ChatGPT

Code-Level Hardening and Privacy for a Static Portfolio Site

Executive summary

This report focuses **exclusively on code-level measures** inside your static portfolio (HTML/CSS/JS/assets and build-time hygiene scripts) and **explicitly excludes** hosting/CDN/DNS configuration. Within that scope, the single most urgent issue is that your intended Content Security Policy is **not being applied** because the markup in your HTML files uses an invalid CSP meta tag pattern (it literally contains `\1…\3` instead of a valid `<meta http-equiv="Content-Security-Policy" …>`). That means your current runtime protections against content injection and XSS are significantly weaker than you likely expect, even though your pages already include useful privacy controls like `Referrer-Policy` via `<meta name="referrer" content="no-referrer">`. [1]

For a **public static portfolio**, confidentiality is fundamentally limited (anyone can download the HTML/JS/assets); therefore “make it as private as possible” at the code level mainly means:

- minimize third-party requests and identifiers (fonts/media/analytics), because those leak visitor IP/user agent and enable cross-site correlation, which is central to web tracking and fingerprinting. [2]
- harden the browser execution environment with **CSP (strict, hash-based)**, avoidance of dangerous DOM sinks (e.g., `innerHTML`), and (optionally) Trusted Types to prevent future DOM-XSS regressions. [3]
- treat client storage (localStorage/sessionStorage) as **untrusted and non-confidential**, avoid storing secrets, and validate stored values before using them. [4]
- implement contact/email patterns that raise the cost of harvesting **without claiming perfect protection** (bots can execute JS and OCR images); the best code-only posture is usually “don’t publish the email address at all” and offer a contact mechanism that does not expose it. [5]
- automate asset hygiene (EXIF/metadata stripping) and enforce guardrails with linting and CI checks.

Because you requested rigor: client-side encryption and “JavaScript password protection” can be useful as **obfuscation** or as “reduce incidental disclosure,” but they do **not** provide strong confidentiality against a determined attacker—especially if an attacker can modify the JavaScript delivered to the user. Academic and practitioner sources document both misuse risk and attack models where injected JS captures plaintext before encryption. [6]

Scope and assumptions

This report applies to:

- HTML/CSS/JS changes (including `<meta>`-based policies and DOM coding patterns).
- Build-time scripting that generates JSON content (`update-repo-descriptions.py` / `repos.json`) and local developer workflows (pre-commit hooks).

- CI-as-code (example GitHub Actions YAML) for linting/scanning **without discussing hosting or CDN settings**.

Assumptions (because you asked not to ask clarifying questions):

- The site is a classic static site (no server-side runtime) with multiple `.html` pages and a shared `app.js` + `styles.css`.
- You control the repository and can add dev dependencies (Node-based linters) if desired, but you may also want “no-build-tool” options.
- You prefer a privacy-first posture (no analytics/tracking) and a conservative browser surface.

Important boundary: several high-value protections (HSTS, many security headers) are strongest as **HTTP response headers**. A CSP can be delivered via `<meta http-equiv="Content-Security-Policy">` in code, but many other headers cannot be reliably emulated via HTML meta tags. This report uses code-level alternatives and flags limitations. [7]

Threat models and security objectives

Threat models

Casual visitors
Risks: opportunistic scraping of email/contact info, automated scanning for obvious vulnerabilities, and accidental privacy leaks via third-party resources.

Targeted attackers
Risks: injection of malicious scripts via supply chain, compromised third-party resources, or content injection bugs; tampering with assets (e.g., swapping JS) to exfiltrate user-entered data; “water-holing” where your site is used to target specific visitors. XSS is a core browser-side risk because it lets attacker-controlled code run “as your site.” [8]

Browser fingerprinting and tracking
Risks: third parties (fonts/CDNs/widgets) and embedded scripts can observe visitors and create stable identifiers from browser/device characteristics. Fingerprinting remains possible even without cookies. [2]

Email harvesting
Risks: bots scrape HTML, execute JS, or use OCR on rendered output to extract addresses at scale; obfuscation typically only increases cost, not prevent. [9]

Confidentiality and privacy goals

- Avoid disclosing visitor data to third parties unless the visitor intentionally navigates there (minimize third-party fonts/media, tracking scripts, external form endpoints).
- Avoid creating durable client identifiers (cookies, localStorage IDs, fingerprinting scripts).
- Minimize referrer leakage (already partially addressed with `no-referrer`). [10]
- Reduce the likelihood of email harvesting at scale.

Integrity and availability goals (code-only)

- Integrity: prevent script/content injection from turning into code execution (CSP + safe DOM APIs + dependency pinning + build-time sanitation). CSP is explicitly designed to reduce content injection impact, particularly XSS. [11]
- Availability: ensure the code does not DoS clients (avoid costly loops, avoid oversized assets), and reduce reliance on fragile third-party resources.

Usability and design trade-offs

- A strict CSP can break inline scripts and some third-party resources until refactored or hashed. [12]
- Self-hosting fonts/media improves privacy but can increase repo size and initial load time.
- Aggressive email obfuscation can harm accessibility and still be defeated by advanced scrapers (JS execution or OCR).
- Client-side encryption adds complexity and can create false confidence; misuse risk is nontrivial. [13]

Code attack surface inventory in your site

This inventory is organized by surfaces you highlighted (and a few closely related ones that matter in this codebase). Each item also hints at "what can go wrong" under your threat models.

Inline scripts and CSP delivery

Your HTML pages include an inline bootstrap script (used for early theme/background/font preference application before render). Inline scripts materially raise CSP complexity: strict CSP guidance recommends nonce- or hash-based approaches to prevent execution of untrusted scripts. [14]

Separately, your CSP meta tag is malformed (it uses literal `\1 … \3`), so the browser likely ignores it. This is a high-impact gap because CSP is one of the strongest "second layer" protections against XSS. [15]

External resources (third-party fonts, third-party media)

Your pages load fonts from a third-party CSS endpoint (`api.fonts.coollabs.io`), and some pages load images/videos from `raw.githubusercontent.com`. Each such request reveals visitor network metadata to those third parties and creates correlation surfaces (timing, caching, request headers). Fingerprinting and tracking literature emphasizes that tracking can occur even without cookies, and third-party resource inclusion is a common enabler. [2]

Your site also contains a PayPal purchase form (in `product.html`), which is intentionally third-party on use; the key is to ensure it's user-initiated and not silently loading trackers.

DOM injection surfaces and client-side navigation

Your `app.js` performs client-side page transitions: it fetches an internal `.html`, parses it with `DOMParser.parseFromString()`, extracts `<main>`, and injects it into the live DOM. MDN notes that parsing via `DOMParser.parseFromString()` yields an inert document where scripts don't execute, but scripts and event handlers can run if inserted into the visible DOM. [16]

This is *not automatically unsafe* if you only insert trusted same-origin HTML you control, but it becomes a critical sink if any untrusted HTML ever enters the pipeline (e.g., future "markdown to HTML," user-generated content, or compromised build output).

Your code already uses safer patterns in many places (e.g., `textContent`) consistent with OWASP guidance for preventing DOM-based XSS. [17]

localStorage / sessionStorage

You store user preferences (theme/background/font/size) in localStorage. Web storage is convenient but must be treated as both **non-confidential** and **untrusted**:

- OWASP warns not to store sensitive information in localStorage, because authentication assumptions can be bypassed and a single XSS can steal or poison the stored data. [18]
- Any script running in your origin can read it; therefore storage safety depends heavily on preventing XSS. [19]

Service workers (currently absent, but high-impact if added)

If you add a service worker later, it can intercept network requests within its scope and return modified responses. MDN explicitly calls out that service workers can intercept requests and that user-controlled inputs influencing service worker registration can be an XSS vector. [20]

Forms and contact UX

You currently have (at least) a PayPal `<form action=…>` and email/contact UX in JS. Forms can leak data through:

- autocomplete storing sensitive entries locally
- accidental inclusion of third-party endpoints
- logging or analytics scripts capturing input

MDN documents the purpose and behavior of `autocomplete` as a browser hint. [21]

Metadata/EXIF in assets

Images (and sometimes videos) may carry metadata such as device model, timestamps, or location. MITRE CWE-212 highlights the risk class of sensitive metadata not being removed before sharing. [22]

Repo-scrape ingestion and generated JSON

Your `update-repo-descriptions.py` fetches GitHub HTML and extracts `<meta name="description" content="…">`, then writes `repos.json` that the frontend loads. This introduces:

- supply-chain/data ingestion fragility (HTML can change; remote data is untrusted)

- injection risk if you ever render descriptions as HTML (currently you mostly use `textContent`, which is good)

OWASP stresses safe sinks (use `textContent` instead of `innerHTML`) and that DOM insertion of untrusted data is a DOM-XSS vector. [23]

Code-level mitigations with implementation guidance

This section proposes concrete, implementable techniques per attack surface. Each technique includes rationale, threats mitigated, prerequisites, step-by-step implementation, compatibility notes, and usability trade-offs.

Repair and strengthen CSP using code-only delivery

Fix the malformed CSP meta tag

Rationale
CSP is designed to restrict what resources the browser may load and is a key mitigation against XSS and content injection. [24]

Threats mitigated
Targeted attackers attempting XSS/content injection; opportunistic injection introduced by future code changes.

Prerequisites
Edit all HTML pages (or your template) to correct the CSP meta tag.

Implementation steps
Replace your malformed CSP meta line with a valid meta http-equiv CSP:

```
<meta http-equiv="Content-Security-Policy"
      content="default-src 'self'; base-uri 'self'; object-src 'none'">
```

MDN explicitly documents that `http-equiv="content-security-policy"` can be used in HTML to define CSP for a page. [25]

Compatibility and trade-offs
* Meta-delivered CSP is supported, but many teams prefer response headers for full control; that configuration aspect is out of scope here. [26]

Move toward a “strict CSP” (hash-based) and remove inline script execution

Rationale
MDN and the CSP3 specification describe “strict CSP” approaches (nonce/hash-based) as effective XSS mitigation compared to broad allowlists. [14]

Threats mitigated
XSS and malicious script execution; accidental introduction of unsafe inline scripts.

Prerequisites
Decide whether to eliminate inline scripts or allow a minimal one via hashes.

Implementation option A (recommended): remove inline scripts entirely
1. Move the inline “bootstrap” theme script from `<head>` into an external file, e.g., `bootstrap.js`. 2. Load it synchronously in `<head>` before CSS so it still runs early but doesn’t require inline CSP exceptions:

```
<script src="/bootstrap.js"></script>
<link rel="stylesheet" href="/styles.css">
<script src="/app.js" defer></script>
```

1. Then your CSP can omit any inline allowances:

```
<meta http-equiv="Content-Security-Policy"
      content="
        default-src 'self';
        base-uri 'self';
        object-src 'none';
        script-src 'self';
        style-src 'self';
        img-src 'self' data:;
        font-src 'self';
        media-src 'self';
        connect-src 'self';
        form-action 'self';
      ">
```

Trade-offs
* Adds one extra request (`bootstrap.js`) unless you later bundle. * Simplifies CSP dramatically; fewer future footguns.

Implementation option B: keep a minimal inline script and allow it via hash
If you must keep inline bootstrap logic, allow only that exact script using a CSP hash source expression (OWASP provides hash-based policy examples). [27]

Step-by-step
1. Minimize and freeze the exact inline script bytes (whitespace changes change the hash).
2. Compute the hash. Example using OpenSSL (pattern from MDN’s SRI hash generation; same concept for CSP hashing): [28]

```
python - << 'PY'
import hashlib, base64, pathlib, re
html = pathlib.Path("index.html").read_text(encoding="utf-8")
m = re.search(r"<script>(.*?)</script>", html, re.S)
script = m.group(1).strip().encode("utf-8")
h = base64.b64encode(hashlib.sha256(script).digest()).decode()
print("sha256-" + h)
PY
```

1. Put that hash into `script-src` and remove `'unsafe-inline'` entirely:

```
<meta http-equiv="Content-Security-Policy"
      content="
        default-src 'self';
        base-uri 'self';
        object-src 'none';
        script-src 'self' 'sha256-REPLACE_WITH_OUTPUT_FROM_SCRIPT';
      ">
```

Compatibility and trade-offs
* Works well for static sites but requires hash updates whenever the inline script changes. * If you later add additional inline scripts, you must hash each or you lose strictness.

Add Trusted Types as a "future mistake" guardrail (optional)

Rationale
Trusted Types is designed to prevent DOM-based XSS by restricting assignments into dangerous sinks (e.g., `innerHTML`) unless the value came from an approved policy. The W3C specification and MDN describe how it targets DOM XSS sink functions. [29]

Threats mitigated
Future regressions where a developer introduces `innerHTML` or similar sinks with untrusted input.

Prerequisites
Your CSP must include Trusted Types directives; browser support is not universal.

Implementation steps
1. Add to CSP:

```
<meta http-equiv="Content-Security-Policy"
      content="
        ...;
        require-trusted-types-for 'script';
        trusted-types default;
      ">
```

MDN notes `require-trusted-types-for` exists but has limited availability; treat this as defense-in-depth rather than guaranteed. [30]

1. In JS, avoid dangerous sinks entirely; if absolutely necessary, define a Trusted Types policy and sanitize (e.g., with DOMPurify). MDN shows the API conceptually. [31]

Trade-offs
* Not supported everywhere; can break functionality if you rely on string-to-HTML sinks without a policy. * Adds maintenance complexity but can prevent accidental XSS introduction.

Reduce third-party exposure by self-hosting assets and tightening allowed origins

Self-host fonts and remove third-party font CSS endpoints

Rationale
Every third-party request leaks visitor metadata and enables cross-site correlation. Fingerprinting/tracking research and EFF's Cover Your Tracks emphasize that identification can happen without cookies; minimizing third-party dependencies is a high-value privacy control. [2]

Threats mitigated
Fingerprinting/correlation by third parties; CDN compromise risk; fragility when external endpoints change.

Prerequisites
Ability to download and commit WOFF2 font files, or switch to system fonts.

Implementation steps
1. Download the required font files (WOFF2 preferred).
2. Add `fonts/` directory, e.g.:

```
fonts/
  JetBrainsMono-Variable.woff2
  SourceCodePro-Variable.woff2
```

1. Add `@font-face` rules in `styles.css`:

```
@font-face {
  font-family: "JetBrains Mono";
  src: url("/fonts/JetBrainsMono-Variable.woff2") format("woff2");
  font-weight: 100 800;
  font-style: normal;
  font-display: swap;
}
```

1. Remove the external `<link rel="stylesheet" href="...fonts...">` from HTML.
2. Tighten CSP by removing external font/style origins.

Compatibility and trade-offs
* Larger first-party payload; mitigated by compression and caching (configuration-side). * Better determinism; fewer external failures.

Self-host media assets currently served from third-party origins

Rationale
Hosting images/videos on third-party domains creates third-party requests. “Lazy-loading” helps by reducing requests until needed, but the cleanest path is to serve them first-party. MDN’s caching guidance also notes service workers and CDNs act as managed caches; first-party control helps consistency. [32]

Threats mitigated
Third-party tracking/correlation; dependency fragility.

Implementation steps
1. Mirror critical screenshots/videos into your repo under e.g. `media/` or `images/`. 2. Replace `src="https://raw.githubusercontent.com/...` with `src="/media/..."` 3. Keep `loading="lazy"` and `decoding="async"` on images:

```
<img src="/images/project.png" alt="…" loading="lazy" decoding="async">
```

Trade-offs
* Repo size increase; consider replacing heavy videos with a click-to-load pattern to preserve performance.

Use SRI for any remaining third-party JS/CSS and understand the caveats

Rationale
SRI lets browsers verify that external resources match an expected cryptographic hash; if modified, the browser refuses to load them. [33]

Threats mitigated
Supply-chain tampering at the CDN/resource provider.

Prerequisites
You must have a stable, versioned asset URL. The CDN must support CORS for SRI to work (MDN explicitly notes CDNs must set `Access-Control-Allow-Origin`). [28]

Implementation steps (example for a third-party script)
1. Generate an SRI hash (MDN provides an OpenSSL pattern): [28]

```
curl -s https://example-cdn.invalid/library.min.js | \
  openssl dgst -sha384 -binary | \
  openssl base64 -A
```

1. Add `integrity` + `crossorigin`:

```
<script
  src="https://example-cdn.invalid/library.min.js"
  integrity="sha384-PASTE_HASH_HERE"
  crossorigin="anonymous"></script>
```

SRI caveats (important)
Any upstream change breaks loads until you update the hash (a feature, not a bug). [28]
For dynamic endpoints (like "Google Fonts style URLs" or any service that varies output), SRI becomes brittle. In such cases, self-hosting is typically better for privacy and reliability.

DOM XSS prevention and safe HTML manipulation patterns

Prefer safe DOM APIs and enforce "no dangerous sinks" with lint rules

Rationale
OWASP's DOM-based XSS and HTML5 security guidance stresses using safe sinks like `textContent` rather than `innerHTML`, and treating data as data (not markup). [17]

Threats mitigated
XSS via DOM injection.

Implementation pattern (safe rendering)
Instead of:

```
container.innerHTML = `<p>${userString}</p>`; // unsafe
```

Use:

```
const p = document.createElement("p");
p.textContent = userString;          // safe sink
container.replaceChildren(p);
```

Compatible and trade-offs
* Slightly more verbose than `innerHTML`, but far safer and easier to audit.

Tooling guardrail (recommended)
Use `eslint-plugin-no-unsanitized`, which explicitly disallows unsafe assignments/calls like `innerHTML` without sanitization. [34]

Example `.eslintrc.cjs` excerpt:

```
module.exports = {
  env: { browser: true, es2022: true },
  extends: ["eslint:recommended"],
  plugins: ["no-unsanitized"],
  rules: {
    "no-unsanitized/method": "error",
    "no-unsanitized/property": "error"
  }
};
```

Harden your client-side navigation that injects `<main>` from fetched HTML

Rationale
Your SPA-like navigation uses `DOMParser.parseFromString()`. MDN notes this produces an inert document where scripts and events don't run, but once inserted into the visible DOM, scripts/event handlers *can* run. [16]

Threats mitigated
If an attacker ever causes untrusted HTML to be fetched/inserted (future feature drift, compromised build artifacts), this becomes a DOM-XSS sink.

Prerequisites
Willingness to enforce a rule that fetched page fragments contain no scripts and no inline event handler attributes.

Implementation steps (defense-in-depth sanitizer before insertion)
Immediately after extracting `newMain` and before `main.replaceWith(newMain)`, remove scripts and inline event handlers:

```
function stripActiveContent(root) {
  // Remove scripts
  root.querySelectorAll("script").forEach((s) => s.remove());

  // Remove inline event handlers like onclick=...
  root.querySelectorAll("*").forEach((el) => {
    for (const attr of Array.from(el.attributes)) {
      if (attr.name.toLowerCase().startsWith("on")) {
        el.removeAttribute(attr.name);
      }
    }
  });
}

// After: const newMain = doc.querySelector('main');
stripActiveContent(newMain);
```

Compatibility and trade-offs
* Compatible across modern browsers. * If you intentionally rely on inline event handlers in page content, this will break them (that's typically a positive constraint under a strict CSP).

Additionally, ensure you keep the existing "same-origin + `.html` only" navigation guard; OWASP's testing guidance notes that letting user-controlled inputs influence resource URLs can lead to injection scenarios. [35]

Storage hygiene: localStorage usage patterns that stay privacy-first

Rationale
OWASP's HTML5 Security Cheat Sheet emphasizes that localStorage should not store sensitive data, should not be trusted, and can be stolen/poisoned via XSS. [18]

Threats mitigated
Privacy leakage (persistent identifiers), attacker persistence via poisoned storage values, and "stored XSS-like" behavior when unsafe sinks are used.

Implementation steps
1. Keep storage keyspace minimal and preference-only.
2. Validate stored values via allowlists before applying to DOM/CSS variables:

```
const STORAGE_PREFIX = "nv:";
const THEME_KEY = STORAGE_PREFIX + "theme";

const ALLOWED_THEMES = new Set([
  "default-dark",
  "default-light",
  "gruvbox-dark",
  "gruvbox-light"
]);

function safeGetEnum(key, allowed, fallback) {
  try {
    const v = localStorage.getItem(key);
    return allowed.has(v) ? v : fallback;
  } catch {
    return fallback;
  }
}

function safeSet(key, value) {
  try {
    localStorage.setItem(key, value);
  } catch {
    // Quota exceeded or blocked storage; ignore for UX-only preferences
  }
}

const theme = safeGetEnum(THEME_KEY, ALLOWED_THEMES, "default-dark");
document.documentElement.dataset.theme = theme;
```

1. If persistence is not needed, prefer `sessionStorage` (OWASP explicitly recommends it when persistence isn't required). [18]

Trade-offs
* Slightly more code; substantially better resilience against "weird stored values" and future mistakes. * Does not make storage confidential; confidentiality still depends on preventing XSS. [36]

Service worker stance: avoid unless you truly need it

Rationale
Service workers can intercept and modify network requests in their scope and can create long-lived client-side behavior. MDN highlights interception power and warns about input affecting registration. [20]

Threats mitigated
Persistent client compromise-like behaviors, caching privacy issues, accidental "stuck" content updates.

Code-only recommendation
* Do not register a service worker for a simple portfolio unless you have a clear offline/performance need. * If you previously experimented with one and want to ensure it's not controlling pages, add a debug-only "unregister" snippet during development:

```
if (location.hostname === "localhost" && "serviceWorker" in navigator) {
  navigator.serviceWorker.getRegistrations()
    .then((regs) => Promise.all(regs.map((r) => r.unregister())));
}
```

Trade-offs
* Losing offline capability; simplified risk profile.

Email harvesting defenses beyond simple splitters

No code-only technique fully prevents harvesting. Obfuscation is best understood as raising the cost for bots; research on bots and obfuscation emphasizes this "complicate extraction" framing. [9]

Below are options with increasing friction and trade-offs.

Reveal on explicit user gesture (lowest collateral damage)

Rationale
Do not place the email in initial HTML/DOM; only render it after a click. This blocks trivial “view source” harvesting and some simplistic crawlers.

Threats mitigated
Casual harvesting; simplistic scrapers.

Implementation
HTML:

```html
<button id="reveal-email" type="button">Show email</button>
<span id="email-slot" aria-live="polite"></span>
```

JS (example: base64-encoded, then reversed, then charcode-mapped—still not “secure,” just less obvious):

```javascript
function decodeEmail() {
  const u = atob("b3R1aG9u");     // "otuhon" example
  const d = atob("bW9jLmtjZHVk"); // "moc.kcduk" example
  const user = u.split("").reverse().join("");
  const domain = d.split("").reverse().join("");
  return `${user}@${domain}`;
}

document.getElementById("reveal-email").addEventListener("click", () => {
  const email = decodeEmail();
  const slot = document.getElementById("email-slot");

  // Render as text (safe sink).
  slot.textContent = email;

  // Optional: add a mailto link only after reveal.
  // const a = document.createElement("a");
  // a.href = "mailto:" + email;
  // a.rel = "noreferrer noopener";
  // a.textContent = email;
  // slot.replaceChildren(a);
});
```

Trade-offs
* Bots that execute JS can still harvest. * Keeps accessibility (screen readers will read the email after reveal).

Canvas rendering (stronger against static HTML parsing, worse for fingerprinting/accessibility)

Rationale
Canvas output is not present as plain text in the DOM, blocking naïve parsers.

Threats mitigated
Basic DOM scrapers.

Implementation sketch
* Draw the email into `<canvas>` on click; provide a separate “Copy email” button that writes to clipboard (Clipboard API requires secure contexts and has security considerations). [37]

Trade-offs (important)
Canvas APIs are frequently used in fingerprinting; avoid unless you accept that trade-off (fingerprinting literature highlights canvas-based fingerprints). [38]
Accessibility is worse unless you provide an alternative text representation.

WebFont glyph substitution (obscure, but high maintenance)

Rationale
Render the email using a custom font where glyph mapping does not correspond to visible characters, while the underlying text might be nonsense.

Trade-offs
* Accessibility failure risk and brittle across rendering contexts. * OCR can defeat it. * Usually not worth it unless you enjoy “security theater with style.”

Pragmatic recommendation
Prefer “reveal-on-click + contact form” and accept that published email can be harvested by sufficiently motivated scrapers.

Privacy-preserving contact forms (client-side code only)

A contact form is often the best way to avoid publishing an email. The key design is: the browser encrypts content to your public key and posts ciphertext to a receiver. This improves privacy **at rest** and reduces accidental disclosure, but does not protect against a compromised origin delivering modified JS. This limitation is demonstrated in browser-crypto discussions and examples where injected JS captures plaintext before encryption. [39]

Implement hybrid encryption via Web Crypto (example)

Rationale
Web Crypto provides primitives for encryption. MDN warns the API is easy to misuse and that system design is hard; an ACM paper analyzes WebCrypto misuse patterns. [13]

Threats mitigated
Email harvesting (no public email), passive disclosure if the receiver stores ciphertext, some third-party endpoint visibility (content unreadable without private key).

Prerequisites
* You publish a stable public key (e.g., RSA-OAEP public key in JWK form) in your repo as `public-key.jwk`. * You have (elsewhere) a receiver that accepts `POST` JSON (out of scope here; described only as an endpoint).

Client-side example (`contact.js`)

```
const PUBLIC_KEY_URL = "/public-key.jwk";
const ENDPOINT_URL = "https://receiver.invalid/submit"; // placeholder

function b64(bytes) {
  const bin = String.fromCharCode(...bytes);
  return btoa(bin);
}

async function importRsaOaepPublicKey(jwk) {
  return crypto.subtle.importKey(
    "jwk",
    jwk,
    { name: "RSA-OAEP", hash: "SHA-256" },
    false,
    ["encrypt"]
  );
}

async function encryptMessageToRecipient(message, recipientPublicKey) {
  const enc = new TextEncoder();
  const plaintext = enc.encode(message);

  // 1) Generate symmetric key (AES-GCM)
  const aesKey = await crypto.subtle.generateKey(
    { name: "AES-GCM", length: 256 },
    true,
    ["encrypt"]
  );

  // 2) Encrypt plaintext
  const iv = crypto.getRandomValues(new Uint8Array(12));
  const ciphertextBuf = await crypto.subtle.encrypt(
    { name: "AES-GCM", iv },
    aesKey,
    plaintext
  );

  // 3) Wrap/Encrypt AES key with RSA-OAEP
  const rawAes = new Uint8Array(await crypto.subtle.exportKey("raw", aesKey));
  const wrappedKeyBuf = await crypto.subtle.encrypt(
    { name: "RSA-OAEP" },
    recipientPublicKey,
    rawAes
  );

  return {
    v: 1,
    alg: "RSA-OAEP-256 + AES-256-GCM",
    iv: b64(iv),
    wrappedKey: b64(new Uint8Array(wrappedKeyBuf)),
    ciphertext: b64(new Uint8Array(ciphertextBuf))
  };
}

async function setupContactForm() {
  const form = document.getElementById("contact-form");
  const status = document.getElementById("contact-status");

  const jwk = await (await fetch(PUBLIC_KEY_URL, { cache: "no-store" })).json();
  const pub = await importRsaOaepPublicKey(jwk);

  form.addEventListener("submit", async (e) => {
    e.preventDefault();
    status.textContent = "Encrypting…";

    const msg = form.elements.message.value;
    const payload = await encryptMessageToRecipient(msg, pub);

    // Send ciphertext only
    await fetch(ENDPOINT_URL, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(payload),
      cache: "no-store",
      credentials: "omit"
    });

    form.reset();
    status.textContent = "Sent (encrypted).";
  });
}

setupContactForm().catch(() => {
  const status = document.getElementById("contact-status");
  if (status) status.textContent = "Contact form unavailable.";
});
```

Compatibility and trade-offs
Web Crypto is widely available but only in secure contexts; also easy to misuse. [40]
This does **not** defeat a targeted attacker who can modify delivered JS (the “malicious JS reads plaintext” problem). [39]
* Adds UX friction compared to `mailto:`; improves privacy posture.

Static per-page “password protection” via JS: risks and a safer framing

Rationale
Sometimes you want “discourage casual browsing” of a page. Client-side password-gating can provide **obfuscation**, but not real access control, because the client ultimately controls the code execution and can download ciphertext. PortSwigger emphasizes that client-side controls can be bypassed because the user controls the client. [41]

Threats mitigated
Casual visitors; basic indexing; superficial scraping.

Not mitigated
Targeted attackers; offline brute-force of weak passwords; anyone who can fetch the ciphertext and run cracking.

Implementation sketch (AES-GCM + PBKDF2)
* Store encrypted page content as base64 in JS. * Prompt for password. * Derive key via PBKDF2 and decrypt with AES-GCM.

Caveats
* You must treat this as “privacy by friction,” not security. * Web Crypto misuse risk applies; MDN warns explicitly. [13]

If you include this feature, disclose its limitation so you don’t create a false sense of confidentiality.

Asset metadata/EXIF stripping automation

Rationale
Metadata can leak sensitive details; CWE-212 describes sensitive data exposure via failure to remove such information. [22]

Threats mitigated
Accidental disclosure (location/device/time); doxxing risk.

Prerequisites
Install ExifTool for local workflow.

Implementation steps (manual)
To remove all metadata from an image:

```
exiftool -all= image.jpg
```

ExifTool shows `-all=` as the “delete all meta information” mechanism and provides examples. [42]

To avoid leaving `_original` backups, add:

```
exiftool -all= -overwrite_original image.jpg
```

ExifTool documents `-overwrite_original` behavior. [43]

Important caution
ExifTool warns you should not delete all metadata from most RAW images (except DNG) because proprietary RAW formats may rely on maker notes for conversion. [42]

Implementation steps (pre-commit hook)
Create `.git/hooks/pre-commit` (make executable) to strip metadata on staged images:

```
#!/usr/bin/env bash
set -euo pipefail

files=$(git diff --cached --name-only --diff-filter=ACM | \
  grep -Ei '\.(jpg|jpeg|png|webp|tif|tiff)$' || true)

[ -z "$files" ] && exit 0

echo "Stripping EXIF metadata from:"
echo "$files"

exiftool -all= -overwrite_original $files >/dev/null
git add $files
```

Trade-offs
* Adds a dependency (ExifTool) for contributors. * Prevents accidental leakage and is usually worth it for personal portfolios.

Link and navigation hardening: noopener/noreferrer and referrer policy coherence

Rationale
Links opened in a new tab can expose `window.opener` unless you use `rel="noopener"`. MDN explains `rel="noopener"` prevents the opened context from accessing the opener. [44]

OWASP also documents reverse tabnabbing and recommends `rel="noopener"` / `rel="noopener noreferrer"`. [45]

Threats mitigated
Reverse tabnabbing; referrer leakage.

Implementation steps
For external links that open in a new tab:

```
<a href="https://external.example"
   target="_blank"
   rel="noopener noreferrer">External</a>
```

You already set `<meta name="referrer" content="no-referrer">`, which MDN documents as controlling the `Referer` header for requests from the document. [10]
Keep it consistent (don't mix weaker per-link policies unless needed).

"No tracking" discipline and anti-fingerprinting code practices

Rationale
Fingerprinting can rely on stable characteristics (hardware, settings, rendering quirks) and is widely used for tracking; EFF's tool explains how trackers see a browser, and academic surveys summarize techniques and defenses. [2]

Threat model addressed
Browser fingerprinting and third-party correlation.

Code-level practices (high-value, low-cost)
*Don't include analytics scripts or marketing trackers (you currently appear clean). * Avoid adding canvas/WebGL/audio rendering "just for fun" if you don't need them; these APIs are often used in fingerprinting.* [38]
Avoid enumerating fonts, devices, or high-entropy APIs and sending them anywhere. * Keep third-party dependencies minimal; if you must use one, pin versions and use SRI. [46]

Trade-offs
* Less instrumentation/analytics; improved user trust and privacy.

Prioritized implementation roadmap, audit checklist, and automated checks

Prioritized code-only roadmap

Priority	Code-only task	What you change	Effort	Impact	Difficulty
P0	Repair CSP meta tag	Replace malformed CSP meta with valid `<meta http-equiv="Content-Security-Policy" …>`. [25]	0.5–2h	Very high	Low
P0	Eliminate inline scripts or hash them	Move bootstrap inline script to `bootstrap.js` or add hash-based `script-src` without `'unsafe-inline'`. [27]	2–6h	High	Medium
P0	Self-host fonts (or switch to system fonts)	Replace third-party font CSS with local WOFF2 + `@font-face`, tighten CSP. [47]	2–8h	High	Medium
P1	Self-host media now loaded from third-party origins	Mirror images/videos into repo; update CSP `img-src`/`media-src`.	4–16h	High	Medium
P1	Add DOM injection guardrails	Strip scripts/`on*` attrs from dynamically inserted `<main>`, keep safe sinks (`textContent`). [48]	2–6h	Medium–High	Medium
P1	Upgrade email/contact posture	Replace “email in JS” with click-to-reveal + (optional) encrypted contact form client code. [49]	4–24h	Medium–High	Medium–High
P2	Automate EXIF stripping	Add ExifTool pre-commit + CI check; strip images before commit. [50]	1–3h	Medium	Low
P2	Add security linting & CI	html-validate + ESLint + no-unsanitized + CSP evaluator checks. [51]	3–10h	Medium	Medium

Audit checklist (code-only)

CSP and script execution
Confirm CSP is actually applied in the browser (DevTools → Console/Security) and that inline script execution is either eliminated or controlled via hashes/Trusted Types. [52]

DOM XSS surfaces
Search for dangerous sinks (`innerHTML`, `insertAdjacentHTML`, `outerHTML`, `document.write`, `eval`, `new Function`) and require justification or sanitization. Trusted Types can help prevent future sink introduction. [53]

Third-party resources
Ensure there are no external `<script>`/`<link rel=stylesheet>` calls unless intentionally allowed via CSP and (where feasible) SRI. [28]

Storage
Ensure localStorage contains only preference values; validate allowlists; do not store secrets; treat stored values as attacker-controlled. [18]

Email/contact
Confirm email is not present in server-delivered HTML/JS as a clean string; confirm reveal requires user action; confirm any contact form does not leak plaintext to third parties and does not store messages in localStorage.

Assets
Ensure images in `images/` and similar folders contain no EXIF/GPS metadata. [54]

Automated tests and linting rules with example CI-as-code

Below is an example CI workflow that enforces code-only constraints. It assumes you adopt a Node-based dev toolchain for linting.

Example `package.json` scripts (minimal)

```
{
  "private": true,
  "type": "module",
  "scripts": {
    "lint:html": "html-validate \"**/*.html\"",
    "lint:js": "eslint \"**/*.js\"",
    "lint:security": "node tools/check-no-third-party.mjs && node tools/check-
csp.mjs && node tools/check-sri.mjs",
    "test": "npm run lint:html && npm run lint:js && npm run lint:security"
  },
  "devDependencies": {
    "eslint": "^9.0.0",
    "html-validate": "^8.0.0",
    "eslint-plugin-no-unsanitized": "^4.0.0",
    "csp_evaluator": "^1.1.0"
  }
}
```

- `html-validate` has built-in rule sets and a rule reference; it helps prevent malformed meta tags (including CSP markup errors). [55]
- ESLint is the standard JS linter; add a security-focused plugin for dangerous sinks. [56]
- Google's CSP Evaluator helps analyze subtle CSP bypasses (usable via the `csp_evaluator` npm package). [57]

Example security guard scripts

`tools/check-no-third-party.mjs` (fail build if certain external origins appear):

```
import { readFileSync, readdirSync } from "node:fs";
import { join } from "node:path";

const BLOCKED = [
  "api.fonts.coollabs.io",
  "fonts.coollabs.io",
  "raw.githubusercontent.com"
];

const htmlFiles = readdirSync(".", { withFileTypes: true })
  .filter((d) => d.isFile() && d.name.endsWith(".html"))
  .map((d) => d.name);

let bad = false;
for (const f of htmlFiles) {
  const s = readFileSync(f, "utf8");
  for (const host of BLOCKED) {
    if (s.includes(host)) {
      console.error(`Disallowed third-party origin "${host}" found in ${f}`);
      bad = true;
    }
  }
}

process.exit(bad ? 1 : 0);
```

`tools/check-sri.mjs` (require `integrity` on cross-origin script/link):

```
import { readFileSync, readdirSync } from "node:fs";

const htmlFiles = readdirSync(".").filter((f) => f.endsWith(".html"));
const re = /<(script|link)\b[^>]*(src|href)="https?:\/\/[^"]+"[^>]*>/gi;
```

```
let bad = false;
for (const f of htmlFiles) {
  const s = readFileSync(f, "utf8");
  const tags = s.match(re) || [];
  for (const tag of tags) {
    const hasIntegrity = /integrity="/i.test(tag);
    const hasCrossorigin = /crossorigin="/i.test(tag);
    if (!hasIntegrity || !hasCrossorigin) {
      console.error(`Missing SRI on tag in ${f}: ${tag}`);
      bad = true;
    }
  }
}
process.exit(bad ? 1 : 0);
```

This aligns with MDN guidance that SRI locks external resources and typically uses `integrity` + `crossorigin="anonymous"`, and that CORS headers are required from the CDN. [28]

`tools/check-csp.mjs` (basic sanity: ensure a CSP meta tag exists and is not malformed)

```
import { readFileSync, readdirSync } from "node:fs";

const htmlFiles = readdirSync(".").filter((f) => f.endsWith(".html"));
let bad = false;

for (const f of htmlFiles) {
  const s = readFileSync(f, "utf8");

  // Must contain a valid CSP meta declaration
  const ok = /<meta\s+http-equiv=["']Content-Security-Policy["']\s+content=["']
[^"']+["']\s*\/?>/i.test(s);
  const looksMalformed = /<meta\s+\\1default-src/i.test(s) || /\\3>/i.test(s);

  if (!ok || looksMalformed) {
    console.error(`CSP meta missing or malformed in ${f}`);
    bad = true;
  }
}

process.exit(bad ? 1 : 0);
```

This is grounded in MDN's definition of how CSP can be delivered via meta http-equiv and what CSP is intended to do. [25]

Example GitHub Actions workflow (CI-as-code)

`.github/workflows/code-security.yml`:

```
name: code-security

on:
  pull_request:
  push:
    branches: ["main"]

jobs:
  lint-and-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: "20"

      - run: npm ci
      - run: npm test
```

This is intentionally code-only: it validates that your repository contains a consistent CSP meta tag, doesn't reintroduce third-party hosts, and avoids unsafe DOM sinks via linting.

Mermaid diagrams for privacy-sensitive flows

Client-side encrypted contact flow (ciphertext-only submission)

```
sequenceDiagram
  participant U as User
  participant B as Browser (your JS)
  participant R as Receiver endpoint (out of scope)
  participant K as Your private key (offline)

  U->>B: Enter message + submit form
  B->>B: Encrypt (AES-GCM) + wrap key (RSA-OAEP)
  B->>R: POST ciphertext JSON
  R-->>B: 200 OK (no plaintext stored by client)
  Note over K: You later decrypt ciphertext offline
```

Email reveal flow (gesture-gated)

```
sequenceDiagram
  participant V as Visitor
  participant P as Page HTML
  participant J as JS (email decode)
  participant D as DOM

  V->>P: Load page
  P-->>V: No email rendered as text initially
  V->>J: Click "Show email"
  J->>D: Insert email via textContent
  D-->>V: Email becomes visible/copyable
```

These flows reduce casual scraping, but you should avoid overstating their strength against determined scrapers (JS execution/OCR). [58]

---

[1] [10] https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/meta/name/referrer
https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/meta/name/referrer

[2] https://coveryourtracks.eff.org/
https://coveryourtracks.eff.org/

[3] [12] [14] [52] https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP
https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP

[4] [5] [18] [23] [36] https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html
https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html

[6] [13] [40] https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API
https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API

[7] [25] https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/meta/http-equiv
https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/meta/http-equiv

[8] [19] https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/XSS
https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/XSS

[9] https://ediss.sub.uni-hamburg.de/bitstream/ediss/11820/1/see_diss_published.pdf
https://ediss.sub.uni-hamburg.de/bitstream/ediss/11820/1/see_diss_published.pdf

[11] [15] [24] [26] https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Security-Policy
https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Security-Policy

[16] [48] https://developer.mozilla.org/en-US/docs/Web/API/DOMParser/parseFromString
https://developer.mozilla.org/en-US/docs/Web/API/DOMParser/parseFromString

[17] https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html
https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html

[20] https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerContainer/register
https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerContainer/register

[21] https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Attributes/autocomplete
https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Attributes/autocomplete

[22] [54] https://cwe.mitre.org/data/definitions/212.html
https://cwe.mitre.org/data/definitions/212.html

[27] https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html
https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html

[28] [46] https://developer.mozilla.org/en-US/docs/Web/Security/Practical_implementation_guides/SRI
https://developer.mozilla.org/en-US/docs/Web/Security/Practical_implementation_guides/SRI

[29] [53] https://www.w3.org/TR/trusted-types/
https://www.w3.org/TR/trusted-types/

[30] https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Security-Policy/require-trusted-types-for
https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Security-Policy/require-trusted-types-for

[31] https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API
https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API

[32] https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching
https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching

[33] https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/Subresource_Integrity
https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/Subresource_Integrity

[34] https://github.com/mozilla/eslint-plugin-no-unsanitized
https://github.com/mozilla/eslint-plugin-no-unsanitized

[35] https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/06-Testing_for_Client-side_Resource_Manipulation
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/06-Testing_for_Client-side_Resource_Manipulation

[37] https://developer.mozilla.org/en-US/docs/Web/API/Clipboard_API
https://developer.mozilla.org/en-US/docs/Web/API/Clipboard_API

[38] [47] https://dl.acm.org/doi/10.1145/3386040
https://dl.acm.org/doi/10.1145/3386040

[39] https://www.cs.tufts.edu/comp/116/archive/fall2015/mseltzer.pdf
https://www.cs.tufts.edu/comp/116/archive/fall2015/mseltzer.pdf

[41] https://portswigger.net/burp/documentation/desktop/testing-workflow/vulnerabilities/input-validation/client-side-controls
https://portswigger.net/burp/documentation/desktop/testing-workflow/vulnerabilities/input-validation/client-side-controls

[42] [43] [50] https://exiftool.org/exiftool_pod.pdf
https://exiftool.org/exiftool_pod.pdf

[44] https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Attributes/rel/noopener
https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Attributes/rel/noopener

[45] https://owasp.org/www-community/attacks/Reverse_Tabnabbing
https://owasp.org/www-community/attacks/Reverse_Tabnabbing

[49] [58] https://spencermortensen.com/articles/email-obfuscation/
https://spencermortensen.com/articles/email-obfuscation/

[51] [55] https://html-validate.org/rules/
https://html-validate.org/rules/

[56] https://eslint.org/docs/latest/
https://eslint.org/docs/latest/

[57] https://github.com/google/csp-evaluator
https://github.com/google/csp-evaluator