

Content Security Policy Estricta

Nonces, strict-dynamic, Report-Only y Trusted Types contra XSS

PuigFlows - Edición en español - 17/10/2026

Por qué una Content Security Policy sigue importando

El cross-site scripting aparece una y otra vez en los informes de bug bounty por una razón aburrida: basta un escape olvidado en una plantilla. Una Content Security Policy (CSP) es la red de seguridad bajo ese error. Es una cabecera de respuesta HTTP que le dice al navegador qué scripts puede ejecutar, de modo que, aunque un atacante consiga inyectar marcado, el navegador se niega a ejecutarlo.

CSP es defensa en profundidad, no un sustituto de la codificación de salida. Sigues escapando, sanitizando y validando. CSP limita el daño cuando fallas, y además te da algo raro en seguridad: un flujo de informes que te dice dónde has fallado.

Por qué fallan las políticas con lista de permitidos

La política que casi todos escriben primero se ve así: `script-src 'self' https://cdn.example.com https://www.google-analytics.com`. Parece precisa, pero los estudios de políticas reales encuentran siempre el mismo fallo: cualquier host permitido que sirva un endpoint JSONP, una versión antigua de Angular o un archivo controlado por usuarios se convierte en un bypass. Una lista de hosts confía en todo lo que ese host llegue a servir, y eso tú no lo controlas.

La respuesta moderna es una CSP estricta basada en dos ideas. Primero, cada script legítimo lleva un nonce aleatorio distinto en cada respuesta. Segundo, 'strict-dynamic' permite que un script ya confiable cargue otros scripts, así que dejas de enumerar hosts por completo.

Una política estricta para empezar

```
Content-Security-Policy:
  script-src 'nonce-{ALEATORIO}' 'strict-dynamic' https: 'unsafe-inline';
  object-src 'none';
  base-uri 'none';
  frame-ancestors 'self';
  report-to csp-endpoint
```

Léela directiva por directiva:

- `script-src 'nonce-...' 'strict-dynamic'` ejecuta solo los scripts con el nonce, más lo que esos scripts carguen dinámicamente.
- `https: 'unsafe-inline'` asusta, pero es un respaldo. Los navegadores que entienden nonces ignoran ambos; solo los muy antiguos los usan, y de otro modo se romperían por completo.
- `object-src 'none'` elimina el contenido de plugins, una vía clásica de bypass.
- `base-uri 'none'` impide que una etiqueta `<base>` inyectada redirija al atacante todas las URL relativas de scripts.
- `frame-ancestors` es el reemplazo moderno de X-Frame-Options. Ojo: solo funciona como cabecera, nunca en una etiqueta `<meta>`.

Generar el nonce correctamente

Las reglas son pocas e inflexibles: al menos 128 bits de un generador criptográficamente seguro, un valor nuevo en cada respuesta, y el mismo valor en la cabecera y en cada atributo nonce. Un nonce que se repite es una contraseña que acabas de publicar.

```
// server.mjs (Node 20+, Express)
import crypto from "node:crypto";
import express from "express";

const app = express();
```

```

const ENFORCE = process.env.CSP_ENFORCE === "true";

app.use((req, res, next) => {
  const nonce = crypto.randomBytes(16).toString("base64");
  res.locals.cspNonce = nonce;

  const policy = [
    "script-src 'nonce-" + nonce + "' 'strict-dynamic' https: 'unsafe-inline'",
    "object-src 'none'",
    "base-uri 'none'",
    "frame-ancestors 'self'",
    "report-to csp-endpoint",
  ].join("; ");

  // Primero Report-Only: el navegador informa de las violaciones pero no bloquea nada.
  res.setHeader(
    ENFORCE ? "Content-Security-Policy" : "Content-Security-Policy-Report-Only",
    policy
  );
  res.setHeader("Reporting-Endpoints", 'csp-endpoint="/csp-reports"');
  // Una página cacheada repetiría un nonce viejo: mantén el HTML fuera de cachés compartidas.
  res.setHeader("Cache-Control", "private, no-cache");
  next();
});

```

En una plantilla escribes entonces `<script nonce="<%= cspNonce %>" src="/app.js"></script>`. La misma idea en Python con FastAPI:

```

# main.py
import base64, logging, os
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse, Response

app = FastAPI()
log = logging.getLogger("csp")

@app.middleware("http")
async def csp_headers(request: Request, call_next):
    nonce = base64.b64encode(os.urandom(16)).decode()
    request.state.csp_nonce = nonce
    response = await call_next(request)
    response.headers["Content-Security-Policy-Report-Only"] = "; ".join([
        f"script-src 'nonce-{nonce}' 'strict-dynamic' https: 'unsafe-inline'",
        "object-src 'none'",
        "base-uri 'none'",
        "frame-ancestors 'self'",
        "report-to csp-endpoint",
    ])
    response.headers["Reporting-Endpoints"] = 'csp-endpoint="/csp-reports"'
    response.headers["Cache-Control"] = "private, no-cache"
    return response

@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    n = request.state.csp_nonce
    return f'<!doctype html><title>Demo</title><script nonce="{n}" src="/static/app.js"></script>'

```

Despliega primero en modo Report-Only

Activar el bloqueo el primer día es la forma de romper páginas de pago y acabar desactivando CSP para siempre. Usa antes Content-Security-Policy-Report-Only. El navegador evalúa la política, envía un informe por cada violación y no bloquea nada. Déjalo una o dos semanas, corrige lo que aparezca y después activa el bloqueo.

El flujo de informes moderno combina la cabecera Reporting-Endpoints con la directiva report-to, como arriba. Los navegadores antiguos aún usan la directiva heredada report-uri y un formato distinto, así que un receptor robusto acepta ambos:

```

@app.post("/csp-reports", status_code=204)
async def csp_reports(request: Request):
    raw = await request.body()
    if len(raw) > 64_000: # los informes son entrada no confiable
        return Response(status_code=413)
    try:
        payload = await request.json()
    except ValueError:
        return Response(status_code=400)

    # Reporting API: una lista de {"type": "csp-violation", "body": {...}}
    # report-uri heredado: {"csp-report": {...}}
    items = payload if isinstance(payload, list) else [payload]
    for item in items:
        body = item.get("body") or item.get("csp-report") or {}
        log.warning(
            "csp_violation",
            extra={
                "directive": body.get("effectiveDirective") or body.get("effective-directive"),
                "blocked": body.get("blockedURL") or body.get("blocked-uri"),
                "page": body.get("documentURL") or body.get("document-uri"),
            },
        )

```

Espera ruido. Las extensiones del navegador inyectan scripts y generan violaciones que no son un fallo tuyo. Agrupa los informes por directiva y URL bloqueada, e ignora orígenes como chrome-extension:// o moz-extension:// en lugar de perseguirlos.

Refactorizar lo que se rompe

Cuando empiecen a llegar informes, tres patrones concentran casi todo el trabajo.

- Manejadores de eventos en línea. Atributos como onclick se bloquean y un nonce no puede legitimarlos. Mueve la lógica a un script y añade los listeners desde allí.
- Scripts en línea sin nonce. Añade el atributo o mueve el código a un archivo externo.
- eval y similares. eval(), new Function() y setTimeout con cadenas requieren 'unsafe-eval'. Elimínalos en lugar de permitirlos; los empaquetadores suelen poder configurarse para evitar source maps basados en eval en producción.

```

<!-- Antes: bloqueado por CSP -->
<button onclick="save()">Guardar</button>

<!-- Después -->
<button id="save">Guardar</button>
<script nonce="{{ nonce }}">
    document.getElementById("save").addEventListener("click", save);
</script>

```

Los estilos merecen una nota: la prioridad es un script-src estricto, y muchos equipos mantienen un style-src 'self' 'unsafe-inline' más laxo durante un tiempo, porque los estilos en línea son mucho menos peligrosos que los scripts en línea. Endurécelo después con nonces o hashes si puedes.

Trusted Types: cerrar el XSS basado en DOM en el sumidero

Una CSP con nonces detiene las etiquetas script inyectadas, pero el XSS basado en DOM ocurre cuando tu propio código escribe datos del atacante en innerHTML. Trusted Types ataca esa clase directamente: con las directivas adecuadas, los sumideros peligrosos solo aceptan objetos producidos por una política que tú defines, y las cadenas simples lanzan un error.

```
Content-Security-Policy: require-trusted-types-for 'script'; trusted-types app-html
```

```
import DOMPurify from "dompurify";

// Detecta la función: el soporte varía entre navegadores, así que trátalo como endurecimiento.
const policy = window.trustedTypes?.createPolicy("app-html", {
  createHTML: (dirty) => DOMPurify.sanitize(dirty),
});

function renderComment(el, userComment) {
  if (policy) {
    el.innerHTML = policy.createHTML(userComment); // permitido: sanitizado por la política
  } else {
    el.textContent = userComment; // alternativa segura: nada de HTML
  }
}
```

La gran ventaja es la auditabilidad. En lugar de revisar cada línea en busca de asignaciones peligrosas, revisas un puñado de políticas y el navegador hace cumplir el resto.

Prueba la política en CI

Una CSP que retrocede en silencio es peor que ninguna, porque el equipo cree estar protegido. Una pequeña prueba end-to-end detecta las regresiones habituales: un script en línea nuevo, una etiqueta de terceros añadida por marketing, una cabecera eliminada por un proxy.

```
// csp.spec.ts (Playwright)
import { test, expect } from "@playwright/test";

test("la home tiene una CSP estricta y ninguna violación", async ({ page }) => {
  await page.addInitScript(() => {
    document.addEventListener("securitypolicyviolation", (e) => {
      const w = window as any;
      w.__csp = (w.__csp ?? []).concat(e.effectiveDirective + " <- " + e.blockedURI);
    });
  });

  const res = await page.goto("/");
  const header =
    res?.headers()["content-security-policy"] ??
    res?.headers()["content-security-policy-report-only"];

  expect(header).toContain("'strict-dynamic'");
  expect(header).toContain("object-src 'none'");

  await page.waitForLoadState("networkidle");
  const violations = await page.evaluate(() => (window as any).__csp ?? []);
  expect(violations).toEqual([]);
});
```

Errores habituales

- Nonces estáticos. Un nonce fijado en el build, o cacheado por un CDN junto con el HTML, no protege nada. Si sirves HTML estático, usa hashes ('sha256-...') en lugar de nonces.
- Política en una etiqueta <meta>. Funciona para algunas directivas, pero no admite frame-ancestors, report-to ni sandbox, y solo se aplica una vez analizada la etiqueta.
- Permitir 'unsafe-inline' sin nonce. En un navegador moderno, un nonce o un hash hace que 'unsafe-inline' se ignore; sin ellos, desactiva la protección por completo.
- Comodines y data: en script-src. Cada uno es un bypass listo para usar.
- Varias cabeceras. Si llegan varias cabeceras CSP, el navegador aplica todas, así que el resultado es la intersección. Un proxy que añade la suya puede romper tus páginas.

- Olvidar las demás directivas. `default-src`, `connect-src`, `img-src` y `form-action` limitan la exfiltración y los envíos de formularios maliciosos. Empieza estricto con los scripts y endurece el resto después.

Checklist de producción

- El nonce tiene 128 bits o más, es aleatorio por respuesta y aparece en la cabecera y en cada etiqueta script.
- Las respuestas HTML nunca se sirven desde una caché compartida con un nonce obsoleto.
- `object-src 'none'`, `base-uri 'none'` y `frame-ancestors` están definidos.
- Report-Only se ejecutó el tiempo suficiente para cubrir tráfico real; los informes se recogen, con límite de tamaño y agrupados.
- No quedan manejadores en línea ni eval; las etiquetas de terceros se cargan mediante un cargador con nonce.
- Una prueba en CI rompe el build ante violaciones o si falta la cabecera.
- Se ha evaluado Trusted Types para todo el código que escribe en innerHTML.

Cómo evalúa el navegador una CSP: el modelo mental que evita el 90% de los errores

La mayoría de los problemas con CSP no vienen de la sintaxis, sino de un modelo mental incorrecto sobre cuándo y cómo decide el navegador. Conviene entenderlo una vez, con precisión, porque explica por qué una política "estricta" funciona y por qué una que parece equivalente no.

Cada vez que el navegador está a punto de hacer algo que CSP regula (ejecutar un script, cargar una imagen, abrir una conexión, enviar un formulario), consulta la política asociada al documento. Cada tipo de acción pertenece a una directiva concreta: `script-src` para scripts, `img-src` para imágenes, `connect-src` para fetch, XHR y WebSocket, y así sucesivamente. Si la directiva específica no existe, el navegador recurre a `default-src`; si tampoco existe, la acción se permite. Esa última frase importa: una política que solo define `script-src` no restringe nada más.

Para los scripts, la decisión sigue un orden que merece memorizarse:

1. Si la fuente del script lleva un nonce que coincide con el de la política, o su contenido coincide con un hash declarado, se permite.
2. Si el script fue insertado por otro script que ya era de confianza y la política incluye `'strict-dynamic'`, se permite aunque no lleve nonce.
3. Si la política contiene una lista de hosts y el origen del script encaja, se permite, salvo que `'strict-dynamic'` esté presente: en ese caso la lista de hosts se ignora por completo.
4. En cualquier otro caso se bloquea y, si hay un destino de informes, se envía una violación.

El paso 3 es la razón de que la política de partida del artículo incluya `https:` y `'unsafe-inline'` sin que eso la debilite. Un navegador con soporte de CSP3 ve el nonce, activa la lógica de nonce más `'strict-dynamic'` e ignora tanto `https:` como `'unsafe-inline'`. Un navegador de hace diez años no entiende nonces y cae a esos dos valores, que mantienen el sitio funcionando aunque con protección mucho menor. Es degradación elegante, no una puerta trasera.

Scripts insertados por el parser frente a scripts insertados dinámicamente

La distinción más sutil de `'strict-dynamic'` es quién insertó el script. Un script que aparece en el HTML original, analizado por el parser, necesita nonce o hash por sí mismo. Un script creado desde JavaScript con `document.createElement("script")` hereda la confianza del script que lo creó, siempre que el creador fuera de confianza. Esto es lo que permite que un cargador con nonce cargue cincuenta dependencias sin

enumerarlas.

Pero hay una excepción deliberada: los scripts insertados mediante APIs que analizan HTML, como `document.write` o `innerHTML` con una etiqueta `script`, se consideran "insertados por el parser" y pierden la confianza heredada. Si una librería de terceros depende de `document.write` para cargar más scripts, 'strict-dynamic' la romperá y verás una violación clara en los informes. La solución correcta es sustituirla o aislarla en un `iframe` con `sandbox`, no relajar la política.

Qué ignora cada versión de navegador

Para diseñar el respaldo de forma consciente, ayuda tener la tabla delante:

- CSP1 (navegadores muy antiguos): solo entiende listas de hosts y 'unsafe-inline'. Verá `https:` y 'unsafe-inline' y funcionará como una política laxa.
- CSP2: añade nonces y hashes. Si hay nonce, 'unsafe-inline' se ignora, pero 'strict-dynamic' le resulta desconocido, así que seguirá usando la lista de hosts (`https:`).
- CSP3 (todos los navegadores actuales): entiende 'strict-dynamic', ignora la lista de hosts y 'unsafe-inline' cuando hay nonce, y es la lógica que realmente quieres que se ejecute.

La consecuencia práctica es que la política estricta se evalúa con el navegador actual de tus usuarios, no con el más antiguo. Revisa tus analíticas de navegador antes de preocuparte por el respaldo: en la mayoría de los sitios, el porcentaje de navegadores sin CSP3 es despreciable.

Hashes, SRI y HTML estático: cuando un nonce no es posible

Un nonce exige que el servidor genere HTML nuevo en cada petición. Si sirves un sitio estático desde un CDN o un bucket, no hay servidor que genere nada, y cachear el HTML con un nonce fijo anula la protección. Para ese caso existen los hashes: en lugar de marcar cada script con un valor aleatorio, declaras en la cabecera el SHA-256, SHA-384 o SHA-512 del contenido exacto del script en línea permitido.

```
Content-Security-Policy:
  script-src 'sha256-3NaxZ2gH5tLw8fQyqk4nV0ZbH8m1ZrW0eQ0jq7dY9xE=' 'strict-dynamic' https;;
  object-src 'none';
  base-uri 'none'
```

El hash se calcula sobre los bytes exactos entre las etiquetas `<script>` y `</script>`, incluidos espacios y saltos de línea. Cambiar un solo espacio invalida el hash, lo cual es una virtud (nadie puede alterar el script sin que la política lo note) y una molestia (cualquier reformato del build rompe la política). Por eso conviene calcular los hashes automáticamente en el mismo paso del build que genera el HTML:

```
// scripts/csp-hashes.mjs
// Recorre el HTML generado, calcula el hash de cada script en línea
// y escribe la cabecera lista para pegar en el CDN.
import { createHash } from "node:crypto";
import { readdirSync, readFileSync, writeFileSync } from "node:fs";
import { join } from "node:path";

const DIST = "dist";
const hashes = new Set();

function walk(dir) {
  for (const entry of readdirSync(dir, { withFileTypes: true })) {
    const full = join(dir, entry.name);
    if (entry.isDirectory()) walk(full);
    else if (entry.name.endsWith(".html")) scan(full);
  }
}

function scan(file) {
  const html = readFileSync(file, "utf8");
  // Solo scripts en línea sin atributo src
  const re = /<script(?:![^>]*\bsrc=)[^>]*>([\s\S]*?)<\/script>/gi;
```

```

let m;
while ((m = re.exec(html))) {
  const body = m[1];
  if (!body.trim()) continue;
  const digest = createHash("sha256").update(body, "utf8").digest("base64");
  hashes.add("'sha256-" + digest + "'");
}

walk(DIST);
const header = "script-src " + [...hashes].join(" ") + " 'strict-dynamic' https; object-src 'none';
base-uri 'none'";
writeFileSync("csp-header.txt", header + "\n");
console.log("Hashes únicos: " + hashes.size);

```

Dos advertencias sobre este enfoque. Primero, una política con cientos de hashes crece mucho y se paga en cada respuesta; si superas unas decenas, plantéate un renderizado con servidor ligero que ponga nonces, o un Worker en el borde. Segundo, el hash cubre scripts en línea, no manejadores como onclick. Existe 'unsafe-hashes' para permitir manejadores concretos por su hash, pero amplía la superficie y debe tratarse como una muleta temporal mientras refactorizas.

Subresource Integrity: el complemento natural para scripts externos

Los hashes de CSP protegen contenido en línea; Subresource Integrity (SRI) protege contenido externo. El atributo integrity hace que el navegador compare el hash del archivo descargado con el declarado y se niegue a ejecutarlo si no coincide, lo que neutraliza el caso de un CDN comprometido.

```

<script
  nonce="{ { nonce } }"
  src="https://cdn.example.com/lib@4.2.1/lib.min.js"
  integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"
  crossorigin="anonymous"></script>

```

Para que SRI funcione, la versión debe estar fijada: con una URL sin versión que el proveedor actualiza, el hash dejará de coincidir el día del próximo despliegue ajeno y tu página se quedará sin esa librería. La regla es sencilla: SRI para lo que tú versionas, y para lo que no puedes versionar, alojar la copia en tu propio origen.

Una prueba de CI que recorra el HTML y falle si encuentra un script de otro origen sin integrity convierte esa regla en una garantía, en lugar de una buena intención.

El mapa completo de directivas: más allá de script-src

Un script estricto es la prioridad, pero una CSP completa limita también por dónde puede salir la información y qué recursos pueden cargarse. Estas son las directivas que de verdad aportan en la práctica, agrupadas por la pregunta que responden.

¿Qué puede cargar la página? default-src es la red de seguridad para todo lo que no tenga directiva propia. img-src, font-src, media-src, style-src y worker-src controlan cada tipo de recurso. manifest-src acota el manifiesto de la aplicación web. frame-src decide qué páginas puedes incrustar, y child-src existe sobre todo por compatibilidad.

¿A dónde puede hablar la página? connect-src gobierna fetch, XHR, WebSocket, EventSource y navigator.sendBeacon. Es la directiva que más limita la exfiltración: un atacante que logra ejecutar código pero no puede abrir conexiones a su servidor lo tiene mucho más difícil para sacar datos. form-action hace lo propio con los envíos de formularios, que no están cubiertos por default-src.

¿Quién puede incrustarte? frame-ancestors sustituye a X-Frame-Options y defiende contra el clickjacking. Los valores habituales son 'none' (nadie), 'self' (solo tu origen) o una lista concreta de orígenes de socios.

¿Qué entorno de ejecución tiene la página? sandbox aplica restricciones estilo iframe a todo el documento,

upgrade-insecure-requests reescribe URLs http: a https: y require-trusted-types-for activa Trusted Types, que vemos más adelante.

Una política base razonable para una aplicación web con API en el mismo dominio, un CDN de imágenes y un proveedor de analítica se vería así:

```
Content-Security-Policy:
  default-src 'self';
  script-src 'nonce-{RANDOM}' 'strict-dynamic' https: 'unsafe-inline';
  style-src 'self' 'unsafe-inline';
  img-src 'self' data: https://images.example-cdn.com;
  font-src 'self';
  connect-src 'self' https://api.example.com https://analytics.example.net;
  frame-src 'none';
  form-action 'self';
  frame-ancestors 'none';
  object-src 'none';
  base-uri 'none';
  upgrade-insecure-requests;
  report-to csp-endpoint
```

Dos detalles de esta política. Primero, img-src incluye data: porque muchas aplicaciones incrustan iconos pequeños; las imágenes con data: no ejecutan código, así que el riesgo es mucho menor que con data: en script-src. Segundo, la directiva navigate-to aparece en borradores antiguos pero ningún navegador de producción la implementa; si la ves en una guía, ignórala.

Nonces en la práctica: frameworks y plataformas

El patrón es el mismo en todas partes (un valor aleatorio por petición, puesto en la cabecera y en cada etiqueta), pero cada framework tiene su manera de hacerlo sin pelearte con él. Estos son los casos que más aparecen.

Next.js (App Router)

Next.js genera el nonce en un middleware y lo propaga a los componentes del servidor mediante una cabecera de petición. Cuando el framework ve el nonce en la cabecera Content-Security-Policy de la petición, lo aplica automáticamente a sus propios scripts. Un detalle fundamental: el renderizado debe ser dinámico, porque una página generada estáticamente en el build no puede llevar un nonce distinto por petición.

```
// middleware.ts
import { NextRequest, NextResponse } from "next/server";

export function middleware(request: NextRequest) {
  const nonce = Buffer.from(crypto.randomUUID()).toString("base64");
  const isDev = process.env.NODE_ENV === "development";

  const csp = [
    "default-src 'self'",
    "script-src 'self' 'nonce-" + nonce + "' 'strict-dynamic'" + (isDev ? " 'unsafe-eval'" : ""),
    "style-src 'self' 'unsafe-inline'",
    "img-src 'self' blob: data:",
    "object-src 'none'",
    "base-uri 'none'",
    "form-action 'self'",
    "frame-ancestors 'none'",
    "upgrade-insecure-requests",
  ].join("; ");

  const headers = new Headers(request.headers);
  headers.set("x-nonce", nonce);
  headers.set("Content-Security-Policy", csp);

  const response = NextResponse.next({ request: { headers } });
  response.headers.set("Content-Security-Policy", csp);
}
```

```

    return response;
  }

  export const config = {
    matcher: [{
      source: "/((?!api|_next/static|_next/image|favicon.ico).*)",
      missing: [{ type: "header", key: "next-router-prefetch" }],
    }],
  };
};

```

Observa que 'unsafe-eval' solo se añade en desarrollo, donde las herramientas de recarga en caliente lo necesitan. En producción debe desaparecer. En un componente del servidor lees el nonce con `headers().get("x-nonce")` y se lo pasas a cualquier etiqueta `<Script>` que añadas tú.

Express con Helmet

Helmet permite que el valor de una directiva sea una función que recibe la petición y la respuesta, lo que encaja con el nonce por respuesta sin escribir la cabecera a mano:

```

import crypto from "node:crypto";
import express from "express";
import helmet from "helmet";

const app = express();

app.use((req, res, next) => {
  res.locals.cspNonce = crypto.randomBytes(16).toString("base64");
  next();
});

app.use(helmet({
  contentSecurityPolicy: {
    useDefaults: false,
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: [
        (req, res) => "'nonce-' + res.locals.cspNonce + '",",
        "'strict-dynamic'",
      ],
      objectSrc: ["'none'"],
      baseUri: ["'none'"],
      frameAncestors: ["'none'"],
      formAction: ["'self'"],
    },
    reportOnly: process.env.CSP_ENFORCE !== "true",
  },
}));

```

Django

Desde Django 6.0 existe soporte integrado para CSP mediante middleware y una configuración declarativa; en versiones anteriores, el paquete `django-csp` cubre lo mismo. El punto clave es que la plantilla recibe el nonce a través del contexto y lo imprime en cada etiqueta.

```

# settings.py (django-csp 4.x)
MIDDLEWARE = [
    "csp.middleware.CSPMiddleware",
    # ...
]

CONTENT_SECURITY_POLICY = {
  "DIRECTIVES": {
    "default-src": ["'self'"],
    "script-src": ["'self'", "'strict-dynamic'"],
    "object-src": ["'none'"],
    "base-uri": ["'none'"],
    "frame-ancestors": ["'none'"],
    "form-action": ["'self'"],
  },
}

```

```

    "include-nonce-in": ["script-src"],
  },
}

# plantilla.html
# <script nonce="{ request.csp_nonce }" src="% static 'app.js' %"></script>

```

Recuerda que la configuración exacta cambia entre versiones del paquete; confirma los nombres de las claves con la documentación de la versión que tengas instalada.

El borde: Cloudflare Workers y HTMLRewriter

Si no puedes tocar la aplicación (un CMS heredado, un producto de terceros), puedes añadir nonces desde un Worker que reescribe el HTML en el borde. El Worker genera el nonce, lo pone en cada etiqueta script y en la cabecera. Es una solución potente, pero asume la responsabilidad de que ningún script de la página pase sin nonce, así que empieza siempre en Report-Only.

```

export default {
  async fetch(request, env) {
    const upstream = await fetch(request);
    const type = upstream.headers.get("content-type") || "";
    if (!type.includes("text/html")) return upstream;

    const bytes = crypto.getRandomValues(new Uint8Array(16));
    const nonce = btoa(String.fromCharCode(...bytes));

    const rewritten = new HTMLRewriter()
      .on("script", { element(el) { el.setAttribute("nonce", nonce); } })
      .transform(upstream);

    const res = new Response(rewritten.body, rewritten);
    res.headers.set(
      "Content-Security-Policy-Report-Only",
      "script-src 'nonce-' + nonce + "' 'strict-dynamic' https: 'unsafe-inline'; object-src 'none';
base-uri 'none'"
    );
    res.headers.set("Cache-Control", "private, no-cache");
    return res;
  },
};

```

Ojo con una trampa de esta técnica: si la aplicación original ya pone un nonce o una CSP, las dos políticas se intersectan, y si el HTML original contiene un <script> inyectado por un atacante, el Worker también le pondrá nonce. Un reescritor de nonces protege contra scripts que se añaden después de la respuesta, no contra el HTML malicioso que ya viene dentro. Por eso es un puente de migración, no un destino final.

Terceros: analítica, gestores de etiquetas y widgets sin perder el control

Las etiquetas de terceros son la causa número uno de violaciones y de equipos que abandonan CSP. La estrategia que funciona se apoya en una idea sencilla: tu política debe confiar en un único punto de entrada controlado por ti, y ese punto decide qué más se carga.

Un cargador con nonce

En lugar de pegar diez fragmentos de proveedores en el HTML, mantienes un solo script propio, con nonce, que inyecta los demás. Con 'strict-dynamic' todo lo que ese cargador añada hereda la confianza.

```

<script nonce="{ nonce }">
(function () {
  var tags = [
    "https://www.googletagmanager.com/gtm.js?id=GTM-XXXXXXX",
    "https://widget.example-chat.com/loader.js"
  ]

```

```

];
tags.forEach(function (src) {
  var s = document.createElement("script");
  s.src = src;
  s.async = true;
  document.head.appendChild(s);
});
})();
</script>

```

Esto traslada el problema de gobernanza al lugar correcto: quien edita la lista de etiquetas pasa por revisión de código, no por un panel de marketing sin control de versiones. Si usas un gestor de etiquetas, ten presente que cualquiera con permisos de publicación en él puede ejecutar JavaScript arbitrario en tu sitio, con o sin CSP. Restringe esos permisos con el mismo rigor que el acceso a producción.

Iframes con sandbox para lo que no controlas

Los widgets que no puedes auditar (chats, mapas, vídeos, formularios de pago embebidos) no deberían ejecutarse en tu origen. Un iframe con el atributo sandbox los aísla: sin los permisos que concedas explícitamente, no pueden acceder a tus cookies, navegar la ventana superior ni abrir ventanas emergentes.

```

<iframe
  src="https://widget.example-chat.com/embed"
  sandbox="allow-scripts allow-forms allow-same-origin"
  referrerpolicy="no-referrer"
  loading="lazy"
  title="Chat de soporte"></iframe>

```

Atención: combinar allow-scripts con allow-same-origin en un iframe del mismo origen que la página permite al contenido quitarse el sandbox. Con un origen distinto es seguro; con el mismo origen, no lo combines.

Qué hacer con lo que no cabe

Algunas etiquetas simplemente no funcionan con una política estricta: usan document.write, eval, manejadores en línea o inyectan estilos de forma incompatible. Tienes tres salidas honestas. Pedir al proveedor una versión compatible (muchos ya la tienen), sustituirlo por una alternativa que sí lo sea, o aislarlo en un iframe de otro origen. La salida deshonestas, que conviene evitar, es abrir un agujero en la política que anula la protección de toda la página para satisfacer a un solo proveedor.

Informes a fondo: de ruido a señal

El valor de la fase Report-Only depende por completo de lo bien que recojas e interpretes los informes. Un receptor que solo escribe líneas de log produce miles de entradas inservibles en un sitio con tráfico real; uno que agrega y deduplica produce una lista corta y accionable.

Los dos formatos y las dos cabeceras

Hay dos mecanismos que conviven. El heredado usa la directiva report-uri y envía un cuerpo con tipo application/csp-report que contiene un objeto csp-report. El moderno usa la Reporting API: declaras un punto de destino con la cabecera Reporting-Endpoints y lo referencias desde la directiva report-to; el navegador envía una lista de informes con tipo application/reports+json. Para cubrir ambos, se envían las dos directivas a la vez, y los navegadores modernos usan report-to e ignoran report-uri.

```

Content-Security-Policy-Report-Only:
  script-src 'nonce-{RANDOM}' 'strict-dynamic' https: 'unsafe-inline';
  object-src 'none';
  base-uri 'none';
  report-uri /csp-reports;
  report-to csp-endpoint
Reporting-Endpoints: csp-endpoint="/csp-reports"

```

Pedir muestras con report-sample

Por defecto, un informe dice que se bloqueó "inline" pero no cuál era el script, lo que dificulta localizarlo. Añadir 'report-sample' a la directiva hace que el navegador incluya los primeros 40 caracteres del código bloqueado en el campo script-sample. Basta para reconocer el fragmento y saber qué plantilla lo generó, sin transmitir el código completo.

Agregar en lugar de registrar

La clave es reducir cada informe a una huella estable: directiva efectiva, URL bloqueada normalizada, ruta de la página (sin parámetros) y la muestra. Contar ocurrencias por huella convierte diez mil informes en veinte filas. Un esquema mínimo en PostgreSQL:

```
CREATE TABLE csp_violations (
  fingerprint text PRIMARY KEY,
  directive text NOT NULL,
  blocked text NOT NULL,
  page_path text NOT NULL,
  sample text,
  first_seen timestampz NOT NULL DEFAULT now(),
  last_seen timestampz NOT NULL DEFAULT now(),
  hits bigint NOT NULL DEFAULT 1
);

-- Upsert por huella: cada informe incrementa un contador en vez de crear una fila
INSERT INTO csp_violations (fingerprint, directive, blocked, page_path, sample)
VALUES ($1, $2, $3, $4, $5)
ON CONFLICT (fingerprint) DO UPDATE
SET hits = csp_violations.hits + 1,
    last_seen = now();
```

```
import hashlib
from urllib.parse import urlsplit

IGNORED_SCHEMES = ("chrome-extension", "moz-extension", "safari-extension", "safari-web-extension")

def normalize_blocked(value: str) -> str:
    """Reduce una URL bloqueada a origen + ruta; deja 'inline' y 'eval' tal cual."""
    if not value or value in ("inline", "eval", "data", "blob", "self"):
        return value or "unknown"
    parts = urlsplit(value)
    if parts.scheme in IGNORED_SCHEMES:
        return "ignored"
    return parts.scheme + "://" + parts.netloc + parts.path

def fingerprint(directive: str, blocked: str, page_path: str, sample: str) -> str:
    raw = "|".join([directive, blocked, page_path, sample[:40]])
    return hashlib.sha256(raw.encode()).hexdigest()[:32]
```

Con esa tabla, la pregunta de cada mañana durante el despliegue es una sola consulta: qué huellas nuevas aparecieron en las últimas 24 horas y cuáles tienen más impacto.

```
SELECT directive, blocked, page_path, sample, hits, last_seen
FROM csp_violations
WHERE first_seen > now() - interval '24 hours'
    AND blocked <> 'ignored'
ORDER BY hits DESC
LIMIT 25;
```

Protege el receptor

El endpoint de informes es una URL pública que acepta POST sin autenticación, así que es un objetivo evidente para el abuso. Limita el tamaño del cuerpo, aplica límites de tasa por IP, valida los tipos de contenido aceptados, no reflejes nunca el contenido del informe en una respuesta y trata cada campo como entrada no confiable antes de mostrarlo en un panel (un informe puede contener HTML malicioso destinado a

quien lo lea). Un detalle más: el navegador envía los informes sin credenciales, de modo que el endpoint no debe depender de cookies de sesión.

Trusted Types a fondo: de la teoría al despliegue

La sección anterior presentó Trusted Types como idea. Aquí van los detalles que necesitas para desplegarlo sin que tu aplicación deje de funcionar. Trusted Types actúa sobre los llamados sumideros de inyección: las APIs del DOM que convierten una cadena en código o en marcado ejecutable. Con la directiva activa, esos sumideros dejan de aceptar cadenas y exigen objetos tipados (TrustedHTML, TrustedScript o TrustedScriptURL) que solo puede fabricar una política registrada.

Qué sumideros cubre

- TrustedHTML: innerHTML, outerHTML, insertAdjacentHTML, document.write, DOMParser.parseFromString y el atributo srcdoc de un iframe.
- TrustedScript: eval, new Function y la propiedad text o textContent de un elemento script.
- TrustedScriptURL: el atributo src de un script, importScripts en workers y la construcción de Worker o SharedWorker.

Es importante notar qué no cubre: textContent sobre un nodo normal, setAttribute con atributos inofensivos o createElement son seguros por naturaleza y no requieren política. La directiva solo endurece lo que realmente puede ejecutar código.

Una política de verdad, no un pase libre

El error más frecuente al adoptar Trusted Types es crear una política cuyo createHTML devuelve la cadena sin tocarla. Eso cumple formalmente con el navegador, pero no protege nada: has movido la vulnerabilidad a un nombre nuevo. Una política útil sanitiza de verdad, o rechaza.

```
// trusted-types.ts: el único archivo que crea políticas
import DOMPurify from "dompurify";

declare global {
  interface Window { trustedTypes?: TrustedTypePolicyFactory }
}

const noop = (s: string) => s;

export const htmlPolicy = window.trustedTypes?.createPolicy("app-html", {
  createHTML: (dirty: string) =>
    DOMPurify.sanitize(dirty, { RETURN_TRUSTED_TYPE: false }),
});

export const scriptUrlPolicy = window.trustedTypes?.createPolicy("app-script-url", {
  createScriptURL: (url: string) => {
    const parsed = new URL(url, location.origin);
    // Solo orígenes que tú controlas; cualquier otra cosa lanza un error
    if (parsed.origin !== location.origin) {
      throw new TypeError("Origen de script no permitido: " + parsed.origin);
    }
    return parsed.toString();
  },
});

export function setHtml(el: Element, dirty: string): void {
  if (htmlPolicy) el.innerHTML = htmlPolicy.createHTML(dirty) as unknown as string;
  else el.textContent = dirty; // sin soporte: texto plano, nunca HTML
}
```

Fíjate en la asimetría de tratamiento: la política de HTML sanitiza porque el HTML es contenido que a veces es legítimo mostrar; la política de URLs de script valida y lanza, porque una URL de script no tiene versión "limpia" posible. Y la ruta sin soporte degrada a texto, no a HTML sin sanear.

La directiva trusted-types y la política por defecto

La directiva trusted-types es la lista de nombres de política permitidos. Si un script intenta crear una política con otro nombre, el navegador lo bloquea. Es lo que impide que un atacante registre su propia política permisiva. Con 'allow-duplicates' permites reutilizar nombres (útil con micro-frontends que se cargan varias veces), pero conviene evitarlo si puedes.

```
Content-Security-Policy:
  require-trusted-types-for 'script';
  trusted-types app-html app-script-url dompurify default
```

La política llamada default es especial: el navegador la invoca automáticamente cuando un sumidero recibe una cadena. Sirve como red de seguridad temporal durante la migración, porque permite que el código antiguo siga funcionando mientras registras, con un log, cada lugar donde cae en ella. La regla de oro es tratarla como una herramienta de descubrimiento, no como destino: cuando el contador de usos baje a cero, retírala.

```
// Solo durante la migración: registra cada uso y sanitiza
window.trustedTypes?.createPolicy("default", {
  createHTML: (s) => {
    console.warn("[trusted-types] sumidero HTML con cadena", new Error().stack);
    navigator.sendBeacon("/tt-reports", JSON.stringify({ sink: "html", stack: new Error().stack }));
    return DOMPurify.sanitize(s);
  },
  createScript: () => { throw new TypeError("eval bloqueado"); },
  createScriptURL: () => { throw new TypeError("script dinámico bloqueado"); },
});
```

Un plan de migración en cuatro pasos

1. Report-Only primero. Despliega Content-Security-Policy-Report-Only: require-trusted-types-for 'script'. Cada violación indica un sumidero que recibe una cadena. Agrégalas con la misma técnica de huellas que en los informes de script.
2. Elimina los sumideros evitables. La mayoría de los innerHTML con texto plano se resuelven con textContent, y los de plantillas con construcción de nodos o un motor de plantillas que escape por defecto. Es el paso más barato y el que más reduce el volumen.
3. Centraliza lo inevitable. Para el HTML que de verdad debes mostrar (comentarios con formato, contenido de un CMS), pasa por una única función que use la política con sanitizado. Un único punto es auditable; veinte puntos no lo son.
4. Aplica y vigila. Cambia a la cabecera de bloqueo, mantén el informe activo y añade a CI una regla de lint que prohíba asignar a innerHTML fuera de ese módulo.

Frameworks y librerías

React escapa por defecto y solo expone un sumidero explícito, dangerouslySetInnerHTML, que debes pasar por la política. Angular tiene integración nativa con Trusted Types y su propia política interna. Lit y otras librerías de plantillas con etiquetas crean sus propias políticas, así que deberás añadir sus nombres a la directiva. Para las dependencias que escriben en innerHTML internamente, revisa si publican una versión compatible; si no, la política default con sanitizado te cubre mientras tanto.

El estado del soporte

Trusted Types nació en Chromium y durante años fue una característica exclusiva de ese motor. El soporte en otros navegadores ha ido llegando de forma escalonada; comprueba el estado actual en MDN o en las tablas de compatibilidad antes de decidir cuánto confías en la capa. La consecuencia de diseño es la misma sea cual sea la tabla: trata Trusted Types como endurecimiento que se suma a un código ya correcto, y mantén siempre la ruta de degradación segura (texto plano) para los navegadores donde la API no exista.

Lo que CSP no hace: bypasses reales y cómo pensarlos

Una CSP estricta cierra la mayoría de las vías del XSS clásico, pero no es una barrera mágica. Conocer los ataques que sobreviven te ayuda a decidir qué complementar.

Marcado colgante (dangling markup)

Si el atacante puede inyectar una etiqueta sin cerrar, como `<img src='https://evil.example/?'`, el navegador consume el HTML siguiente como parte de la URL, incluyendo tokens CSRF o datos de la página, y los envía al atacante sin ejecutar un solo script. Una CSP centrada en `script-src` no lo detiene. Lo mitigan `img-src` restrictivo, `form-action` y, sobre todo, escapar correctamente la salida. Es el ejemplo perfecto de por qué CSP es defensa en profundidad y no sustituto de la codificación.

Gadgets de script y confianza heredada

Con 'strict-dynamic' la confianza se propaga: si un script legítimo de tu aplicación lee una cadena del DOM y la usa para crear un script nuevo, el atacante que controla esa cadena controla el script. Se llaman gadgets. Los patrones peligrosos son los que construyen rutas de script a partir de atributos `data-`, de parámetros de la URL o de contenido del DOM. La defensa es revisar tu propio código de carga dinámica y validar contra una lista cerrada, no confiar en lo que haya en el documento.

Inyección de la etiqueta base y de scripts que reutilizan nonces

Sin `base-uri`, una etiqueta `<base href="https://evil.example/">` inyectada cambia la resolución de todas las URL relativas, y un script con nonce válido cargado de forma relativa pasa a descargarse del atacante. Por eso `base-uri` 'none' está en toda política estricta. De modo parecido, ciertos ataques intentan leer el nonce desde el DOM mediante selectores CSS o ataques de caché del navegador para reutilizarlo; la mejor defensa es que el nonce no se filtre (nada de volcarlo en URLs, cabeceras de caché compartidas ni logs) y que la vida útil sea una sola respuesta.

JSONP, Angular antiguo y endpoints reflectantes

Son los bypasses clásicos de las listas de hosts, y justo la razón por la que un nonce con 'strict-dynamic' las sustituye. Si en una migración mantienes durante un tiempo alguna lista de hosts, pasa tu política por un evaluador como el CSP Evaluator de Google: marca automáticamente los hosts conocidos por servir endpoints explotables.

El DOM clobbering y otras primitivas

Un atacante que solo puede inyectar elementos HTML inertes (por ejemplo, una etiqueta con un atributo `id` o `name`) puede sobrescribir variables globales y alterar el flujo de tu JavaScript sin ejecutar nada propio. CSP no ve esto como una violación. El remedio es el de siempre: sanitizar con una lista de permitidos estricta (DOMPurify, por ejemplo, elimina `id` y `name` peligrosos en su configuración recomendada) y no confiar en variables globales que puedan ser sobrescritas.

Cómo evaluar tu propia política

Dos herramientas gratuitas cubren casi todo. El CSP Evaluator de Google analiza una política y señala las directivas débiles, con explicación de cada bypass. Y una auditoría con Lighthouse o una prueba como la de Playwright que ya hemos visto comprueba que la política desplegada es la esperada. Úsalas como parte de la revisión de cada cambio, no una sola vez al inicio.

Cabeceras hermanas: lo que completa a CSP

CSP rara vez va sola. Estas cabeceras cubren riesgos adyacentes y suelen desplegarse en el mismo proyecto:

- X-Content-Type-Options: nosniff impide que el navegador adivine tipos MIME y ejecute como script un

archivo servido como texto. Complementa a script-src, porque cierra la vía de subir un "archivo de texto" que se interpreta como JavaScript.

- Referrer-Policy controla cuánta información de la URL de origen se envía a otros sitios. strict-origin-when-cross-origin es un valor por defecto razonable.
- Permissions-Policy deshabilita capacidades del navegador (cámara, micrófono, geolocalización) que tu aplicación no usa, de modo que ni un script inyectado ni un iframe puedan pedirlos.
- Cross-Origin-Opener-Policy y Cross-Origin-Embedder-Policy aíslan tu documento de otras ventanas y habilitan el aislamiento entre orígenes, relevante frente a ataques de canal lateral.
- Strict-Transport-Security obliga a usar HTTPS, y tiene sentido junto a upgrade-insecure-requests.

```
X-Content-Type-Options: nosniff
Referrer-Policy: strict-origin-when-cross-origin
Permissions-Policy: camera=(), microphone=(), geolocation=(), payment=()
Cross-Origin-Opener-Policy: same-origin
Strict-Transport-Security: max-age=63072000; includeSubDomains
```

Caso práctico: migrar una aplicación de 40 páginas a CSP estricta en cinco semanas

Un escenario ilustrativo (las cifras son un ejemplo plausible, no datos de un cliente concreto) ayuda a ver cómo encajan las piezas. Imagina una aplicación web con un panel de usuario, un blog y un flujo de pago, unas cuarenta plantillas de servidor, un gestor de etiquetas con siete proveedores y una política actual basada en una lista de hosts de veintidós entradas.

Semana 1: inventario y Report-Only. El equipo despliega la política estricta en modo Report-Only con el receptor agregador. El primer día llegan miles de informes, pero la tabla de huellas los reduce a unas sesenta filas distintas. Un tercio es ruido de extensiones del navegador, que se filtra de inmediato. Quedan unas cuarenta huellas reales, de las cuales casi la mitad son manejadores onclick en tres plantillas compartidas.

Semana 2: refactorizar lo barato. Los manejadores en línea se sustituyen por listeners delegados en un único archivo. Dos plantillas que usaban eval para evaluar expresiones de configuración se reescriben con JSON. El volumen de huellas baja a una docena. Lo que queda son etiquetas de terceros.

Semana 3: terceros. Se sustituye el fragmento pegado de cada proveedor por el cargador único con nonce. Dos proveedores usan document.write: uno publica una versión compatible, el otro se aísla en un iframe con sandbox. Los permisos de publicación del gestor de etiquetas se reducen a dos personas con doble revisión.

Semana 4: pruebas y ensayo. Se añade la prueba de Playwright a CI para las diez rutas críticas, incluido el flujo de pago completo. Se activa el bloqueo en un entorno de pruebas durante dos días con el equipo de producto recorriendo la aplicación. Aparecen dos violaciones nuevas en rutas poco visitadas, que se corrigen.

Semana 5: despliegue por porcentaje. El bloqueo se activa con un interruptor de funcionalidad: primero el 5% del tráfico, luego el 25%, 50% y 100%, con una pausa de un día en cada escalón mientras se vigilan la tasa de errores de JavaScript y las conversiones del pago. Al llegar al 100%, el flujo de informes se mantiene activo para siempre.

Lo importante de este recorrido no son los números, sino el orden: medir primero, eliminar lo evitable, centralizar lo inevitable y desplegar de forma gradual con una vía de retroceso. Ninguno de los pasos requiere conocimientos de seguridad especiales; requiere disciplina de ingeniería normal.

El interruptor de despliegue por porcentaje

Poder pasar de Report-Only a bloqueo para un porcentaje de usuarios es lo que convierte el despliegue en algo reversible en segundos. Basta una función que decida la cabecera según un identificador estable del usuario:

```
import hashlib
import os

ENFORCE_PERCENT = int(os.environ.get("CSP_ENFORCE_PERCENT", "0"))

def csp_header_name(visitor_id: str) -> str:
    """Devuelve la cabecera a usar para este visitante.
    El hash hace que el mismo visitante caiga siempre en el mismo grupo."""
    bucket = int(hashlib.sha256(visitor_id.encode()).hexdigest(), 16) % 100
    if bucket < ENFORCE_PERCENT:
        return "Content-Security-Policy"
    return "Content-Security-Policy-Report-Only"
```

Con la variable de entorno en cero, nadie está bloqueado; con cien, todos. Revertir un despliegue problemático es cambiar un número y reiniciar, sin tocar código ni esperar a un nuevo build.

Runbook: qué hacer cuando una CSP rompe producción

Tarde o temprano una política rompe algo. Tener el procedimiento escrito de antemano evita decisiones apresuradas, como desactivar CSP por completo.

1. Confirma que es CSP. Abre la consola del navegador: las violaciones aparecen como errores que empiezan por "Refused to execute" o "Refused to load". Si no hay ninguno, el problema es otro.
2. Reduce el alcance. Baja el porcentaje de bloqueo a cero con el interruptor. Los usuarios recuperan el servicio en segundos y la política sigue informando en Report-Only.
3. Identifica la huella. Consulta la tabla de violaciones ordenada por última aparición. La directiva y la URL bloqueada casi siempre explican la causa: una etiqueta nueva de marketing, un cambio de CDN, una plantilla con un manejador en línea.
4. Corrige en el origen. Añade nonce, mueve el código a un archivo o actualiza la dependencia. Evita la tentación de añadir 'unsafe-inline' o un comodín: resuelven el síntoma y anulan la protección.
5. Añade una prueba. Convierte el caso en una prueba de CI para que no vuelva a pasar desapercibido.
6. Documenta. Una línea en el registro de cambios de la política con la fecha, la causa y la corrección. Dentro de un año, nadie recordará por qué existe una excepción.

Métricas que importan

Una CSP desplegada sin métricas se pudre sin que nadie lo note. Cuatro números bastan para saber si la política sigue sana:

- Huellas nuevas por día. En régimen estable debería ser cercano a cero. Un pico significa un despliegue que añadió algo sin pasar por el cargador con nonce.
- Violaciones por cada mil páginas vistas. Normaliza por tráfico para comparar semanas distintas.
- Porcentaje de respuestas HTML con la cabecera. Una caída indica una ruta nueva, un proxy o una caché que quita la cabecera.
- Antigüedad de la excepción más vieja. Cada excepción temporal (una entrada en connect-src, un 'unsafe-hashes') debería tener fecha y responsable. Una excepción sin fecha de caducidad es permanente.

Plan de adopción en 30 días

Para un equipo que parte de cero, este calendario es realista y deja un margen de seguridad:

- Días 1 a 3: cabeceras hermanas (nosniff, Referrer-Policy, frame-ancestors) y receptor de informes. Sin riesgo, con valor inmediato.
- Días 4 a 14: política estricta en Report-Only, agregación de huellas y primera tanda de refactorizaciones.
- Días 15 a 21: cargador de terceros con nonce, iframes con sandbox para lo que no cabe y prueba de CI.
- Días 22 a 28: bloqueo gradual por porcentaje con vigilancia de métricas.
- Día 29 en adelante: Trusted Types en Report-Only y revisión mensual de las huellas nuevas.

Preguntas frecuentes

¿Puedo usar una CSP estricta con una aplicación de una sola página servida como HTML estático? Sí, con hashes para los scripts en línea y SRI para los externos, o poniendo un Worker o un servidor ligero delante que inyecte nonces. Lo que no puedes hacer es un nonce fijo en el HTML cacheado.

¿Qué pasa con los estilos en línea? Son mucho menos peligrosos que los scripts, y es razonable mantener style-src 'unsafe-inline' al principio. Si quieres endurecerlos, usa nonces o hashes también para estilos, pero hazlo después de resolver los scripts.

¿Una CSP reemplaza al WAF o a la sanitización? No. Es una capa más. La codificación de salida evita la vulnerabilidad; CSP limita el daño si falla; el WAF filtra tráfico conocido; y los informes te dicen dónde aprender.

¿Cuánto tiempo debo dejar Report-Only? El suficiente para cubrir un ciclo completo de tu tráfico: al menos una semana, y mejor dos si hay tareas mensuales (facturación, informes) que usan rutas poco frecuentes.

¿Los informes pueden filtrar datos personales? Sí: la URL del documento y la URL bloqueada pueden contener identificadores en la ruta o en parámetros. Normaliza o recorta antes de guardar, y define una retención corta.

¿Es seguro mantener https: en script-src? Sí, siempre que haya nonce o hash y 'strict-dynamic': un navegador con CSP3 lo ignora. Solo actúa como respaldo para navegadores antiguos.

Glosario

- CSP (Content Security Policy): cabecera que restringe qué recursos puede cargar y ejecutar una página.
- Nonce: valor aleatorio, distinto en cada respuesta, que marca los scripts permitidos.
- Hash: huella criptográfica del contenido de un script en línea que lo autoriza sin nonce.
- strict-dynamic: palabra clave que propaga la confianza de un script autorizado a los scripts que este cargue.
- Report-Only: modo en el que el navegador informa de violaciones sin bloquear nada.
- Sumidero (sink): API del DOM que convierte una cadena en código o marcado ejecutable.
- Trusted Types: mecanismo que exige objetos tipados, creados por políticas, para usar los sumideros.
- SRI (Subresource Integrity): atributo que verifica el hash de un recurso externo antes de ejecutarlo.
- Gadget: fragmento de código legítimo que un atacante puede aprovechar para ejecutar su carga.
- Dangling markup: inyección de una etiqueta sin cerrar para exfiltrar contenido sin ejecutar scripts.

Pruebas más allá de la home: una batería que recorre toda la aplicación

La prueba de Playwright del principio comprueba una sola página. Una política estricta se rompe en las rutas

que nadie mira: el correo de restablecimiento de contraseña, el panel de administración, la página de error 500. Una batería que recorre una lista de rutas críticas y falla ante cualquier violación detecta esos casos antes que tus usuarios.

```
// csp-routes.spec.ts
import { test, expect } from "@playwright/test";

const ROUTES = ["/", "/login", "/precios", "/blog", "/checkout", "/cuenta", "/404-que-no-existe"];

for (const route of ROUTES) {
  test("sin violaciones de CSP en " + route, async ({ page }) => {
    const violations: string[] = [];

    page.on("console", (msg) => {
      const text = msg.text();
      if (text.includes("Content Security Policy") || text.includes("Content-Security-Policy")) {
        violations.push(text);
      }
    });

    await page.addInitScript(() => {
      document.addEventListener("securitypolicyviolation", (e) => {
        (window as any).__csp = ((window as any).__csp ?? []).concat(
          e.effectiveDirective + " <- " + (e.blockedURI || "inline") + " @ " + e.sourceFile + ":" +
            e.lineNumber
        );
      });
    });

    await page.goto(route, { waitUntil: "networkidle" });
    // Interacciones mínimas que disparan código diferido
    await page.mouse.move(100, 100);
    await page.keyboard.press("Tab");

    const fromEvents = await page.evaluate(() => (window as any).__csp ?? []);
    expect([...violations, ...fromEvents]).toEqual([]);
  });
}
```

Dos matices útiles. El evento `securitypolicyviolation` incluye `sourceFile` y `lineNumber`, que apuntan al código culpable, algo que el informe enviado al servidor no siempre trae. Y las interacciones mínimas (mover el ratón, pulsar una tecla) despiertan scripts que cargan de forma diferida y que de otro modo no aparecerían en una navegación pasiva.

Para las plantillas de correo electrónico y los documentos generados, que no pasan por el navegador, añade una comprobación estática: un script que busque en el HTML renderizado atributos `on*=` y etiquetas `<script>` sin nonce. Es una expresión regular sencilla que bloquea el 90% de las regresiones en la revisión de código, antes incluso de ejecutar un navegador.

Casos límite: service workers, WebAssembly y estilos dinámicos

Tres situaciones desconciertan a casi todos los equipos la primera vez que las encuentran con una política estricta.

Service workers. Un service worker tiene su propia política, que es la cabecera de la respuesta que sirve el archivo del worker, no la de la página que lo registra. Si sirves `/sw.js` sin CSP, el worker se ejecuta sin restricciones aunque tu página esté blindada. Aplica las cabeceras también a los scripts de workers y sirve ese archivo con `Cache-Control: no-cache` para que las actualizaciones lleguen rápido.

WebAssembly. Compilar y ejecutar módulos Wasm requiere la palabra clave `'wasm-unsafe-eval'` en `script-src`. Es mucho más acotada que `'unsafe-eval'`: no habilita eval de JavaScript, solo la compilación de Wasm. Si tu aplicación usa un módulo Wasm (un editor de imágenes, un motor de base de datos en el

navegador), añade esa palabra clave y nada más.

Estilos que cambian desde JavaScript. Asignar `element.style.color = "red"` es siempre legal, porque manipula el objeto de estilo y no analiza texto CSS. Lo que se bloquea es `element.setAttribute("style", "...")` o insertar una etiqueta `<style>` sin nonce cuando `style-src` es estricto. Casi todas las librerías de componentes modernas se adaptan, pero conviene probarlas con la política activa antes de endurecer los estilos.

Conclusiones

Olvida las listas de hosts y construye sobre nonces más 'strict-dynamic'. Despliega en modo Report-Only, aprende de los informes y después aplica el bloqueo. Mantén el HTML fuera de cachés compartidas para que los nonces sean únicos y pon una prueba en CI para que la política no se pudra. Bien hecha, CSP convierte un escape olvidado de una brecha en una línea de tu registro de violaciones.

Strict Content Security Policy

Nonces, strict-dynamic, Report-Only and Trusted Types against XSS

PuigFlows - English edition - 17/10/2026

Why a Content Security Policy still matters

Cross-site scripting keeps showing up in bug-bounty reports for one boring reason: one forgotten escape in one template is enough. A Content Security Policy (CSP) is the safety net underneath that mistake. It is an HTTP response header that tells the browser which scripts it may run, so that even when an attacker manages to inject markup, the browser refuses to execute it.

CSP is defense in depth, not a replacement for output encoding. You still escape, sanitize and validate. CSP limits the blast radius when you get it wrong, and it also gives you something rare in security: a stream of reports telling you where it went wrong.

Why allowlist policies fail

The policy most people write first looks like this: `script-src 'self' https://cdn.example.com https://www.google-analytics.com`. It feels precise, but studies of real-world policies keep finding the same flaw: any allowed host that serves a JSONP endpoint, an old Angular build or a user-controlled file becomes a bypass. A host allowlist trusts everything that host will ever serve, and you do not control that.

The modern answer is a strict CSP built on two ideas. First, every legitimate script carries a random per-response nonce. Second, 'strict-dynamic' lets a script that was already trusted load further scripts, so you stop enumerating hosts entirely.

A strict policy you can start from

```
Content-Security-Policy:
  script-src 'nonce-{RANDOM}' 'strict-dynamic' https: 'unsafe-inline';
  object-src 'none';
  base-uri 'none';
  frame-ancestors 'self';
  report-to csp-endpoint
```

Read it directive by directive:

- `script-src 'nonce-...' 'strict-dynamic'` runs only scripts carrying the nonce, plus whatever those scripts load dynamically.
- `https: 'unsafe-inline'` looks scary but is a fallback. Browsers that understand nonces ignore both; only very old browsers use them, and they would otherwise break completely.
- `object-src 'none'` kills plugin content, a classic bypass route.
- `base-uri 'none'` stops an injected `<base>` tag from redirecting every relative script URL to the attacker.
- `frame-ancestors` is the modern replacement for X-Frame-Options. Note that it works only as a header, never in a `<meta>` tag.

Generating the nonce correctly

The rules are short and unforgiving: at least 128 bits from a cryptographically secure generator, a fresh value on every response, and the same value in the header and in every nonce attribute. A nonce that repeats is just a password you published.

```
// server.mjs (Node 20+, Express)
import crypto from "node:crypto";
import express from "express";

const app = express();
const ENFORCE = process.env.CSP_ENFORCE === "true";

app.use((req, res, next) => {
```

```

const nonce = crypto.randomBytes(16).toString("base64");
res.locals.cspNonce = nonce;

const policy = [
  "script-src 'nonce-" + nonce + "' 'strict-dynamic' https: 'unsafe-inline'",
  "object-src 'none'",
  "base-uri 'none'",
  "frame-ancestors 'self'",
  "report-to csp-endpoint",
].join("; ");

// Report-Only first: the browser reports violations but blocks nothing.
res.setHeader(
  ENFORCE ? "Content-Security-Policy" : "Content-Security-Policy-Report-Only",
  policy
);
res.setHeader("Reporting-Endpoints", 'csp-endpoint="/csp-reports"');
// A cached page would replay an old nonce, so keep HTML out of shared caches.
res.setHeader("Cache-Control", "private, no-cache");
next();
});

```

In a template you then write `<script nonce="<%= cspNonce %>" src="/app.js"></script>`. The same idea in Python with FastAPI:

```

# main.py
import base64, logging, os
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse, Response

app = FastAPI()
log = logging.getLogger("csp")

@app.middleware("http")
async def csp_headers(request: Request, call_next):
    nonce = base64.b64encode(os.urandom(16)).decode()
    request.state.csp_nonce = nonce
    response = await call_next(request)
    response.headers["Content-Security-Policy-Report-Only"] = "; ".join([
        f"script-src 'nonce-{nonce}' 'strict-dynamic' https: 'unsafe-inline'",
        "object-src 'none'",
        "base-uri 'none'",
        "frame-ancestors 'self'",
        "report-to csp-endpoint",
    ])
    response.headers["Reporting-Endpoints"] = 'csp-endpoint="/csp-reports"'
    response.headers["Cache-Control"] = "private, no-cache"
    return response

@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    n = request.state.csp_nonce
    return f'<!doctype html><title>Demo</title><script nonce="{n}" src="/static/app.js"></script>'

```

Roll out in Report-Only mode first

Turning on enforcement on day one is how teams break checkout pages and then switch CSP off for good. Use Content-Security-Policy-Report-Only first. The browser evaluates the policy, sends a report for every violation and blocks nothing. Run it for a week or two, fix what shows up, then flip enforcement.

The modern reporting pipeline pairs the Reporting-Endpoints header with the report-to directive, as above. Older browsers still use the legacy report-uri directive and a different payload, so a robust receiver accepts both:

```

@app.post("/csp-reports", status_code=204)
async def csp_reports(request: Request):
    raw = await request.body()
    if len(raw) > 64_000:
        return Response(status_code=413)
    try:
        payload = await request.json()
    except ValueError:
        return Response(status_code=400)

    # Reporting API: a list of {"type": "csp-violation", "body": {...}}
    # Legacy report-uri: {"csp-report": {...}}
    items = payload if isinstance(payload, list) else [payload]
    for item in items:
        body = item.get("body") or item.get("csp-report") or {}
        log.warning(
            "csp_violation",
            extra={
                "directive": body.get("effectiveDirective") or body.get("effective-directive"),
                "blocked": body.get("blockedURL") or body.get("blocked-uri"),
                "page": body.get("documentURL") or body.get("document-uri"),
            },
        )

```

Expect noise. Browser extensions inject scripts and trigger violations that are not your bug. Group reports by directive and blocked URL, and ignore sources such as `chrome-extension://` or `moz-extension://` instead of chasing them.

Refactoring what breaks

Once reports start arriving, three patterns account for most of the work.

- Inline event handlers. Attributes like `onclick` are blocked, and a nonce cannot legitimize them. Move the logic into a script and attach listeners.
- Inline scripts without a nonce. Add the attribute, or move the code into an external file.
- `eval` and friends. `eval()`, `new Function()` and string-based `setTimeout` need `'unsafe-eval'`. Remove them rather than allowing them; bundlers can usually be configured to avoid eval-based source maps in production.

```

<!-- Before: blocked by CSP -->
<button onclick="save()">Save</button>

<!-- After -->
<button id="save">Save</button>
<script nonce="{{ nonce }}">
    document.getElementById("save").addEventListener("click", save);
</script>

```

Styles deserve a note: a strict `script-src` is the priority, and many teams keep a looser `style-src 'self' 'unsafe-inline'` for a while because inline styles are far less dangerous than inline scripts. Tighten it later with nonces or hashes if you can.

Trusted Types: closing DOM XSS at the sink

A nonce-based CSP stops injected script tags, but DOM-based XSS happens when your own code writes attacker data into `innerHTML`. Trusted Types attacks that class directly: with the right directives, dangerous sinks accept only objects produced by a policy you defined, and plain strings throw.

```
Content-Security-Policy: require-trusted-types-for 'script'; trusted-types app-html
```

```
import DOMPurify from "dompurify";

// Feature-detect: support varies across browsers, so treat this as hardening.
const policy = window.trustedTypes?.createPolicy("app-html", {
  createHTML: (dirty) => DOMPurify.sanitize(dirty),
});

function renderComment(el, userComment) {
  if (policy) {
    el.innerHTML = policy.createHTML(userComment); // allowed: sanitized by the policy
  } else {
    el.textContent = userComment; // safe fallback: no HTML at all
  }
}
```

The big win is auditability. Instead of reviewing every line for dangerous assignments, you review the handful of policies, and the browser enforces the rest.

Test the policy in CI

A CSP that silently regresses is worse than none, because the team believes it is protected. A small end-to-end test catches the common regressions: a new inline script, a third-party tag added by marketing, a header dropped by a proxy.

```
// csp.spec.ts (Playwright)
import { test, expect } from "@playwright/test";

test("home page has a strict CSP and no violations", async ({ page }) => {
  await page.addInitScript(() => {
    document.addEventListener("securitypolicyviolation", (e) => {
      const w = window as any;
      w.__csp = (w.__csp ?? []).concat(e.effectiveDirective + " <- " + e.blockedURI);
    });
  });

  const res = await page.goto("/");
  const header =
    res?.headers()["content-security-policy"] ??
    res?.headers()["content-security-policy-report-only"];

  expect(header).toContain("'strict-dynamic'");
  expect(header).toContain("object-src 'none'");

  await page.waitForLoadState("networkidle");
  const violations = await page.evaluate(() => (window as any).__csp ?? []);
  expect(violations).toEqual([]);
});
```

Common pitfalls

- Static nonces. A nonce baked into a build, or cached by a CDN along with the HTML, protects nothing. If you serve static HTML, use hashes ('sha256-...') instead of nonces.
- Policy in a <meta> tag. It works for some directives but cannot carry frame-ancestors, report-to or sandbox, and it applies only after the tag is parsed.
- Allowing 'unsafe-inline' without a nonce. In a modern browser, a nonce or hash makes 'unsafe-inline' ignored; without one, it disables the protection entirely.
- Wildcards and data: in script-src. Each is a ready-made bypass.
- Multiple headers. When several CSP headers arrive, the browser enforces all of them, so the result is the intersection. A proxy adding its own header can break your pages.
- Forgetting the other directives. default-src, connect-src, img-src and form-action limit exfiltration and rogue form posts. Start strict on scripts, then tighten the rest.

Production checklist

- Nonce is 128+ bits, random per response, and appears in the header and in every script tag.
- HTML responses are never served from a shared cache with a stale nonce.
- object-src 'none', base-uri 'none' and frame-ancestors are set.
- Report-Only ran long enough to cover real traffic; reports are collected, size-limited and grouped.
- No inline handlers and no eval remain; third-party tags load through a nonced loader.
- A CI test fails the build on violations or a missing header.
- Trusted Types is evaluated for any code that writes to innerHTML.

How the browser evaluates a CSP: the mental model that prevents 90% of mistakes

Most CSP problems do not come from syntax but from a wrong mental model of when and how the browser decides. It is worth understanding once, precisely, because it explains why a "strict" policy works and why one that looks equivalent does not.

Every time the browser is about to do something CSP regulates (run a script, load an image, open a connection, submit a form), it consults the policy attached to the document. Each kind of action belongs to a specific directive: script-src for scripts, img-src for images, connect-src for fetch, XHR and WebSocket, and so on. If the specific directive is missing, the browser falls back to default-src; if that is missing too, the action is allowed. That last sentence matters: a policy that only defines script-src restricts nothing else.

For scripts, the decision follows an order worth memorizing:

1. If the script carries a nonce that matches the policy's, or its content matches a declared hash, it is allowed.
2. If the script was inserted by another script that was already trusted and the policy includes 'strict-dynamic', it is allowed even without a nonce.
3. If the policy contains a host allowlist and the script's origin matches, it is allowed, unless 'strict-dynamic' is present: in that case the host allowlist is ignored entirely.
4. Otherwise it is blocked and, if there is a reporting destination, a violation is sent.

Step 3 is why the starting policy in this article can include https: and 'unsafe-inline' without weakening it. A CSP3-capable browser sees the nonce, switches on the nonce plus 'strict-dynamic' logic and ignores both https: and 'unsafe-inline'. A ten-year-old browser does not understand nonces and falls back to those two values, which keep the site working albeit with much weaker protection. That is graceful degradation, not a back door.

Parser-inserted versus dynamically inserted scripts

The subtlest part of 'strict-dynamic' is who inserted the script. A script that appears in the original HTML, processed by the parser, needs a nonce or hash on its own. A script created from JavaScript with document.createElement("script") inherits trust from the script that created it, as long as the creator was trusted. That is what lets one nonced loader pull in fifty dependencies without listing them.

There is one deliberate exception: scripts inserted through APIs that parse HTML, such as document.write or innerHTML with a script tag, count as "parser-inserted" and lose the inherited trust. If a third-party library relies on document.write to load more scripts, 'strict-dynamic' will break it and you will see a clear violation in your reports. The right fix is to replace it or isolate it in a sandboxed iframe, not to relax the policy.

What each browser generation ignores

To design the fallback deliberately, it helps to have the table in front of you:

- CSP1 (very old browsers): only understands host lists and 'unsafe-inline'. It sees https: and 'unsafe-inline' and behaves like a loose policy.
- CSP2: adds nonces and hashes. When a nonce is present, 'unsafe-inline' is ignored, but 'strict-dynamic' is unknown, so it keeps using the host list (https:).
- CSP3 (every current browser): understands 'strict-dynamic', ignores the host list and 'unsafe-inline' when a nonce is present, and is the logic you actually want to run.

The practical consequence is that the strict policy is evaluated against your users' current browser, not the oldest one. Check your browser analytics before worrying about the fallback: on most sites the share of browsers without CSP3 is negligible.

Hashes, SRI and static HTML: when a nonce is not possible

A nonce requires the server to generate fresh HTML on every request. If you serve a static site from a CDN or a bucket, there is no server to generate anything, and caching HTML with a fixed nonce defeats the protection. For that case there are hashes: instead of tagging each script with a random value, you declare in the header the SHA-256, SHA-384 or SHA-512 of the exact content of each allowed inline script.

```
Content-Security-Policy:
  script-src 'sha256-3NaxZ2gH5tLw8fQyqk4nV0ZbH8m1ZrW0eQ0jq7dY9xE=' 'strict-dynamic' https;;
  object-src 'none';
  base-uri 'none'
```

The hash is computed over the exact bytes between the <script> and </script> tags, including spaces and newlines. Changing a single space invalidates the hash, which is a virtue (nobody can alter the script without the policy noticing) and a nuisance (any build reformatting breaks the policy). That is why you should compute hashes automatically in the same build step that produces the HTML:

```
// scripts/csp-hashes.mjs
// Walks the generated HTML, hashes every inline script
// and writes a header ready to paste into the CDN.
import { createHash } from "node:crypto";
import { readdirSync, readFileSync, writeFileSync } from "node:fs";
import { join } from "node:path";

const DIST = "dist";
const hashes = new Set();

function walk(dir) {
  for (const entry of readdirSync(dir, { withFileTypes: true })) {
    const full = join(dir, entry.name);
    if (entry.isDirectory()) walk(full);
    else if (entry.name.endsWith(".html")) scan(full);
  }
}

function scan(file) {
  const html = readFileSync(file, "utf8");
  // Inline scripts only (no src attribute)
  const re = /<script(?:![^>]*\bsrc=)[^>]*>([\s\S]*)</script>/gi;
  let m;
  while ((m = re.exec(html))) {
    const body = m[1];
    if (!body.trim()) continue;
    const digest = createHash("sha256").update(body, "utf8").digest("base64");
    hashes.add("sha256-" + digest + "");
  }
}

walk(DIST);
const header = "script-src " + [...hashes].join(" ") + " 'strict-dynamic' https;; object-src 'none';
base-uri 'none'";
writeFileSync("csp-header.txt", header + "\n");
console.log("Unique hashes: " + hashes.size);
```

Two warnings about this approach. First, a policy with hundreds of hashes grows large and you pay for it on every response; if you pass a few dozen, consider a lightweight server render that sets nonces, or an edge Worker. Second, the hash covers inline scripts, not handlers such as onclick. 'unsafe-hashes' exists to allow specific handlers by hash, but it widens the surface and should be treated as a temporary crutch while you refactor.

Subresource Integrity: the natural companion for external scripts

CSP hashes protect inline content; Subresource Integrity (SRI) protects external content. The integrity attribute makes the browser compare the downloaded file's hash against the declared one and refuse to run it on mismatch, which neutralizes the compromised-CDN scenario.

```
<script
  nonce="{{ nonce }}"
  src="https://cdn.example.com/lib@4.2.1/lib.min.js"
  integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQh01wx4JwY8wC"
  crossorigin="anonymous"></script>
```

For SRI to work, the version must be pinned: with an unversioned URL that the vendor updates, the hash will stop matching the day of the next upstream release and your page will silently lose that library. The rule is simple: SRI for what you version, and for what you cannot version, host the copy on your own origin.

A CI test that walks the HTML and fails when it finds a cross-origin script without integrity turns that rule into a guarantee instead of a good intention.

The complete directive map: beyond script-src

A strict script policy is the priority, but a complete CSP also limits where information can leave and which resources can load. These are the directives that earn their keep in practice, grouped by the question they answer.

What can the page load? default-src is the safety net for everything without its own directive. img-src, font-src, media-src, style-src and worker-src control each resource type. manifest-src scopes the web app manifest. frame-src decides which pages you can embed, and child-src exists mostly for compatibility.

Where can the page talk to? connect-src governs fetch, XHR, WebSocket, EventSource and navigator.sendBeacon. It is the directive that limits exfiltration the most: an attacker who manages to run code but cannot open connections to their server has a much harder time getting data out. form-action does the same for form submissions, which default-src does not cover.

Who can embed you? frame-ancestors replaces X-Frame-Options and defends against clickjacking. Typical values are 'none' (nobody), 'self' (your origin only) or a specific list of partner origins.

What execution environment does the page have? sandbox applies iframe-style restrictions to the whole document, upgrade-insecure-requests rewrites http: URLs to https:, and require-trusted-types-for switches on Trusted Types, covered below.

A reasonable baseline for a web app with its API on the same domain, an image CDN and an analytics provider looks like this:

```
Content-Security-Policy:
  default-src 'self';
  script-src 'nonce-{RANDOM}' 'strict-dynamic' https: 'unsafe-inline';
  style-src 'self' 'unsafe-inline';
  img-src 'self' data: https://images.example-cdn.com;
  font-src 'self';
  connect-src 'self' https://api.example.com https://analytics.example.net;
  frame-src 'none';
  form-action 'self';
  frame-ancestors 'none';
  object-src 'none';
```

```
base-uri 'none';
upgrade-insecure-requests;
report-to csp-endpoint
```

Two details about this policy. First, `img-src` includes `data:` because many apps embed small icons; `data:` images do not execute code, so the risk is far lower than with `data:` in `script-src`. Second, the `navigate-to` directive appears in old drafts but no production browser implements it; if you see it in a guide, ignore it.

Nonces in practice: frameworks and platforms

The pattern is the same everywhere (one random value per request, placed in the header and on every tag), but each framework has its own way of doing it without fighting you. These are the cases that come up most.

Next.js (App Router)

Next.js generates the nonce in a middleware and propagates it to server components through a request header. When the framework sees the nonce in the request's Content-Security-Policy header, it applies it automatically to its own scripts. One crucial detail: rendering must be dynamic, because a page generated statically at build time cannot carry a different nonce per request.

```
// middleware.ts
import { NextRequest, NextResponse } from "next/server";

export function middleware(request: NextRequest) {
  const nonce = Buffer.from(crypto.randomUUID()).toString("base64");
  const isDev = process.env.NODE_ENV === "development";

  const csp = [
    "default-src 'self'",
    "script-src 'self' 'nonce-" + nonce + "' 'strict-dynamic'" + (isDev ? " 'unsafe-eval'" : ""),
    "style-src 'self' 'unsafe-inline'",
    "img-src 'self' blob: data:",
    "object-src 'none'",
    "base-uri 'none'",
    "form-action 'self'",
    "frame-ancestors 'none'",
    "upgrade-insecure-requests",
  ].join("; ");

  const headers = new Headers(request.headers);
  headers.set("x-nonce", nonce);
  headers.set("Content-Security-Policy", csp);

  const response = NextResponse.next({ request: { headers } });
  response.headers.set("Content-Security-Policy", csp);
  return response;
}

export const config = {
  matcher: [{
    source: "/((?!api|_next/static|_next/image|favicon.ico).*)",
    missing: [{ type: "header", key: "next-router-prefetch" }],
  }],
};
```

Note that `'unsafe-eval'` is only added in development, where hot-reload tooling needs it. In production it must go away. In a server component you read the nonce with `headers().get("x-nonce")` and pass it to any `<Script>` tag you add yourself.

Express with Helmet

Helmet lets a directive value be a function that receives the request and response, which fits a per-response nonce without hand-writing the header:

```
import crypto from "node:crypto";
import express from "express";
import helmet from "helmet";

const app = express();

app.use((req, res, next) => {
  res.locals.cspNonce = crypto.randomBytes(16).toString("base64");
  next();
});

app.use(helmet({
  contentSecurityPolicy: {
    useDefaults: false,
    directives: {
      defaultSrc: ['self'],
      scriptSrc: [
        (req, res) => `nonce-${res.locals.cspNonce}`,
        'strict-dynamic',
      ],
      objectSrc: ['none'],
      baseUri: ['none'],
      frameAncestors: ['none'],
      formAction: ['self'],
    },
    reportOnly: process.env.CSP_ENFORCE !== "true",
  },
}));
```

Django

Since Django 6.0 there is built-in CSP support through middleware and declarative settings; in earlier versions, the `django-csp` package covers the same ground. The key point is that the template receives the nonce through the context and prints it on every tag.

```
# settings.py (django-csp 4.x)
MIDDLEWARE = [
  "csp.middleware.CSPMiddleware",
  # ...
]

CONTENT_SECURITY_POLICY = {
  "DIRECTIVES": {
    "default-src": ['self'],
    "script-src": ['self', 'strict-dynamic'],
    "object-src": ['none'],
    "base-uri": ['none'],
    "frame-ancestors": ['none'],
    "form-action": ['self'],
    "include-nonce-in": ["script-src"],
  },
}

# template.html
# <script nonce="{{ request.csp_nonce }}" src="{% static 'app.js' %}"></script>
```

Remember that the exact configuration changes between package versions; confirm the key names against the documentation for the version you have installed.

The edge: Cloudflare Workers and HTMLRewriter

If you cannot touch the application (a legacy CMS, a third-party product), you can add nonces from a Worker that rewrites HTML at the edge. The Worker generates the nonce, sets it on every script tag and in the header. It is a powerful solution, but it takes on the responsibility that no script on the page goes without a nonce, so always start in Report-Only.

```
export default {
  async fetch(request, env) {
    const upstream = await fetch(request);
    const type = upstream.headers.get("content-type") || "";
    if (!type.includes("text/html")) return upstream;

    const bytes = crypto.getRandomValues(new Uint8Array(16));
    const nonce = btoa(String.fromCharCode(...bytes));

    const rewritten = new HTMLRewriter()
      .on("script", { element(el) { el.setAttribute("nonce", nonce); } })
      .transform(upstream);

    const res = new Response(rewritten.body, rewritten);
    res.headers.set(
      "Content-Security-Policy-Report-Only",
      "script-src 'nonce-' + nonce + ' 'strict-dynamic' https: 'unsafe-inline'; object-src 'none';
      base-uri 'none'"
    );
    res.headers.set("Cache-Control", "private, no-cache");
    return res;
  },
};
```

Beware a trap in this technique: if the original application already sets a nonce or a CSP, the two policies intersect, and if the original HTML contains a `<script>` injected by an attacker, the Worker will put a nonce on that too. A nonce rewriter protects against scripts added after the response, not against malicious HTML that already arrives inside it. That makes it a migration bridge, not a destination.

Third parties: analytics, tag managers and widgets without losing control

Third-party tags are the number one cause of violations and of teams abandoning CSP. The strategy that works rests on a simple idea: your policy should trust a single entry point you control, and that entry point decides what else loads.

A nonced loader

Instead of pasting ten vendor snippets into the HTML, you keep one script of your own, with a nonce, that injects the others. With 'strict-dynamic' everything that loader adds inherits trust.

```
<script nonce="{{ nonce }}">
  (function () {
    var tags = [
      "https://www.googletagmanager.com/gtm.js?id=GTM-XXXXXXX",
      "https://widget.example-chat.com/loader.js"
    ];
    tags.forEach(function (src) {
      var s = document.createElement("script");
      s.src = src;
      s.async = true;
      document.head.appendChild(s);
    });
  })();
</script>
```

This moves the governance problem to the right place: whoever edits the tag list goes through code review, not through a marketing panel with no version control. If you use a tag manager, remember that anyone with publish rights on it can run arbitrary JavaScript on your site, with or without CSP. Restrict those permissions as rigorously as production access.

Sandboxed iframes for what you do not control

Widgets you cannot audit (chat, maps, video, embedded payment forms) should not run on your origin. An

iframe with the sandbox attribute isolates them: without the permissions you explicitly grant, they cannot access your cookies, navigate the top window or open popups.

```
<iframe
  src="https://widget.example-chat.com/embed"
  sandbox="allow-scripts allow-forms allow-same-origin"
  referrerpolicy="no-referrer"
  loading="lazy"
  title="Support chat"></iframe>
```

Careful: combining allow-scripts with allow-same-origin in an iframe from the same origin as the page lets the content remove its own sandbox. With a different origin it is safe; with the same origin, do not combine them.

What to do with what does not fit

Some tags simply do not work with a strict policy: they use document.write, eval, inline handlers or inject styles in incompatible ways. You have three honest options: ask the vendor for a compatible version (many already have one), replace it with an alternative that is compatible, or isolate it in a cross-origin iframe. The dishonest option, which you should avoid, is punching a hole in the policy that voids protection for the whole page to satisfy a single vendor.

Reporting in depth: from noise to signal

The value of the Report-Only phase depends entirely on how well you collect and interpret reports. A receiver that only writes log lines produces thousands of useless entries on a site with real traffic; one that aggregates and deduplicates produces a short, actionable list.

The two formats and the two headers

Two mechanisms coexist. The legacy one uses the report-uri directive and sends a body of type application/csp-report containing a csp-report object. The modern one uses the Reporting API: you declare a destination with the Reporting-Endpoints header and reference it from the report-to directive; the browser sends a list of reports of type application/reports+json. To cover both, send both directives at once, and modern browsers use report-to and ignore report-uri.

```
Content-Security-Policy-Report-Only:
  script-src 'nonce-{RANDOM}' 'strict-dynamic' https: 'unsafe-inline';
  object-src 'none';
  base-uri 'none';
  report-uri /csp-reports;
  report-to csp-endpoint
Reporting-Endpoints: csp-endpoint="/csp-reports"
```

Asking for samples with report-sample

By default, a report says "inline" was blocked but not which script it was, which makes it hard to find. Adding 'report-sample' to the directive makes the browser include the first 40 characters of the blocked code in the script-sample field. That is enough to recognize the fragment and know which template produced it, without transmitting the full code.

Aggregate instead of logging

The key is to reduce each report to a stable fingerprint: effective directive, normalized blocked URL, page path (without parameters) and the sample. Counting occurrences per fingerprint turns ten thousand reports into twenty rows. A minimal PostgreSQL schema:

```
CREATE TABLE csp_violations (
  fingerprint text PRIMARY KEY,
  directive text NOT NULL,
  blocked text NOT NULL,
  page_path text NOT NULL,
```

```

sample      text,
first_seen  timestampz NOT NULL DEFAULT now(),
last_seen   timestampz NOT NULL DEFAULT now(),
hits        bigint      NOT NULL DEFAULT 1
);

-- Upsert by fingerprint: each report bumps a counter instead of creating a row
INSERT INTO csp_violations (fingerprint, directive, blocked, page_path, sample)
VALUES ($1, $2, $3, $4, $5)
ON CONFLICT (fingerprint) DO UPDATE
SET hits = csp_violations.hits + 1,
    last_seen = now();

```

```

import hashlib
from urllib.parse import urlsplit

IGNORED_SCHEMES = ("chrome-extension", "moz-extension", "safari-extension", "safari-web-extension")

def normalize_blocked(value: str) -> str:
    """Reduce a blocked URL to origin + path; leave 'inline' and 'eval' as they are."""
    if not value or value in ("inline", "eval", "data", "blob", "self"):
        return value or "unknown"
    parts = urlsplit(value)
    if parts.scheme in IGNORED_SCHEMES:
        return "ignored"
    return parts.scheme + "://" + parts.netloc + parts.path

def fingerprint(directive: str, blocked: str, page_path: str, sample: str) -> str:
    raw = "|".join([directive, blocked, page_path, sample[:40]])
    return hashlib.sha256(raw.encode()).hexdigest()[:32]

```

With that table, the question you ask every morning during the rollout is a single query: which new fingerprints appeared in the last 24 hours and which have the most impact.

```

SELECT directive, blocked, page_path, sample, hits, last_seen
FROM csp_violations
WHERE first_seen > now() - interval '24 hours'
    AND blocked <> 'ignored'
ORDER BY hits DESC
LIMIT 25;

```

Protect the receiver

The reporting endpoint is a public URL that accepts unauthenticated POSTs, so it is an obvious abuse target. Cap the body size, apply per-IP rate limits, validate the accepted content types, never echo report content back in a response, and treat every field as untrusted input before showing it in a dashboard (a report can contain malicious HTML aimed at whoever reads it). One more detail: browsers send reports without credentials, so the endpoint must not depend on session cookies.

Trusted Types in depth: from theory to rollout

The previous section introduced Trusted Types as an idea. Here are the details you need to roll it out without your application breaking. Trusted Types acts on injection sinks: the DOM APIs that turn a string into executable code or markup. With the directive on, those sinks stop accepting strings and demand typed objects (TrustedHTML, TrustedScript or TrustedScriptURL) that only a registered policy can produce.

Which sinks it covers

- TrustedHTML: innerHTML, outerHTML, insertAdjacentHTML, document.write, DOMParser.parseFromString and an iframe's srcdoc attribute.
- TrustedScript: eval, new Function and the text or textContent property of a script element.
- TrustedScriptURL: a script's src attribute, importScripts in workers and the construction of Worker or SharedWorker.

It is worth noting what it does not cover: `textContent` on a normal node, `setAttribute` with harmless attributes or `createElement` are safe by nature and need no policy. The directive only hardens what can actually run code.

A real policy, not a free pass

The most common mistake when adopting Trusted Types is creating a policy whose `createHTML` returns the string untouched. That satisfies the browser formally but protects nothing: you have just moved the vulnerability under a new name. A useful policy sanitizes for real, or rejects.

```
// trusted-types.ts: the only file that creates policies
import DOMPurify from "dompurify";

declare global {
  interface Window { trustedTypes?: TrustedTypePolicyFactory }
}

export const htmlPolicy = window.trustedTypes?.createPolicy("app-html", {
  createHTML: (dirty: string) => {
    DOMPurify.sanitize(dirty, { RETURN_TRUSTED_TYPE: false }),
  });

export const scriptUrlPolicy = window.trustedTypes?.createPolicy("app-script-url", {
  createScriptURL: (url: string) => {
    const parsed = new URL(url, location.origin);
    // Only origins you control; anything else throws
    if (parsed.origin !== location.origin) {
      throw new TypeError("Script origin not allowed: " + parsed.origin);
    }
    return parsed.toString();
  },
});

export function setHtml(el: Element, dirty: string): void {
  if (htmlPolicy) el.innerHTML = htmlPolicy.createHTML(dirty) as unknown as string;
  else el.textContent = dirty; // no support: plain text, never HTML
}
```

Notice the asymmetry: the HTML policy sanitizes because HTML is content that is sometimes legitimate to display; the script URL policy validates and throws, because a script URL has no possible "clean" version. And the unsupported path degrades to text, not to unsanitized HTML.

The trusted-types directive and the default policy

The `trusted-types` directive is the list of allowed policy names. If a script tries to create a policy under another name, the browser blocks it. That is what stops an attacker from registering their own permissive policy. With 'allow-duplicates' you permit reusing names (useful with micro-frontends loaded several times), but avoid it if you can.

```
Content-Security-Policy:
  require-trusted-types-for 'script';
  trusted-types app-html app-script-url dompurify default
```

The policy named `default` is special: the browser calls it automatically when a sink receives a string. It serves as a temporary safety net during migration, because it lets old code keep working while you log every place that falls into it. The golden rule is to treat it as a discovery tool, not a destination: when the usage counter drops to zero, remove it.

```
// Migration only: log each use and sanitize
window.trustedTypes?.createPolicy("default", {
  createHTML: (s) => {
    console.warn("[trusted-types] HTML sink received a string", new Error().stack);
    navigator.sendBeacon("/tt-reports", JSON.stringify({ sink: "html", stack: new Error().stack }));
    return DOMPurify.sanitize(s);
  },
```

```
createScript: () => { throw new TypeError("eval blocked"); },
createScriptURL: () => { throw new TypeError("dynamic script blocked"); },
});
```

A four-step migration plan

1. Report-Only first. Deploy Content-Security-Policy-Report-Only: require-trusted-types-for 'script'. Each violation points to a sink receiving a string. Aggregate them with the same fingerprint technique as the script reports.
2. Remove the avoidable sinks. Most innerHTML calls with plain text are solved with textContent, and template ones with node construction or a templating engine that escapes by default. It is the cheapest step and the one that cuts volume the most.
3. Centralize the unavoidable. For HTML you truly must display (formatted comments, CMS content), route through a single function that uses the sanitizing policy. One point is auditable; twenty are not.
4. Enforce and watch. Switch to the blocking header, keep reporting on, and add a lint rule to CI that forbids assigning to innerHTML outside that module.

Frameworks and libraries

React escapes by default and exposes only one explicit sink, dangerouslySetInnerHTML, which you should route through the policy. Angular has native Trusted Types integration and its own internal policy. Lit and other tagged-template libraries create their own policies, so you will need to add their names to the directive. For dependencies that write to innerHTML internally, check whether they publish a compatible version; if not, the default policy with sanitizing covers you in the meantime.

The state of support

Trusted Types was born in Chromium and for years was exclusive to that engine. Support in other browsers has been arriving in stages; check the current state on MDN or the compatibility tables before deciding how much you rely on the layer. The design consequence is the same whatever the table says: treat Trusted Types as hardening on top of already-correct code, and always keep the safe degradation path (plain text) for browsers where the API does not exist.

What CSP does not do: real bypasses and how to think about them

A strict CSP closes most routes of classic XSS, but it is not a magic barrier. Knowing the attacks that survive helps you decide what to complement it with.

Dangling markup

If the attacker can inject an unclosed tag, such as `<img src='https://evil.example/?'`, the browser swallows the following HTML as part of the URL, including CSRF tokens or page data, and sends it to the attacker without running a single script. A CSP focused on script-src does not stop it. A restrictive img-src, form-action and, above all, correct output escaping mitigate it. It is the perfect example of why CSP is defense in depth and not a substitute for encoding.

Script gadgets and inherited trust

With 'strict-dynamic' trust propagates: if a legitimate script in your application reads a string from the DOM and uses it to create a new script, the attacker who controls that string controls the script. These are called gadgets. The dangerous patterns build script paths from data- attributes, URL parameters or DOM content. The defense is to review your own dynamic-loading code and validate against a closed list, not to trust whatever is in the document.

Base tag injection and scripts that reuse nonces

Without base-uri, an injected `<base href="https://evil.example/">` changes the resolution of every relative URL, and a script with a valid nonce loaded relatively starts downloading from the attacker. That is why base-uri 'none' is in every strict policy. Similarly, certain attacks try to read the nonce from the DOM through CSS selectors or browser-cache attacks in order to reuse it; the best defense is that the nonce does not leak (never dump it in URLs, shared-cache headers or logs) and that its lifetime is a single response.

JSONP, old Angular and reflective endpoints

These are the classic bypasses of host allowlists, and exactly why a nonce with 'strict-dynamic' replaces them. If during a migration you keep some host allowlist for a while, run your policy through an evaluator such as Google's CSP Evaluator: it automatically flags hosts known to serve exploitable endpoints.

DOM clobbering and other primitives

An attacker who can only inject inert HTML elements (for example, a tag with an id or name attribute) can overwrite global variables and alter the flow of your JavaScript without running any code of their own. CSP does not see this as a violation. The remedy is the usual one: sanitize with a strict allowlist (DOMPurify, for example, strips dangerous id and name values in its recommended configuration) and do not rely on global variables that can be overwritten.

How to evaluate your own policy

Two free tools cover almost everything. Google's CSP Evaluator analyzes a policy and points out weak directives, explaining each bypass. And an audit with Lighthouse, or a test like the Playwright one we saw earlier, verifies that the deployed policy is the expected one. Use them as part of reviewing every change, not once at the start.

Sibling headers: what completes CSP

CSP rarely travels alone. These headers cover adjacent risks and are usually deployed in the same project:

- X-Content-Type-Options: nosniff stops the browser from guessing MIME types and running a file served as text as a script. It complements script-src, since it closes the route of uploading a "text file" that gets interpreted as JavaScript.
- Referrer-Policy controls how much of the origin URL is sent to other sites. strict-origin-when-cross-origin is a reasonable default.
- Permissions-Policy disables browser capabilities (camera, microphone, geolocation) that your app does not use, so neither an injected script nor an iframe can request them.
- Cross-Origin-Opener-Policy and Cross-Origin-Embedder-Policy isolate your document from other windows and enable cross-origin isolation, relevant against side-channel attacks.
- Strict-Transport-Security forces HTTPS, and makes sense alongside upgrade-insecure-requests.

```
X-Content-Type-Options: nosniff
Referrer-Policy: strict-origin-when-cross-origin
Permissions-Policy: camera=(), microphone=(), geolocation=(), payment=()
Cross-Origin-Opener-Policy: same-origin
Strict-Transport-Security: max-age=63072000; includeSubDomains
```

Case study: migrating a 40-page application to a strict CSP in five weeks

An illustrative scenario (the figures are a plausible example, not data from a specific client) helps show how the pieces fit together. Imagine a web application with a user dashboard, a blog and a checkout flow, about forty server templates, a tag manager with seven vendors and a current policy based on a twenty-two-entry host allowlist.

Week 1: inventory and Report-Only. The team deploys the strict policy in Report-Only mode with the aggregating receiver. On day one thousands of reports arrive, but the fingerprint table reduces them to about sixty distinct rows. A third is browser-extension noise, which is filtered out immediately. About forty real fingerprints remain, nearly half of them onclick handlers in three shared templates.

Week 2: refactor the cheap stuff. Inline handlers are replaced with delegated listeners in a single file. Two templates that used eval to evaluate configuration expressions are rewritten with JSON. The fingerprint volume drops to a dozen. What remains are third-party tags.

Week 3: third parties. Each vendor's pasted snippet is replaced with the single nonced loader. Two vendors use document.write: one publishes a compatible version, the other is isolated in a sandboxed iframe. Publish rights on the tag manager are reduced to two people with double review.

Week 4: tests and rehearsal. The Playwright test is added to CI for the ten critical routes, including the full checkout flow. Enforcement is switched on in a test environment for two days while the product team walks through the app. Two new violations appear on rarely visited routes and are fixed.

Week 5: percentage rollout. Enforcement is enabled behind a feature switch: first 5% of traffic, then 25%, 50% and 100%, with a one-day pause at each step while watching the JavaScript error rate and checkout conversions. At 100%, the reporting stream stays on forever.

What matters in this journey is not the numbers but the order: measure first, remove what is avoidable, centralize what is unavoidable, and roll out gradually with a way back. None of the steps requires special security expertise; it requires ordinary engineering discipline.

The percentage rollout switch

Being able to move from Report-Only to enforcement for a percentage of users is what makes the rollout reversible in seconds. A function that picks the header based on a stable visitor identifier is enough:

```
import hashlib
import os

ENFORCE_PERCENT = int(os.environ.get("CSP_ENFORCE_PERCENT", "0"))

def csp_header_name(visitor_id: str) -> str:
    """Return the header to use for this visitor.
    The hash keeps the same visitor in the same group every time."""
    bucket = int(hashlib.sha256(visitor_id.encode()).hexdigest(), 16) % 100
    if bucket < ENFORCE_PERCENT:
        return "Content-Security-Policy"
    return "Content-Security-Policy-Report-Only"
```

With the environment variable at zero, nobody is blocked; at one hundred, everyone is. Rolling back a problematic deployment means changing a number and restarting, without touching code or waiting for a new build.

Runbook: what to do when a CSP breaks production

Sooner or later a policy breaks something. Having the procedure written in advance prevents rushed decisions, such as turning CSP off entirely.

1. Confirm it is CSP. Open the browser console: violations show up as errors starting with "Refused to execute" or "Refused to load". If there are none, the problem is something else.
2. Reduce the blast radius. Lower the enforcement percentage to zero with the switch. Users get the

service back within seconds and the policy keeps reporting in Report-Only.

3. Identify the fingerprint. Query the violations table sorted by last seen. The directive and blocked URL almost always explain the cause: a new marketing tag, a CDN change, a template with an inline handler.

4. Fix at the source. Add a nonce, move the code to a file or update the dependency. Resist the temptation to add 'unsafe-inline' or a wildcard: they solve the symptom and void the protection.

5. Add a test. Turn the case into a CI test so it does not go unnoticed again.

6. Document it. One line in the policy changelog with the date, the cause and the fix. In a year nobody will remember why an exception exists.

Metrics that matter

A CSP deployed without metrics rots unnoticed. Four numbers are enough to know whether the policy is still healthy:

- New fingerprints per day. In steady state it should be close to zero. A spike means a deployment added something without going through the nonced loader.
- Violations per thousand page views. Normalize by traffic so you can compare different weeks.
- Percentage of HTML responses carrying the header. A drop signals a new route, a proxy or a cache stripping the header.
- Age of the oldest exception. Every temporary exception (an entry in connect-src, an 'unsafe-hashes') should have a date and an owner. An exception without an expiry date is permanent.

A 30-day adoption plan

For a team starting from zero, this calendar is realistic and leaves a safety margin:

- Days 1 to 3: sibling headers (nosniff, Referrer-Policy, frame-ancestors) and the report receiver. No risk, immediate value.
- Days 4 to 14: strict policy in Report-Only, fingerprint aggregation and the first batch of refactors.
- Days 15 to 21: nonced third-party loader, sandboxed iframes for what does not fit, and the CI test.
- Days 22 to 28: gradual percentage enforcement while watching the metrics.
- Day 29 onward: Trusted Types in Report-Only and a monthly review of new fingerprints.

Frequently asked questions

Can I use a strict CSP with a single-page app served as static HTML? Yes, with hashes for inline scripts and SRI for external ones, or by putting a Worker or lightweight server in front that injects nonces. What you cannot do is a fixed nonce in cached HTML.

What about inline styles? They are far less dangerous than scripts, and keeping style-src 'unsafe-inline' at the start is reasonable. If you want to harden them, use nonces or hashes for styles too, but do it after the scripts are solved.

Does CSP replace the WAF or sanitization? No. It is one more layer. Output encoding prevents the vulnerability; CSP limits the damage if it fails; the WAF filters known traffic; and the reports tell you where to learn.

How long should I leave Report-Only on? Long enough to cover a full cycle of your traffic: at least one week, and ideally two if there are monthly tasks (billing, reports) that use rarely visited routes.

Can reports leak personal data? Yes: the document URL and the blocked URL can contain identifiers in the path or in parameters. Normalize or trim before storing, and set a short retention.

Is it safe to keep https: in script-src? Yes, as long as there is a nonce or hash and 'strict-dynamic': a CSP3 browser ignores it. It only acts as a fallback for old browsers.

Glossary

- CSP (Content Security Policy): a header that restricts which resources a page may load and run.
- Nonce: a random value, different on every response, that marks the allowed scripts.
- Hash: a cryptographic fingerprint of an inline script's content that authorizes it without a nonce.
- strict-dynamic: a keyword that propagates trust from an authorized script to the scripts it loads.
- Report-Only: a mode in which the browser reports violations without blocking anything.
- Sink: a DOM API that turns a string into executable code or markup.
- Trusted Types: a mechanism that demands typed objects, created by policies, to use sinks.
- SRI (Subresource Integrity): an attribute that verifies an external resource's hash before running it.
- Gadget: a piece of legitimate code an attacker can leverage to run their payload.
- Dangling markup: injecting an unclosed tag to exfiltrate content without running scripts.

Testing beyond the home page: a suite that walks the whole application

The Playwright test at the start checks a single page. A strict policy breaks on the routes nobody looks at: the password-reset page, the admin panel, the 500 error page. A suite that walks a list of critical routes and fails on any violation catches those cases before your users do.

```
// csp-routes.spec.ts
import { test, expect } from "@playwright/test";

const ROUTES = ["/", "/login", "/pricing", "/blog", "/checkout", "/account",
"/404-that-does-not-exist"];

for (const route of ROUTES) {
  test("no CSP violations on " + route, async ({ page }) => {
    const violations: string[] = [];

    page.on("console", (msg) => {
      const text = msg.text();
      if (text.includes("Content Security Policy") || text.includes("Content-Security-Policy")) {
        violations.push(text);
      }
    });

    await page.addInitScript(() => {
      document.addEventListener("securitypolicyviolation", (e) => {
        (window as any).__csp = ((window as any).__csp ?? []).concat(
          e.effectiveDirective + " <- " + (e.blockedURI || "inline") + " @ " + e.sourceFile + ":" +
            e.lineNumber
        );
      });
    });

    await page.goto(route, { waitUntil: "networkidle" });
    // Minimal interactions that trigger deferred code
    await page.mouse.move(100, 100);
    await page.keyboard.press("Tab");

    const fromEvents = await page.evaluate(() => (window as any).__csp ?? []);
    expect([...violations, ...fromEvents]).toEqual([]);
  });
}
```

```
});  
}
```

Two useful nuances. The securitypolicyviolation event includes sourceFile and lineNumber, which point at the offending code, something the report sent to the server does not always carry. And the minimal interactions (moving the mouse, pressing a key) wake up lazily loaded scripts that would not show up in a passive navigation.

For email templates and generated documents, which do not go through the browser, add a static check: a script that searches the rendered HTML for on*= attributes and <script> tags without nonce. It is a simple regular expression that blocks 90% of regressions at code review, before you even launch a browser.

Edge cases: service workers, WebAssembly and dynamic styles

Three situations puzzle almost every team the first time they meet them under a strict policy.

Service workers. A service worker has its own policy, which is the header of the response that serves the worker file, not the page that registers it. If you serve /sw.js without CSP, the worker runs unrestricted even though your page is locked down. Apply the headers to worker scripts too, and serve that file with Cache-Control: no-cache so updates arrive quickly.

WebAssembly. Compiling and running Wasm modules requires the 'wasm-unsafe-eval' keyword in script-src. It is much narrower than 'unsafe-eval': it does not enable JavaScript eval, only Wasm compilation. If your app uses a Wasm module (an image editor, an in-browser database engine), add that keyword and nothing more.

Styles changed from JavaScript. Assigning element.style.color = "red" is always legal, because it manipulates the style object and does not parse CSS text. What gets blocked is element.setAttribute("style", "...") or inserting a <style> tag without a nonce when style-src is strict. Almost all modern component libraries adapt, but test them with the policy on before tightening styles.

Takeaways

Skip host allowlists and build on nonces plus 'strict-dynamic'. Ship in Report-Only mode, learn from the reports, then enforce. Keep HTML out of shared caches so nonces stay unique, and put a test in CI so the policy cannot rot. Done well, CSP turns a single missed escape from a breach into a line in your violation log.