-src 'self' blob: https://api.x.ai https://api.x.com h
default-src 'self';
form-action 'self' https://twitter.com https://*.twitter.com
font-src 'self' https://*.twimg.com;
frame-src 'self' https://twitter.com https://x.com https://mo
img-src 'self' blob: data: https://*.cdn.twitter.com https://
manifest-src 'self';
media-src 'self' blob: https://twitter.com https://x.com http
object-src 'none';
script-src 'self' 'unsafe-inline' https://*.twimg.com https:/
style-src 'self' 'unsafe-inline' https://accounts.google.com/
worker-src 'self' blob;;
report-uri https://twitter.com/i/csp_report?a=05RXE%3D%3D%3D6
```

Browser support is messy. For example:

- older browsers may support `style-src` , but not `style-src-attr` or `style-src-elem` .
- Firefox: no `report-to` , no `trusted-types` .
- Safari: no `manifest-src` , no `trusted-types` .

Learn CSP the hard way

The `default-src` CSP directive serves as a fallback for the other CSP fetch directives.

Set it to `'none'` and your site **will** break:

```
Content-Security-Policy: default-src 'none';
```

The good news is that you will know **exactly** why it broke. In DevTools you will see errors like these ones:

- Refused to connect to '<URL>' because it violates the following CSP directive: ...
- Refused to load the script 'foo.js' because it violates ...
- Refused to apply inline style because it violates ...
- Refused to load the font 'foo.woff2' because it violates ...

CSP directives: be specific!

Whenever possible*, opt for the **most specific** CSP directive available.

🥉 Good: `default-src 'self'`

🥈 Better: `script-src 'self'`

🥇 Best: `script-src-elem 'self'`





Why this? To mitigate CSP bypasses.

*Remember: browser support for CSP directives is a bit messy.

Maintain your CSP

1. Write the **strictest CSP** for your site.
2. **Test your CSP** on [Mozilla Observatory](#) (personal favorite), [CSP Evaluator](#), or [Security Headers](#).
3. Whenever you **add new content** (e.g. new inline style), a new asset (e.g. image, font) or connect to a new domain (e.g. with a `<link rel="prefetch">`), check for CSP errors/warnings in DevTools. Update your CSP accordingly.
4. Whenever you **remove content** (e.g. you are no longer hosting images on that CDN but you are now self-hosting them), review your CSP.
5. Configure a **security logging service** like [Report URI](#) to catch CSP violations. You can do this using the `report-to` and the `report-uri` directives.

Useful tips

- ❌ Do not write your CSP by hand in a `_headers` file (Cloudflare Pages, Netlify) or in a `vercel.json` file. It will soon become really hard to maintain.
- ❌ Do not rely on a tool that writes a generic CSP for you. Your CSP must be tailored to your site.
- ✅ If your CSP is simple, consider a low-key approach, like generating `_headers` / `vercel.json` using a templating engine. See [this example with Nunjucks](#).
- ✅ If your CSP grows in size, use a tool for writing it. For example: [seespee](#) , [netlify-plugin-csp-generator](#) , [@jackdbd/content-security-policy](#) , [@jackdbd/eleventy-plugin-content-security-policy](#) .
- ✅ If you can't afford breaking your site in production, replace `Content-Security-Policy` with `Content-Security-Policy-Report-Only`.

Permissions-Policy (prev. Feature-Policy)

Permissions Policy allows the developer to control the **browser features** available to a page, its iframes, and subresources, by declaring a set of policies for the browser to enforce.

Example: Instagram

Permissions-Policy:

```
accelerometer=(self), attribution-reporting=(), autoplay=(),bluetooth=(), camera=(self),
ch-device-memory=(), ch-downlink=(), ch-ect=(), ch-rtt=(), ch-save-data=(), ch-ua-arch=(), ch-ua-bitness=(),
clipboard-read=(), clipboard-write=(self), display-capture=(),encrypted-media=(), fullscreen=(self),
gamepad=(), geolocation=(self), gyroscope=(self), hid=(), idle-detection=(), keyboard-map=(), local-fonts=(),
magnetometer=(), microphone=(self), midi=(), otp-credentials=(), payment=(), picture-in-picture=(self),
publickey-credentials-get=(), screen-wake-lock=(), serial=(), usb=(), window-management=(), xr-spatial-tracking=();
report-to="permissions_policy"
```

Those `ch-*` are directives for HTTP client hints.

A server sends the `Accept-CH` header to specify the client hints that it is interested in receiving.

Accept-CH:

```
viewport-width,dpr,
Sec-CH-Prefers-Color-Scheme,Sec-CH-UA-Full-Version-List,Sec-CH-UA-Platform-Version,Sec-CH-UA-Model
```

Referrer-Policy

The `Referer` header can contain **origin**, **path**, and **querystring**.

The `Referrer-Policy` controls how much information should be included in the `Referer` header when making same-origin requests and cross-origin requests.

When no policy is set, the browser's default is used.

This is the default in most browsers:

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





And this is what it means:

- same-origin request: send everything, namely origin, path, and querystring.
- cross-origin request:
 - HTTPS→HTTPS: send just the origin.
 - HTTPS→HTTP: don't send the `Referer` header at all.

Origin

1. **scheme**, i.e. the protocol (e.g. `http` , `https` , `ws` , `wss`)
2. **host**, i.e. the domain (e.g. `example.com`)
3. **port** (e.g. `80` , `8080` , `3000`)

Example: `https://example.com` (80 is the default port for HTTP)

URL	Same origin?
<code>http://example.com</code>	 Different protocol
<code>https://www.example.com</code>	 Different host
<code>https://example.com:8080</code>	 Different port
<code>https://example.com/foo</code>	 Same origin

Origin vs Site

Request from	Request to	Same-site?	Same-origin?
<code>https://example.com</code>	<code>https://example.com</code>	✓	✓
<code>https://app.example.com</code>	<code>https://intranet.example.com</code>	✓	✗ domain name
<code>https://example.com</code>	<code>https://example.com</code>	✓	✗ port
<code>https://example.com</code>	<code>https://example.co.uk</code>	✗ eTLD	✗ domain name
<code>https://example.com</code>	<code>https://example.com</code>	✗ scheme	✗ scheme

Understanding "same-site" and "same-origin"

Cross-origin requests

Let's say this page is hosted at `https://www.foo.com`. Will the image show up (assuming it exists)?

```
<body>
  
</body>
```

What about now?

```
<body>
  <div id="container"></div>
  <script>
    document.addEventListener('DOMContentLoaded', async () => {
      const container = document.getElementById("container")
      try {
        const response = await fetch("https://www.bar.com/image.jpg") // cross-origin fetch
        const blob = await response.blob()
        const img = document.createElement("img")
        img.src = URL.createObjectURL(blob)
        container.appendChild(img)
      } catch(err) {
        console.error(err)
      }
    })
  </script>
</body>
```

Same-origin policy (SOP)

What is permitted and what is blocked?

Cross-origin embedding

```

```



Cross-origin reads

```
const response = await fetch("<img-URL>", { mode: 'cors' })
```

1. Failed to fetch

```
const response = await fetch("<img-URL>", { mode: 'no-cors' })
```

1. image fetched successfully
2. blob.size is 0

Same-origin policy and CORS

Cross-Origin Resource Sharing (CORS) response headers:

- Access-Control-Allow-Credentials
- Access-Control-Allow-Headers
- Access-Control-Allow-Methods
- Access-Control-Allow-Origin
- Access-Control-Expose-Headers
- Access-Control-Max-Age
- Access-Control-Request-Headers
- Access-Control-Request-Method

CORS misconfigurations can be dangerous: Exploiting CORS misconfigurations for Bitcoins and bounties 

Same-origin policy and CORP, COEP and COOP

- Cross-Origin-Resource-Policy (CORP)
- Cross-Origin-Embedder-Policy (COEP)
- Cross-Origin-Opener-Policy (COOP)

Send CORP with a **resource** (e.g. an image).

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

Send COEP and COOP with the **top-level document**.


```
Cross-Origin-Embedder-Policy: require-corp  
Cross-Origin-Opener-Policy: same-origin
```

You need this configuration of COEP and COOP to achieve cross-origin isolation.

Reporting API

The browser can generate reports when there are:

- security violations (configured with `*-Policy` headers)
- deprecated API calls
- browser interventions
- crashes

The browser decides when to send these reports to the endpoint(s) you configured.
You can host your reporting server or use a security logging service like Report URI .

Reporting API v0

- `report-uri` directive +
- `Report-To` header +
- `NEL` (Network Error Logging) header

Reporting API v1

- `report-to` directive +
- `Reporting-Endpoints` header

Reporting-Endpoints example: Instagram

Note the `report-to` directive in **some** of the `*-Policy` headers down below.

```
Content-Security-Policy: <directives-not-shown>;report-uri https://www.facebook.com/csp/reporting/?m=t&minimize=0;  
Cross-Origin-Embedder-Policy-Report-Only: require-corp;report-to="coep_report"  
Cross-Origin-Opener-Policy: same-origin-allow-popups;report-to="coop_report"  
Document-Policy: force-load-at-top  
Permissions-Policy: accelerometer=(self),<directives-not-shown>;report-to="permissions_policy"
```

This is Instagram's `Reporting-Endpoints` header configuration.

```
Reporting-Endpoints:  
  coop_report="https://www.facebook.com/browser_reporting/coop/?minimize=0",  
  coep_report="https://www.facebook.com/browser_reporting/coep/?minimize=0",  
  default="https://www.instagram.com/error/ig_web_error_reports/?device_level=unknown",  
  permissions_policy="https://www.instagram.com/error/ig_web_error_reports/"
```

Despite its name, `default` is **not a fallback** endpoint.

”

Source: Monitor your web application with the Reporting API 

Report URI

<https://report-uri.com/>

TODO: add screenshots from Notion (Report URI note, NEL note) and JSON payload of some CSP violations

Example: CSP violation report

```
{
  "age": 2,
  "body": {
    "blockedURL": "https://site2.example/script.js",
    "disposition": "enforce",
    "documentURL": "https://site.example",
    "effectiveDirective": "script-src-elem",
    "originalPolicy": "script-src 'self'; object-src 'none'; report-to main-endpoint;",
    "referrer": "https://site.example",
    "sample": "",
    "statusCode": 200
  },
  "type": "csp-violation",
  "url": "https://site.example",
  "user_agent": "Mozilla/5.0 ... Chrome/92.0.4504.0"
}
```

Caching

A misconfiguration of one or more headers that influence caching can lead to web cache poisoning.

CWE-525 Use of Web Browser Cache Containing Sensitive Information

To understand how web cache poisoning vulnerabilities arise, it is important to have a basic understanding of how web caches work.

See A Tale of Four Caches.

For example, use `no-store` to avoid caching sensitive information:

```
Cache-Control: no-store
```

When CORS are misconfigured and `Vary: Origin` hasn't been specified, the response may be stored in the browser's cache. Exploiting CORS misconfigurations for Bitcoins and bounties

Use the `Clear-Site-Data` header to purge browsing data (cookies, storage, cache).

As Jake Archibald suggests in What happens when packages go bad?, an `/emergency` URL could serve a `Clear-Site-Data: *` header, deleting everything stored & cached by the origin, then redirect to `/`.

Scott Helme's report February 2024

<https://scotthelme.co.uk/tag/crawler-report/>

Security Headers Grades (February 2024)

<https://securityheaders.com/>

- A+: 4,544
- A: 43,012
- B: 34,871
- C: 34,440
- D: 151,558
- E: 21,550
- F: 479,586
- R: 112

Sites using strict-transport-security: 200,935

Sites using content-security-policy: 95,961

Sites using x-content-type-options: 196,160

Sites using x-frame-options: 233,397

Sites using x-xss-protection: 142,383

Sites using referrer-policy: 119,915

Sites using feature-policy: 4,888

Sites using permissions-policy: 36,689

Sites using report-to: 221,996

Sites using nel: 220,512

Sites using security.txt: 10,715

Sites redirecting to HTTPS: 580,538

OWASP Top 10 Web Application Security Risks

1. Broken Access Control
2. Cryptographic Failures (previously known as Sensitive Data Exposure)
3. Injection (from 2021, Cross-site Scripting is part of this category)
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures (previously known as Broken Authentication)
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery


The OWASP Top 10 is updated every 3-4 years.

The previous version was published in 2017. The next update is planned for September 2024.

Broken access control

https://owasp.org/Top10/A01_2021-Broken_Access_Control/

Common access control vulnerabilities include:

- Violation of the principle of least privilege.
- Bypassing access control checks by modifying the URL, internal application state, or the HTML page, or by using an attack tool modifying API requests.
- Permitting viewing or editing someone else's account, by providing its unique identifier (insecure direct object references).
- Accessing API with missing access controls for POST, PUT and DELETE.
- Privilege escalation.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.
- **CORS misconfiguration**. E.g. Exploiting CORS misconfigurations for Bitcoins and bounties 

Cryptographic Failures

passwords, credit card numbers, health records, personal information, and business secrets require extra protection, mainly if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations, e.g., financial data protection such as PCI Data Security Standard (PCI DSS)

CORS misconfiguration

Probably the most insecure CORS configuration you can have:

```
Access-Control-Allow-Origin: *  
Access-Control-Allow-Methods: *  
Access-Control-Allow-Headers: *
```

This was actually suggested on Stack Overflow: <https://stackoverflow.com/a/75997573/3036129>

No wait, this is worse:

```
Access-Control-Allow-Origin: null
```

<https://jakearchibald.com/2021/cors/>

Learn more

- <https://www.hackinbo.it/>
- CS 253 Web Security (Stanford)
- PortSwigger, HackTheBox, TryHackMe
- <https://owasp.org/www-community/meetings/>
- Google Cybersecurity Professional Certificate

Wrap up

todo

Cost of a data breach

- GDPR
- HIPAA
- PCI DSS

<https://eriskhub.com/mini-calc-usli>

Violators of GDPR may be fined up to €20 million, or up to 4%
of the annual worldwide turnover of the preceding financial year,
whichever is greater.

Source: [GDPR fines and notices](#)

The [Enforcement Tracker](#) gives an overview of reported
fines and penalties which data protection authorities
within the EU have imposed so far.

Estimate the GDPR fine of a German company:

<https://www.enforcementtracker.com/?finemodel-germany>

Le banche devono adottare **tutte** le necessarie misure tecnico-
organizzative e di sicurezza per evitare che i dati dei propri clienti
possano essere sottratti illecitamente.

Lo ha affermato il Garante per la privacy nel sanzionare UniCredit
banca per una violazione di dati personali (data breach) avvenuta
nel 2018, che ha coinvolto migliaia di clienti ed ex clienti.

Source: [Data breach: il Garante sanziona UniCredit per 2,8 milioni
di euro \(Multa di 800mila euro anche alla società incaricata di
effettuare i test di sicurezza\)](#)

GDPR fines

Country	Date of decision	Fine (€)	Controller/Processor	Type	Source
	2024-02-08	300,000	Medtronic Italia	Non-compliance with general data processing principles	link link
	2024-02-22	50,000	Azienda Trasporto Passeggeri Emilia-Romagna S.p.A.	Non-compliance with general data processing principles	link link

The end

todo