PRACTICAL LAYERED SECURITY FOR SMALL PLATFORMS

Chapter 4: Browser Hardening Headers Overview

A website can use valid HTTPS and still leave the browser with weaker defenses than necessary if the application does not send clear response-header rules. This chapter explains the practical role of core browser hardening headers and clarifies why HSTS is valuable but not sufficient on its own -- then proves the point against two real incidents on a live small-platform site.

STATUS	CHAPTER ROLE	PREPARED
Pre-publish review copy Prepared for last content and layout checks before it moves into the real project source.	**Browser hardening baseline** Six-header baseline, CSP strict-vs-basic trade-offs, and two real production incidents.	**2026-07-09** Publication-style local pass with all three chapter diagrams embedded.

Terms Worth Defining First

Eleven terms are worth defining before going further: three foundational ones used since the Problem statement above, then the eight that appear in the Risk paragraph, in the order they appear. Three more (DNS, DNSSEC, DMARC) come up once the chapter separates browser hardening from other security layers.

HTTP

HyperText Transfer Protocol. The protocol browsers and servers use to exchange requests and responses -- by default sent as plain, unencrypted text that anyone on the network path (public Wi-Fi, an ISP, a compromised router) can read or alter in transit.

HTTPS

HTTP Secure. The same protocol wrapped in TLS encryption. It hides the exchange's content from the network path and lets the browser verify it is actually talking to the real server via a certificate. Everything else in this chapter -- CSP, HSTS, secure cookies -- assumes HTTPS is already in place; none of it substitutes for it.

TLS

Transport Layer Security. The encryption protocol that does the actual work behind HTTPS: a private, tamper-evident channel between browser and server, and the thing that verifies the server's certificate during the handshake. "HTTPS" is really just "HTTP running over TLS."

Clickjacking

An attacker invisibly loads your page inside a frame on their own page, then stacks a fake button or image on top of it. A visitor thinks they are clicking the attacker's content; they are actually clicking your site's real button underneath, hidden from view.

Referrer leakage

When a visitor navigates away from your site, their browser can tell the destination site exactly which URL they came from -- including path segments, query parameters, or session details that were never meant to leave your domain.

Content sniffing

A browser ignoring the Content-Type your server declared and guessing the file's real type from its content instead, which can let a file uploaded as an "image" get executed as HTML or JavaScript if its bytes resemble a script.

Content-injection mistakes

The general category of bug where content the site never intended to serve -- attacker-supplied HTML, scripts, or other markup -- ends up rendered as part of the page. XSS is the most common and most severe form of this.

HSTS

HTTP Strict Transport Security. A header that tells the browser "never load this site over plain HTTP again, for the next N seconds -- upgrade to HTTPS yourself, even if a link or bookmark points to http://." It closes the gap where a visitor's very first request could still go out over an insecure connection.

XSS

Cross-site scripting. An attacker gets their own <script> code to run inside your page, as if it were your own code -- usually because user-controlled input got inserted into the page's HTML without being escaped first.

Unsafe script execution

Any situation where a script runs with more trust or capability than intended -- eval()-ing untrusted strings, loading third-party scripts with no integrity check, or failing to block an injected inline script.

Permissive framing behavior

The page allowing itself to be embedded in a <frame> or <iframe> on another site with no restriction -- the underlying condition clickjacking depends on.

DNS

Domain Name System. The lookup that translates a human-readable domain into the IP address computers actually connect to. Every visit to the site starts with this lookup, before any browser header ever comes into play.

DNSSEC

DNS Security Extensions. Adds a cryptographic signature to DNS answers, so a resolver can verify a response genuinely came from the domain's real authoritative nameserver -- the DNS equivalent of the browser hardening this chapter covers, but for name lookups instead of page content.

DMARC

Domain-based Message Authentication, Reporting & Conformance. A DNS-published policy that tells receiving mail servers what to do with email claiming to be "From:" your domain that fails SPF/DKIM checks.

Main Discussion

Browser hardening headers are one of the most cost-effective defensive layers for a public site. They do not replace secure code, but they help the browser enforce safer behavior after the server sends the response.

This matters especially for small platforms because browser trust is often one of the few defensive layers fully controlled by the application owner. A team may host the public site on a managed platform and still remain responsible for how the browser is instructed to render, frame, fetch, or expose data.

This also helps separate browser hardening from other security layers. DNSSEC protects DNS-answer trust, DMARC protects business-email sender identity, and rate limiting helps reduce abusive request pressure. Those layers matter, but they do not tell the browser what it may execute, embed, expose, or trust after the page is delivered.

The practical baseline in this chapter is:

- `Content-Security-Policy` (CSP) restricts where scripts, styles, frames, and other resource types can come from.
- `X-Frame-Options` reduces clickjacking risk by restricting whether the page may be framed by another site.

- `Referrer-Policy` limits how much navigation-source information is leaked to other destinations.
- `X-Content-Type-Options: nosniff` reduces browser content-type guessing.
- `Permissions-Policy` reduces access to unnecessary browser features such as camera, microphone, or geolocation.
- `Strict-Transport-Security` (HSTS) tells the browser to use HTTPS only and remember that rule for future visits.

That third-party discipline becomes important quickly on modern sites. A tag manager, analytics loader, widget, or external embed may be useful for the business, but each one expands the browser trust surface and makes weaker header decisions more expensive later -- exactly what the two case studies later in this chapter walk through on a real tag-manager integration.

It is also important to distinguish between merely having a header and having a useful header. A platform can send a CSP and still keep it too permissive, or send HSTS with a weak scope that leaves parts of the environment outside the intended transport policy. As the case studies show, a header configuration can also be entirely correct on paper and still fail to reach the browser at all, because of how the hosting platform routes the request -- which is why this chapter treats live verification as part of the header, not an optional extra step.

A real, non-hypothetical example

A hosting provider may already return HTTPS and even a default HSTS header of its own, while CSP, frame restrictions, referrer controls, content-sniffing protection, and permissions restrictions still depend entirely on the application owner's own configuration choices. It is easy to mistake a platform's default header for a deliberate security decision -- the two are not the same thing, and only one of them is actually under the operator's control.

This happened on the exact site this chapter's case study covers, caught by running the live check this chapter recommends:

`Live check`

```
curl -I https://www.example.com | grep -Ei 'strict-transport-security|content-security-policy|x-frame-options|referrer-policy|x-content-type-options|permissions-policy'
```

```
strict-transport-security: max-age=63072000
```

Only HSTS came back, and at the hosting platform's own built-in default value -- not the value the project's own configuration had just set. CSP, X-Frame-Options, Referrer-Policy, X-Content-Type-Options, and Permissions-Policy were all completely absent from the response. The platform was supplying transport security on its own; every other header on this list still depended entirely on configuration the application owner had to set explicitly -- and in this case, an unrelated routing bug (Case Study 1, below) was silently preventing that configuration from ever reaching the browser at all.

Figure 4: Browser Hardening Header Flow

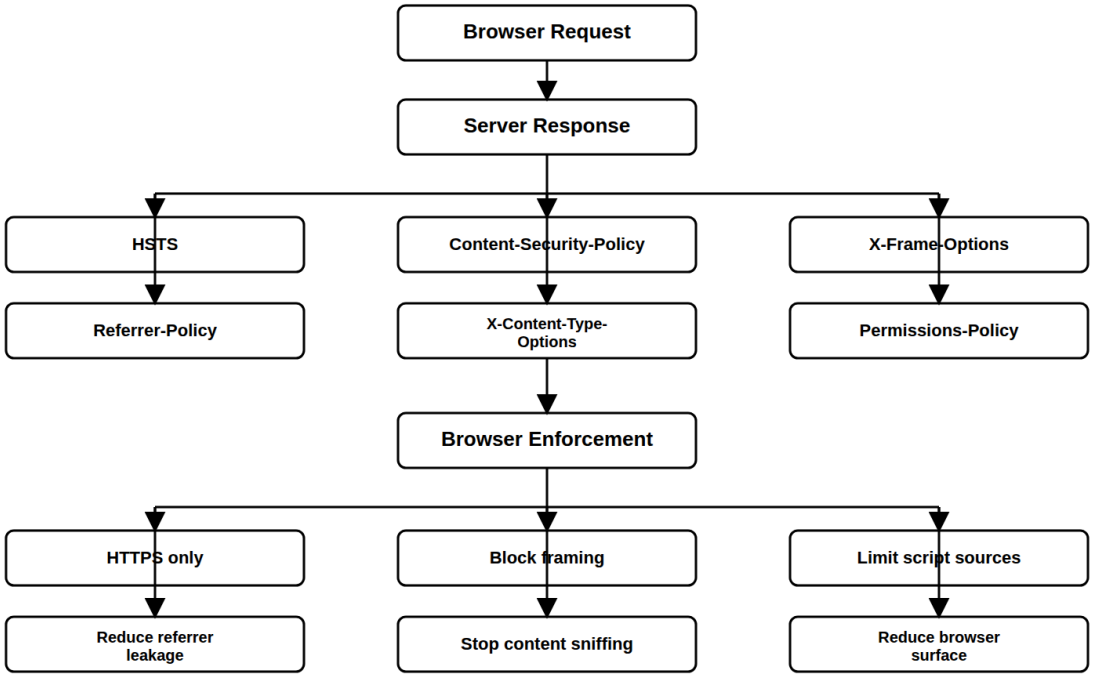

Figure 4. Browser hardening header flow

Response headers are not decorative metadata. They are instructions that shape browser behavior after the server responds -- HSTS, CSP, X-Frame-Options, Referrer-Policy, X-Content-Type-Options, and Permissions-Policy each become a specific browser enforcement rule.

Practical Reading of the Header Set

The safest way to read this chapter is not as a memorization exercise, but as a small model of browser control:

- HSTS keeps the browser on HTTPS.
- CSP narrows what active content is allowed to load or execute.
- X-Frame-Options reduces the chance that another page can visually trap the site inside a hostile frame. CSP's `frame-ancestors` is the modern replacement, and the two are commonly layered together rather than treated as alternatives.
- Referrer-Policy reduces unnecessary outbound information leakage.

- X-Content-Type-Options reduces unsafe type guessing.
- Permissions-Policy reduces exposure to browser features the page does not actually need.

Together, these do not make the application secure by themselves. What they do is reduce how much freedom the browser has to make permissive assumptions when it handles the response.

CSP: Basic Versus Strict

A basic CSP with `script-src 'self'` and `unsafe-inline` is easier to deploy on an existing site, but it leaves a significant gap. The `unsafe-inline` keyword allows any inline script to run, which means an attacker who can inject a `<script>` tag bypasses the policy entirely.

The practical progression is to start with a basic policy, monitor violations, then move toward a strict CSP using nonces or hashes. If you are not ready for strict CSP, use `Content-Security-Policy-Report-Only` first -- it sends the policy to the browser without enforcing it, logging violations to a report URI (the endpoint URL the browser sends violation reports to) so you can identify what would break before you block anything.

A common real obstacle to strict CSP is third-party analytics and tag-manager tooling, which often ships as an inline bootstrap script. This is not a reason to abandon strict CSP -- it is a reason to hash or nonce that specific script rather than opening the whole policy with `unsafe-inline`. The case studies later in this chapter walk through exactly this situation on a live site.

IF THAT DISTINCTION WAS HARD TO FOLLOW

`unsafe-inline` leaves the door unlocked for any inline script, attacker-written or not; a nonce or hash is a key that only your own approved scripts hold.

Beyond the Six-Header Baseline

Once the core six headers are in place, three additional headers form the next layer of cross-origin isolation:

- `Cross-Origin-Opener-Policy` (COOP) isolates your browsing context from pop-ups, closing off `window.opener` attacks.
- `Cross-Origin-Embedder-Policy` (COEP) pairs with COOP for a cross-origin isolated state, a prerequisite for APIs like `SharedArrayBuffer`.
- `Cross-Origin-Resource-Policy` (CORP) prevents your resources from being loaded by cross-origin documents.

These are part of the modern defensive baseline. The practical approach is to add the core six first, then layer these in as the platform matures.

Common Misconfigurations to Avoid

- **CSP with unsafe-inline and a long domain allowlist.** Looks secure but is not -- an attacker who can inject a script tag bypasses the policy. Prefer nonces or hashes, keep allowlists short.
- **HSTS without includeSubDomains.** Leaves subdomains unprotected. Note: `includeSubDomains` itself creates no new exposure -- it is a browser-enforcement instruction, not a DNS record or certificate.
- **HSTS without preload.** First-time visitors remain vulnerable until they hit the site once over HTTPS. Treat preload as a one-way commitment, decided separately from `includeSubDomains`.
- **Permissions-Policy that only lists a few features.** The default is often allow-all -- be explicit about what you do not need.
- **Missing headers on API responses or static asset subdomains.** Headers should be on all responses, not just the main HTML document.
- **A header configuration that never reaches the browser.** A config file can be entirely correct and still fail silently if combined with an unrelated legacy routing rule. See Case Study 1, below.

- **Adding X-XSS-Protection.** Intentionally excluded from this baseline -- deprecated, unsupported by modern browsers, and historically introduced its own vulnerabilities.
- **A CSP hash that does not actually match the script it references.** Per the CSP Level 3 precedence rule, the mere presence of a hash-source, correct or not, causes modern browsers to ignore `'unsafe-inline'` entirely for that directive. See Case Study 2, below, for a documented real example.
- **A per-request value protected only by Cache-Control.** A nonce needs to survive every caching layer between origin and browser, not just the browser's own cache -- a CDN typically honors its own separate cache-control header.

Why HSTS Is Useful but Not Enough

HSTS means HTTP Strict Transport Security. In practical terms, it tells the browser to use HTTPS only and remember that rule for a defined period, reducing downgrade and SSL-stripping style risks.

That is real value, but HSTS is only one layer. It does not control which scripts can run. It does not stop a framed page. It does not sanitize HTML. This is why mature browser hardening always combines HSTS with content restrictions, frame restrictions, safer rendering patterns, and careful third-party script review.

The practical mindset is to treat HSTS as transport enforcement, not application trust enforcement. A site can encrypt the connection properly and still allow weak browser behavior afterward -- secure transport and secure browser behavior are related, but they are not the same thing.

Real-World Case Study: Fixing a Live Site's Headers

A documented remediation pass on a live small-platform site, carried out while this chapter was being written, using the exact validation approach this chapter

recommends: checking the live response, not just the configuration file.

The starting point was a reasonable-looking `Strict-Transport-Security` header (`max-age=63072000`) with no `includeSubDomains` and no `preload`, and a CSP that included `'unsafe-inline'` in `script-src` to accommodate a Google Tag Manager integration. Two decisions had to be made and one hidden bug had to be found before the live site actually matched what this chapter recommends.

HSTS scope

The operator had deliberately left out `includeSubDomains`, reasoning that it might make subdomains easier for an attacker to discover through certificate-transparency log lookups or wordlist-based subdomain enumeration. That reasoning conflated two unrelated things -- `includeSubDomains` is a browser-side enforcement instruction with no DNS or certificate footprint. What actually creates that exposure is issuing a certificate or DNS record for a subdomain, which the operator had not done. The fix: add `includeSubDomains` at no additional exposure, and defer `preload` as a deliberate, later decision given its irreversibility.

The CSP/tag-manager trade-off

The `'unsafe-inline'` value existed to let a GTM bootstrap script run. Three options were on the table: hash the exact bootstrap script, use a per-request nonce with `strict-dynamic` (needed only if the container also injects ad-hoc inline scripts via Custom HTML tags), or add a hash alongside `'unsafe-inline'` as a transitional step that CSP Level 3 browsers honor while older browsers keep working unchanged. After confirming the container used only standard, built-in tag types, the transitional hash-plus-fallback approach was chosen:

`vercel.json -- Content-Security-Policy`

```
script-src 'self' 'sha256-<hash-of-exact-bootstrap-script>' 'unsafe-inline'
  https://www.googletagmanager.com https://www.google-analytics.com;
```

The hidden routing bug

After deploying both fixes, a live `curl -I` check showed something unexpected: every header except a bare, platform-default `Strict-Transport-Security` was missing entirely from the response. The first instinct was to suspect a stale CDN cache -- worth ruling out systematically, but a guaranteed-fresh, cache-cleared redeploy still showed the same missing headers.

The actual root cause was a routing configuration conflict. The project's deployment config combined a legacy, sequentially-processed routing array with a separate header-injection configuration. The routing array ended in a catch-all rewrite rule with no explicit instruction to keep evaluating further rules after a match -- so for every single request, routing terminated there before the header-injection step was ever reached. The fix was to replace the legacy routing array with the platform's modern, purpose-built redirect and rewrite primitives, which are explicitly designed to compose correctly with a separate header configuration.

> The generalizable lesson
>
> A header configuration can be entirely correct and still never reach the browser, for reasons that have nothing to do with the headers themselves. The only way this was caught was by re-running the same live check this chapter already recommends, after every deployment, and treating an unexpected result as a signal to investigate rather than a caching inconvenience to wait out.

Real-World Case Study, the CSP Decision Revisited: a Hash

That Was Wrong From the Start

Figure 5: The Hash-to-Nonce Failure Chain

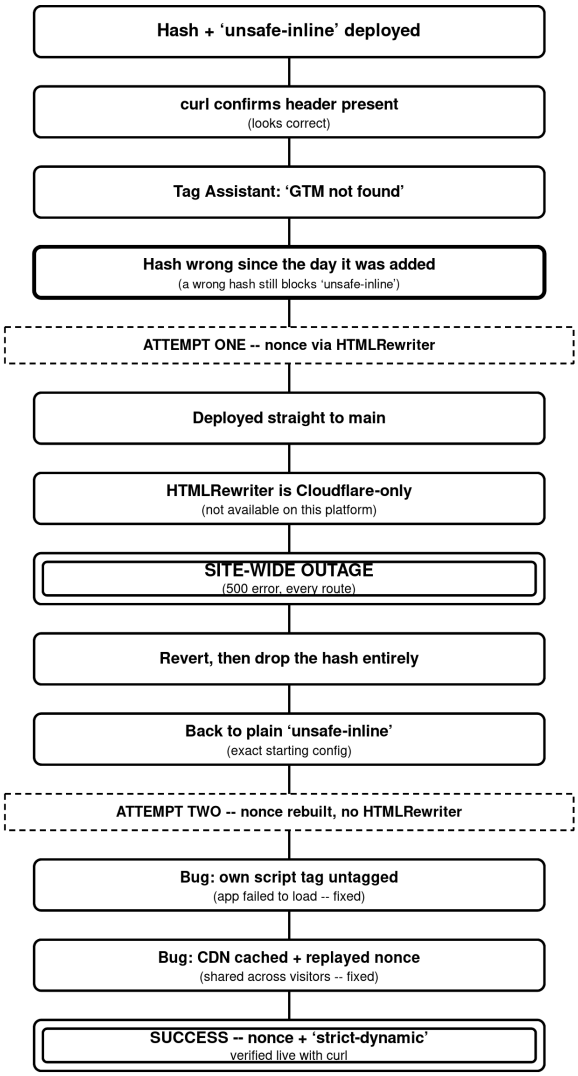

Figure 5. The hash-to-nonce failure chain

Two failures, one success, and the pattern worth noticing: every failure in this chain was caught by checking real, live behavior (Tag Assistant, the browser console, a repeated `curl`) -- never by re-reading a configuration file and trusting that it looked correct.

The Hash Fails

The hash-based fix described in Case Study 1 looked, by every available check, like a clean success: the header was present, the syntax was valid, and a fresh, cache-busted `curl -I` confirmed the exact hash value in the response. None of

that was enough. Checking the tag-manager vendor's own debug tool against the custom tags immediately after deploying the fix -- not waiting for a symptom to surface on its own -- turned up the real result: the container reported itself as "not found" on the live page, even though page views were still being recorded through a separate mechanism. The bootstrap script that was supposed to load the container was never running at all.

The cause was the hash itself. A hash freshly computed from the exact script in the live build did not match the value whitelisted in the CSP header, and checking the exact commit that had first introduced that value confirmed the mismatch went all the way back -- it was wrong from the start, not a previously-correct value broken by a later edit. This is the exact misconfiguration named earlier in this chapter -- the same CSP Level 3 precedence rule applies. The header looked perfect. The browser silently refused to run the one script the entire configuration existed to permit, and nothing in the HTTP response ever hinted at that. `curl` cannot see browser-side precedence rules; it can only confirm that a string is present.

> **IF THAT MECHANISM WAS HARD TO FOLLOW**
>
> A hash that does not exactly match your script does not just fail to help -- it silently breaks the very script it was supposed to protect, while the response header looks completely correct the whole time.

Nonce, Attempt One: Replacing the Hash Entirely, Not Patching It

The hash was abandoned outright, not kept alongside anything new. Reasoning that the tag-manager container could inject custom, unpredictable inline scripts at runtime -- something no static hash could ever cover -- this first attempt used a per-request nonce, generated fresh in the hosting platform's own edge middleware and written onto the bootstrap script tag in the HTML itself. That attempt made things worse before it made them better: the implementation used an HTML-rewriting API carried over from a different edge platform's documentation, without confirming it was actually available on the platform this project runs on. It was not. The moment it was pushed to the live branch, every

route touching that middleware began returning server errors -- a real, site-wide production outage caused by an unverified assumption about platform capability, deployed straight to production instead of a preview branch first. The revert was fast, but the near-miss was avoidable: the same assumption checked against real documentation five minutes earlier would have caught it before any user saw an error.

Research done immediately after the revert -- checking real documentation instead of repeating the earlier mistake -- concluded that the hosting platform's edge middleware could not rewrite an HTML response body at all, under any API, and that a per-request nonce was therefore not achievable on this architecture without a much larger change than the problem justified.

Having already caused one outage from an unverified assumption, the immediate, deliberate choice was not to attempt a second improvisation under pressure. The pragmatic fix was to step back to the simplest working configuration: drop the hash entirely, restore plain `'unsafe-inline'` alongside the existing explicit domain allowlist, and this time verify it properly -- pushed to a separate branch first, checked against the platform's own preview deployment using the tag-manager vendor's debugging export, and only merged into the live branch once the container and its business events were confirmed firing end to end. That was, at the time, a defensible and disciplined way to close out an incident.

> Attempt one ends here
>
> Nonce abandoned for now, hash abandoned for good -- plain `'unsafe-inline'` is once again the only thing authorizing the bootstrap script, the exact same configuration this whole incident started from. Nothing gained yet; nothing broken either.

Nonce, Attempt Two: Same Strategy, Rebuilt Implementation

What failed in attempt one was never the idea of using a nonce -- it was the specific, unverified API it was built on. This second attempt keeps the same goal

(a per-request nonce trusted via `'strict-dynamic'`) and replaces only the broken implementation detail, rather than abandoning the approach entirely.

It was also, it turned out, built on a conclusion that had not actually been tested -- only reasoned about, in the immediate aftermath of the outage. The pressure had not lifted -- the site's production stability came first, which is exactly why the hash was dropped and `'unsafe-inline'` restored before anything else was attempted. What changed for the second attempt was not the pressure; it was where the work happened. Revisited about an hour later, this time on an isolated experiment branch that never touched the live branch, the same nonce approach was rebuilt from scratch using nothing but standard `fetch`, `Headers`, `Response`, and string operations -- deliberately avoiding the non-existent HTML-rewriting API that had caused the outage. It worked. But only after two further problems surfaced, neither of which had anything to do with the tag-manager container itself.

Bug one: the app's own bundle never hydrated

The browser console was explicit about why: the `'strict-dynamic'` keyword disables `'self'`-based allowlisting entirely, for every script tag, not only third-party ones. The nonce had only been added to the tag-manager's own bootstrap script -- but the site's own entry script tag was just as much a static `<script>` element, and under `'strict-dynamic'` it needed the exact same treatment. Without it, the application never hydrated at all.

> **HYDRATION, DEFINED**
>
> Hydration is the step where the site's own JavaScript takes over an already-rendered page and makes it interactive -- attaching click handlers, enabling client-side navigation, running the app's actual logic. This project prerenders each route to real, visible HTML ahead of time, so a blocked entry script does not produce a blank page -- the prerendered markup is still there, untouched, and looks complete. What's missing is everything the JavaScript was supposed to add on top of it: no click gets tracked, no route change works, nothing built on the app's own code runs, even though the page visually looks finished.

That gap -- content present, behavior absent -- is exactly why this bug was consistent with everything the tag-manager container appeared to do (it loaded fine, since its own script did carry the nonce) while nothing the application itself was responsible for -- including the tag manager's own standard pageview tracking, which depends on the application's code running -- ever fired. Adding the nonce to that tag fixed it, but only after discovering that the build tool regenerated that specific script tag during its own transform and silently dropped the placeholder attribute in the process. The fix had to be reapplied one build step later than expected, and was only caught by inspecting the actual built output directly.

> **IF THAT MECHANISM WAS HARD TO FOLLOW**
>
> A nonce has to be added to every script tag that needs to run -- including your own site's code, not only the third-party script you were trying to fix -- and it is worth checking your build tool did not quietly strip it back out.

Bug two: the CDN was caching and replaying the nonce

Once both scripts had working nonces, a live check turned up a response carrying an old, already-broken version of the page -- complete with the stale, blocked script tag -- served from cache, tens of minutes after the fix had shipped. The response's own cache-control header had been set correctly to prevent caching, but that header only governs the requesting browser. The hosting platform's edge network honors a separate, platform-specific header for its own caching layer, and without it, the edge had been caching and replaying these responses regardless of what the browser-facing header said. A nonce that gets cached and replayed is shared across every visitor who hits that cached copy, which defeats the entire purpose of using a nonce in the first place. Adding the platform's own cache-control header alongside the standard one resolved it, confirmed by checking that the response's cache status read as a fresh miss on every request and that the nonce value was different every time.

IF THAT MECHANISM WAS HARD TO FOLLOW

The setting that stops your own browser from caching a page is not the same setting that stops a CDN from caching it, and a security value that must never repeat needs both.

Figure 6: Nonce Implementation Checklist

Both bugs above came from finding out the hard way. Figure 6 is the same approach, done correctly the first time -- the full code walkthrough follows in "The right approach," below.

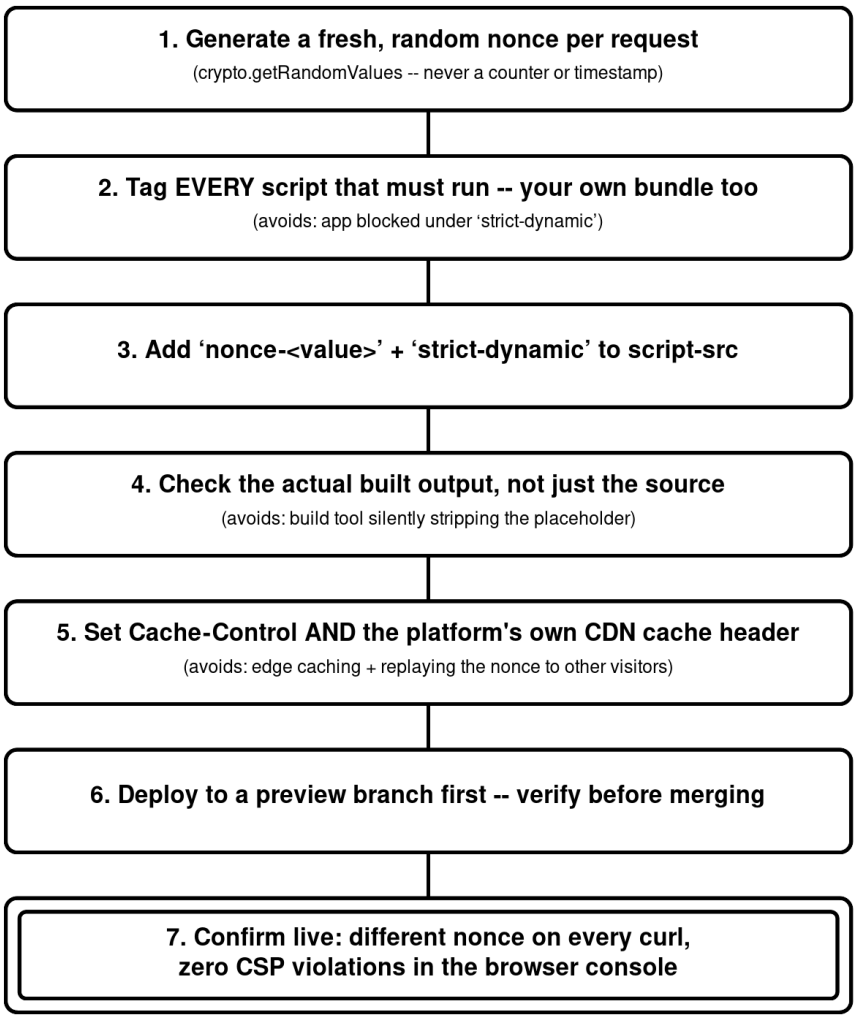

Figure 6. Nonce implementation checklist -- built to avoid the tag and caching bugs from the first attempt

Final Verification

With both bugs fixed, the tag-manager container's standard tags and its custom, dynamically-injected tags were confirmed firing correctly, across multiple pages, on genuinely fresh per-request responses. The fix was merged to the live branch only after that confirmation, and the same narrow header check this chapter recommends throughout was run one more time against production to close out the incident:

Live verification

```
curl -I https://www.example.com | grep -Ei 'strict-transport-security|content-
security-policy|x-frame-options|referrer-policy|x-content-type-options|permissions-
policy'
```

```
content-security-policy: default-src 'self'; base-uri 'self'; object-src 'none';
frame-ancestors 'none'; form-action 'self'; script-src 'self' 'nonce-
hDA69qMV0cpD+hzF7tA2Tg==' 'strict-dynamic' https: 'unsafe-inline'
https://www.googletagmanager.com https://www.google-analytics.com; style-src 'self'
'unsafe-inline'; img-src 'self' data: https://www.googletagmanager.com
https://www.google-analytics.com; connect-src 'self'
https://www.googletagmanager.com https://www.google-analytics.com
https://region1.google-analytics.com; frame-src 'self'
https://www.googletagmanager.com; font-src 'self' data:; upgrade-insecure-requests
permissions-policy: camera=(), microphone=(), geolocation=()
referrer-policy: strict-origin-when-cross-origin
strict-transport-security: max-age=63072000; includeSubDomains
x-content-type-options: nosniff
x-frame-options: DENY
```

All six baseline headers present, with `script-src` now carrying a live nonce and `'strict-dynamic'` instead of `'unsafe-inline'` doing the real work -- and, checked separately by repeating this same request several times in a row, a different nonce value on every single response. That last part is the one number in this whole chapter that actually matters: a nonce that does not change is not a nonce.

Three Validation Blind Spots

Stacked on top of each other across this whole incident, and all three are worth carrying forward.

1. A header can be textually present and completely inert at the same time, because of precedence rules inside the policy language itself that no amount of header inspection can reveal -- only checking that the dependent feature actually works will catch that.
2. An assumption about what a hosting platform supports is not a fact until it is checked against a real, working implementation, and that check should happen before a change reaches the branch serving live traffic, not after it causes an outage.
3. And the one easiest to miss: a negative conclusion deserves exactly the same scrutiny as a positive one. "This platform can't do that" was accepted here on the same weak footing that "this hash is correct" was accepted earlier in this same incident -- reasoned under pressure, not verified against a working test -- and it was wrong for the same underlying reason.

> "The dependent feature," defined
>
> Whatever functionality the header exists to protect or enable -- not the header string itself. In this incident it was concrete, twice over: first, the tag-manager container script itself (`gtm.js`) never loading at all, which the wrong hash silently caused while the header text still looked perfect -- not a subset of its tags failing, the entire container. The app's own `dataLayer.push()` calls for events like `contact_email_click` kept appearing to "work" throughout, but that was just the site's own code writing to a plain array; it says nothing about GTM, since `dataLayer.push()` runs identically whether or not the container that reads it ever loads. Second, the site's own application actually hydrating in the browser, which the missing entry-bundle nonce silently broke the same way. Neither failure showed up in a `curl` response.

The discipline this chapter keeps returning to is not "verify configuration before shipping it." It is "verify the actual end-to-end behavior before believing anything about it, including your own conclusion that something cannot be done."

The Right Approach: Implementing a CSP Nonce Without Repeating These Bugs

The two bugs above were found by accident, after shipping. This section is the same approach written the other way around -- what to put in each file before deploying, so neither bug has a chance to happen.

Step 1 -- Generate the Nonce Server-Side, Once Per Request

This runs in edge middleware, not in any static file, because a value baked into a build can never be different on every request. `npm run build` runs exactly once per deployment -- whatever it writes into `dist/`, including any string that looks like a nonce, is now a fixed, static file served identically to every visitor until the next deploy. Baking a nonce into a build output does not make it invalid syntactically; the CSP header would still parse and the page would still load. It simply stops being a nonce in the sense that matters. Middleware avoids this because it runs fresh on every incoming request, at the edge, after the build already happened.

File: foliovistabooks/middleware.js

```
function generateNonce() {
  const bytes = new Uint8Array(16);
  crypto.getRandomValues(bytes);
  let binary = "";
  for (let i = 0; i < bytes.length; i++) {
    binary += String.fromCharCode(bytes[i]);
  }
  return btoa(binary);
}
```

Step 2 -- Tag Every Script Tag in the Source Template

Not just the third-party one. This is the exact line that was missing the first time: the site's own entry bundle needs the same placeholder as the tag-manager script, or `'strict-dynamic'` blocks it too.

File: foliovistabooks/index.html

```
<!-- third-party bootstrap script -->
<script nonce="__CSP_NONCE__">
  (function(w, d, s, l, i) { /* GTM loader */ })(window, document, "script",
"dataLayer", "GTM-XXXXXXX");
</script>

<!-- the site's own entry bundle -- easy to forget, and the actual cause
     of the first outage's follow-up bug -->
<script type="module" nonce="__CSP_NONCE__" src="/src/main.jsx"></script>
```

Both carry the same literal placeholder string, `__CSP_NONCE__` -- not a real nonce value yet. That gets filled in once, per request, in Step 5.

Step 3 -- Put the Nonce and strict-dynamic in the CSP Header

File: foliovistabooks/middleware.js

```
function buildCsp(nonce) {
  return [
    "default-src 'self'",
    "base-uri 'self'",
    "object-src 'none'",
    "frame-ancestors 'none'",
    "form-action 'self'",
    `script-src 'self' 'nonce-${nonce}' 'strict-dynamic' https: 'unsafe-inline'
https://www.googletagmanager.com https://www.google-analytics.com`,
    "style-src 'self' 'unsafe-inline'",
    "img-src 'self' data: https://www.googletagmanager.com https://www.google-
analytics.com",
    "connect-src 'self' https://www.googletagmanager.com https://www.google-
analytics.com https://region1.google-analytics.com",
    "frame-src 'self' https://www.googletagmanager.com",
    "font-src 'self' data:",
    "upgrade-insecure-requests"
  ].join("; ");
}
```

Step 4 -- Re-Check the Placeholder Survives the Build

On every document the build produces, not only the source template. This project prerenders six separate route documents from one template, and the build tool regenerates the entry script tag during its own transform -- silently dropping the placeholder on that one tag, on every one of those six documents, unless it is put back. This loop is what actually touches "each document in the application":

File: foliovistabooks/scripts/prerender.mjs

```
const routes = [
  "/",
  "/books",
  "/books/operational-bug-bounty-fieldwork",
  "/books/practical-layered-security-for-small-platforms",
  "/contact",
  "/privacy-and-terms"
];

function buildRouteHtml(template, { appHtml, headTags }) {
  return template
    .replace(/<title>[\s\S]*?<\/title>/i, "")
    .replace(/<meta\s+name="description"[\s\S]*?>/i, "")
    .replace(/<div id="root"><\/div>/i, `<div id="root">${appHtml}</div>`)
    .replace("</head>", `${headTags}\n  </head>`)
    // Vite regenerates the entry module script tag during its own build
    // transform and drops unrecognized attributes in the process, so the
    // nonce placeholder from source index.html never survives `vite build`
    // on this tag specifically. Re-inject it here, once, so every route's
    // generated document gets it back.
    .replace(
      /<script type="module" crossorigin src="([^"]+)"><\/script>/,
      '<script type="module" crossorigin nonce="__CSP_NONCE__" src="$1"></script>'
    );
}

for (const routePath of routes) {
  const routeHtml = buildRouteHtml(template, render(routePath));
  // ...written to each route's own output file
}
```

Step 5 -- Replace the Placeholder and Stop Both Caches

At request time, replace the placeholder with the real nonce, and stop both the browser and the CDN from caching the result. This is the one function that ties every earlier step together.

File: foliovistabooks/middleware.js

```
export default async function middleware(request) {
  if (!isHtmlDocumentRequest(request)) {
    return fetch(request);
  }

  const response = await fetch(request);
  const contentType = response.headers.get("content-type") ?? "";
  if (!contentType.includes("text/html")) {
    return response;
  }

  const nonce = generateNonce();
  const html = await response.text();

  // Fills in every "__CSP_NONCE__" placeholder in this document -- both
  // script tags from Steps 2 and 4 -- with the one nonce generated above.
  const rewritten = html.replaceAll("__CSP_NONCE__", nonce);

  const headers = new Headers(response.headers);
  headers.set("Content-Security-Policy", buildCsp(nonce));
  // Cache-Control alone only governs the browser. The platform's own
  // CDN-Cache-Control header is what actually stops the edge layer from
  // caching and replaying this exact nonce to other visitors.
  headers.set("Cache-Control", "private, no-store");
  headers.set("CDN-Cache-Control", "no-store");
  headers.set("Vercel-CDN-Cache-Control", "no-store");
  headers.delete("content-length");

  return new Response(rewritten, {
    status: response.status,
    statusText: response.statusText,
    headers
  });
}
```

Steps 6 and 7 -- Process, Not Code

Stay the same as everywhere else in this chapter: push to a preview branch first, and confirm live with a repeated `curl` before merging -- covered next, in Validation.

Validation

The quickest validation step is:

```
curl -I https://www.example.com
```

Then confirm that the response includes the expected hardening headers. For a narrower check on the core headers discussed here:

```
curl -I https://www.example.com | grep -Ei 'strict-transport-security|content-security-policy|x-frame-options|referrer-policy|x-content-type-options|permissions-policy'
```

For this chapter, the operator should verify at minimum:

- `Strict-Transport-Security`
- `Content-Security-Policy`
- `X-Frame-Options`
- `Referrer-Policy`
- `X-Content-Type-Options`
- `Permissions-Policy`

The same check can also be done visually in browser DevTools: open the Network tab, reload the page, inspect the main document request, and review the response headers directly.

Validation should not stop at a single successful page load. In a stronger review pass, confirm that the expected headers also appear on representative redirects, missing pages, or other response paths that users and crawlers may still reach -- and as Case Study 1 shows, a policy can fail silently on every response path at once if the underlying routing configuration swallows it.

A Related Deployment Lesson

Browser-hardening work can fail to reach production even when the header policy itself is correct. In one instance, an invalid redirect source pattern in the deployment config blocked newer deployments entirely -- written with regex-style optional thinking, when the platform's simplified redirect configuration expects explicit path syntax instead:

`Invalid pattern`

```
/privacy/?
/terms/?
```

`Corrected, explicit route entries`

```
/privacy
/privacy/
/terms
/terms/
```

For a scored assessment, tools like Mozilla Observatory or securityheaders.com will rate your header set against current best practices and flag gaps you may have missed -- run your own site through one of these before publishing anything that references your header configuration as an example.

What curl Cannot See

None of these checks, including your own `curl`, can see inside a CSP's own precedence rules. A `script-src` that lists both a hash-source and `'unsafe-inline'` looks identical in a header dump whether the hash is correct or completely wrong, but the browser's behavior is entirely different in each case. Case Study 2 documents exactly this failure. Whenever a change touches an inline-script policy, validation must include confirming that the dependent functionality still runs in a real browser -- check the browser console for CSP violation messages, or use the relevant vendor's own debugging tool if one exists.

If one or more headers are missing, the next step is not panic. Identify which defensive behavior is currently absent and restore it deliberately. If every header

is missing at once despite a correct configuration file, suspect the routing layer itself, not the header values -- as Case Study 1 illustrates.

Result and What to Remember

Result

The browser becomes a more active enforcement layer instead of a passive rendering surface, and the reader understands HSTS correctly as one strong layer rather than the whole answer.

What to Remember

Security headers are behavioral constraints, not decoration. They tell the browser how to enforce trust, not just where to connect.

HSTS keeps the browser on HTTPS. It does not keep the browser from running bad scripts, framing your page, or leaking data. That is the work of the other headers.

The practical test is not whether you have HSTS -- it is whether HSTS is accompanied by content restrictions, frame restrictions, and safer rendering defaults. Treating transport security and browser behavior security as the same thing is the mistake this chapter is designed to prevent.

Nor is the practical test whether your configuration file lists the right header names. As Case Study 1 shows, a correct configuration can still fail to reach the browser. The only reliable test is what the live response actually contains, checked after every deployment.

Nor is it enough that the live response contains the right header text. A CSP directive can be textually perfect and still silently disable itself through its own precedence rules -- invisible to `curl`, invisible to a scanner, invisible to anything short of confirming the actual dependent feature still works in a real browser. And

a fix for one of these problems is not automatically safe to ship straight to production: an unverified assumption about what the hosting platform supports is a live-outage risk of its own, which is why Case Study 2's fix was tested on a preview branch before it ever touched the branch serving real users.

The lesson underneath the lesson

A conclusion that something cannot be done deserves exactly as much scrutiny as a claim that something has been fixed. Both are just assertions until checked against a real, working test. The discipline this chapter is built around applies to your own reasoning as much as to any configuration file: verify the live, actual, end-to-end behavior -- never the configuration's appearance, never an assumption about the platform underneath it, and never your own prior conclusion, simply because reaching it once already cost something to learn.