PRACTICAL LAYERED SECURITY FOR SMALL PLATFORMS

# Chapter 4 Code Appendix

Baseline headers, and a per-request CSP nonce. A standalone, generic reference for the two configurations described in Chapter 4's case studies -- every identifier below is a placeholder. There is no real domain, container ID, or measurement ID in this file. Replace the placeholders with your own values before use.

STATUS

**Pre-publish review copy**

Companion reference to Chapter 4, sanitized for reuse on any project.

PAIRS WITH

**Chapter 4**

Case Study 1 (baseline headers) and Case Study 2 (the nonce pattern).

PREPARED

**2026-07-09**

Publication-style local pass.

## Part 1: Static Baseline Headers (Vercel vercel.json)

This covers the six-header baseline plus the redirect/rewrite pattern from Chapter 4's first case study. It assumes a static or prerendered single-page app with no per-request CSP nonce -- suitable for a site that does not need to protect an inline script with anything stronger than a hash or `'unsafe-inline'`.

`vercel.json`

```
{
  "framework": "vite",
  "buildCommand": "npm run build",
  "outputDirectory": "dist",
  "headers": [
    {
      "source": "/(.*)",
      "headers": [
        {
          "key": "Content-Security-Policy",
          "value": "default-src 'self'; base-uri 'self'; object-src 'none'; frame-
ancestors 'none'; form-action 'self'; script-src 'self' 'unsafe-inline'
https://your-cdn-or-tag-manager.example.com; style-src 'self' 'unsafe-inline'; img-
src 'self' data: https://your-cdn-or-tag-manager.example.com; connect-src 'self'
https://your-analytics.example.com; frame-src 'self'; font-src 'self' data:;
upgrade-insecure-requests"
        },
        { "key": "X-Content-Type-Options", "value": "nosniff" },
        { "key": "X-Frame-Options", "value": "DENY" },
        { "key": "Referrer-Policy", "value": "strict-origin-when-cross-origin" },
        { "key": "Permissions-Policy", "value": "camera=(), microphone=(),
geolocation=()" },
        { "key": "Strict-Transport-Security", "value": "max-age=63072000;
includeSubDomains" }
      ]
    }
  ],
  "redirects": [
    { "source": "/old-page", "destination": "/new-page", "permanent": true },
    { "source": "/old-page/", "destination": "/new-page", "permanent": true }
  ],
  "rewrites": [
    { "source": "/(.*)", "destination": "/index.html" }
  ]
}
```

Notes

- `redirects` uses explicit `source` / `destination` pairs, not a regex-style optional path like `/old-page/?`. That pattern is not valid in this field and will

either be rejected or silently misbehave -- list the with-and-without-trailing-slash variants separately, as shown.

- `rewrites` is what a single-page app needs for client-side routing, and it checks the filesystem first by default, so real static assets are still served directly rather than being rewritten to `index.html`.
- Do not combine this `headers` block with a legacy `routes` array in the same file. A `routes` array that ends in a catch-all rewrite with no `"continue": true` can silently prevent `headers` from ever being applied, with no error and no warning -- the exact failure mode in Chapter 4's first case study.
- `preload` is deliberately not included in the `Strict-Transport-Security` value above. Add it only as a separate, deliberate decision -- submission to browser preload lists is difficult to reverse.
- Never add `X-XSS-Protection`. It is deprecated and can introduce its own vulnerabilities in the older browsers that still honor it.

## Part 2: Per-Request CSP Nonce (Vercel Edge Middleware)

This is the pattern from Chapter 4's second case study, for a case where a third-party script (a tag manager, a widget) injects its own inline scripts at runtime and a static hash cannot cover them. It requires two things beyond the middleware file itself:

1. The entry point of any static script tag that must be trusted -- including your own application's bundle, not only the third-party script -- needs a `nonce="__CSP_NONCE__"` placeholder baked into the HTML at build time.
2. If your build tool regenerates that script tag during its own bundling step (as Vite does for its module entry tag), confirm the placeholder survives by inspecting the actual built output, not just your source template. If it does not survive, reapply it in whatever post-build step produces your final HTML.

## Example index.html entry points

index.html

```
<script nonce="__CSP_NONCE__">
  /* first-party bootstrap script that must be trusted */
</script>

<script type="module" nonce="__CSP_NONCE__" src="/assets/your-app-bundle.js">
</script>
```

## The full middleware implementation

middleware.js

```
const STATIC_PREFIXES = ["/assets/", "/icons/", "/fonts/"];
const STATIC_PATHS = new Set(["/favicon.ico", "/robots.txt", "/sitemap.xml"]);

function isHtmlDocumentRequest(request) {
  if (request.method !== "GET" && request.method !== "HEAD") return false;

  const { pathname } = new URL(request.url);

  if (STATIC_PATHS.has(pathname)) return false;
  if (STATIC_PREFIXES.some((prefix) => pathname.startsWith(prefix))) return false;

  const extensionMatch = pathname.match(/\.([a-z0-9]+)$/i);
  if (extensionMatch) {
    return extensionMatch[1].toLowerCase() === "html";
  }

  return true;
}

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);
}

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://your-tag-manager.example.com`,
    "style-src 'self' 'unsafe-inline'",
    "img-src 'self' data: https://your-tag-manager.example.com",
    "connect-src 'self' https://your-analytics.example.com",
```

```
    "frame-src 'self'",
    "font-src 'self' data:",
    "upgrade-insecure-requests"
  ].join("; ");
}

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();
  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 requesting browser. The CDN/edge
  // layer in front of the origin honors its own separate cache-control
  // headers, and without setting them explicitly, a nonce can end up
  // cached and replayed to many different visitors -- which defeats the
  // entire purpose of using one. Set all three.
  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
  });
}

export const config = {
  runtime: "edge",
  matcher: ["/((?!assets/|icons/|fonts/).*)"]
};
```

## Verification Checklist

Before trusting this pattern is actually working:

1. Request the same page several times in a row and confirm the `nonce-...` value in the `Content-Security-Policy` response header is different on every single request. If it repeats, the CDN is still caching the response and the nonce is not actually protecting anything.
2. Confirm the response's cache status header reads as a fresh miss, not a hit, on repeated requests.
3. Open the browser console and confirm there are no `Refused to execute inline script` or `Refused to load the script` violation messages -- including for your own application bundle, not only the third-party script.
4. If the third-party script has its own debugging/diagnostic tool, use it to confirm the specific dynamically-injected functionality (not just the base script) is actually running end to end, not just that the base container loaded.

> Before merging to production
>
> Do not merge a version of this pattern to a production branch without completing all four checks against a preview deployment first. This exact pattern caused a real, site-wide outage in an earlier attempt that used an HTML-rewriting API not actually available on this runtime -- verified correctness on a preview branch, not confidence in the approach, is what makes this safe to ship.