Important: Always run these tests in-scope. Do not attempt network interception (MITM/sslstrip) against systems you do not own or have written permission to test.

# 1) X-Content-Type-Options (nosniff)

**Purpose:** prevent MIME sniffing so browsers don’t execute resources as a different MIME type.

### A — Presence check (curl)

curl -I -s https://example.com/path | sed -n '1,120p'

Look for:

X-Content-Type-Options: nosniff

### B — Functional browser test (local proof-of-concept)

Create a small file on the target (or point to an existing static resource you control) that is served with Content-Type: text/plain but contains HTML/JS:

<!-- file: /test-sniff.txt -->

<script>window.sniffed = 'executed'; alert('sniffed');</script>

From your browser, open that file URL. If server sets X-Content-Type-Options: nosniff, the browser should treat it as plain text and **not execute** the script. Without the header, some browsers may execute it (older behaviors or particular contexts).

### Evidence example (capture response headers)

GET /test-sniff.txt HTTP/1.1

Host: example.com

Response:

HTTP/1.1 200 OK

Content-Type: text/plain

X-Content-Type-Options: nosniff

Content-Length: 123

**Expected behaviour:** script contents are not executed. Check browser console for lack of sniffed global or runtime alert.

### Troubleshooting

If header present but browser still executes, ensure the response Content-Type is actually text/plain and not being changed by a proxy/CDN.

Check static/CDN config — these often omit security headers.

# 2) Strict-Transport-Security (HSTS)

**Purpose:** force browsers to use HTTPS for future requests to the domain (protects against downgrade/ssl-stripping once set).

### A — Presence check

curl -I -s https://example.com/ | grep -i strict-transport-security || echo "HSTS missing"

Look for:

Strict-Transport-Security: max-age=31536000; includeSubDomains; preload

### B — Functional checks

**Header present and correct value** — max-age should be sufficiently large (e.g., ≥ 31536000 for production), and consider includeSubDomains if appropriate.

**Redirect test (no MITM):** ensure http://example.com redirects to https://example.com:

curl -I -s http://example.com/ | sed -n '1,120p'

You should see a 301/302 → Location: https://example.com/. HSTS itself is a browser caching mechanism; its effect is observed on subsequent requests by the browser. To verify browser behavior:

Visit https://example.com to get the HSTS header, then try to load http://example.com in the same browser; it should be upgraded internally by the browser to https://example.com (no network redirect visible). Use devtools Network tab to confirm.

### Evidence example:

GET / HTTP/1.1

Host: example.com

Response:

HTTP/1.1 200 OK

Strict-Transport-Security: max-age=31536000; includeSubDomains; preload

### Troubleshooting

Don’t enable preload unless you understand the global consequences (cannot be easily removed).

If header missing on subdomain, browser won’t apply HSTS to them unless includeSubDomains used.

# 3) X-Frame-Options / frame-ancestors

**Purpose:** prevent clickjacking by blocking framing of pages.

Note: X-Frame-Options is legacy; Content-Security-Policy: frame-ancestors is preferred for finer control.

### A — Presence check

curl -I -s https://example.com/ | egrep -i 'X-Frame-Options|Content-Security-Policy' || echo "No frame protection header"

Expect one of:

X-Frame-Options: DENY

X-Frame-Options: SAMEORIGIN

Content-Security-Policy: frame-ancestors 'none' # or a list of allowed origins

### B — Functional test (local framing page)

Create a local HTML file:

<!-- clickjack-test.html (host locally) -->

<!doctype html>

<html>

<body>

<h1>Clickjacking test</h1>

<iframe src="https://example.com/login" width="800" height="600"></iframe>

</body>

</html>

Open clickjack-test.html in a browser. If page is blocked the iframe will be blank or the browser console will say blocked due to X-Frame-Options or frame-ancestors. If it renders, framing is allowed — risky for sensitive pages.

### Evidence example (response header):

HTTP/1.1 200 OK

X-Frame-Options: DENY

**Expected result:** iframe content is blocked; console shows frame blocked message.

### Troubleshooting

If X-Frame-Options present but page still frames, check for conflicting CSP frame-ancestors or multiple headers; browsers might prioritize CSP rules.

# 4) Content-Security-Policy (CSP)

**Purpose:** restrict sources for scripts, styles, frames, etc. Powerful mitigation against XSS and data exfiltration.

### A — Presence check

curl -I -s https://example.com/ | grep -i 'Content-Security-Policy' || echo "CSP missing"

Example header:

Content-Security-Policy: default-src 'self'; script-src 'self' https://cdn.example.com; object-src 'none'; report-uri https://collector.example.com/csp-report

### B — Functional tests

#### 1) Block external script test

Create two pages:

A test page hosted at https://example.com/test-csp.html that includes a script from https://attacker.example/js/evil.js.

If CSP allows only 'self', the script should be blocked and the browser console will show the CSP violation.

Local test HTML (if you can host a test page on the target):

<!-- test-csp.html hosted on the target -->

<!doctype html>

<html>

<head>

<script src="https://attacker.example/evil.js"></script>

</head>

<body>csp test</body>

</html>

Open in a browser — check console for Refused to load the script CSP message.

#### 2) Inline script blocking (unsafe-inline)

If the CSP contains 'unsafe-inline', inline <script> blocks will run. To test, place an inline script:

<script>console.log('inline executed')</script>

If CSP forbids inline scripts, this will be blocked.

#### 3) CSP report-only testing

If server uses Content-Security-Policy-Report-Only, it will not block but will emit violation reports. Check your report-uri endpoint or browser console.

### Evidence example (response):

HTTP/1.1 200 OK

Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none'; frame-ancestors 'none'; report-uri https://collector.example.com/csp-report

**Expected result:** third-party scripts and inline scripts are blocked; check console messages like:

Refused to load the script 'https://attacker.example/evil.js' because it violates the following Content Security Policy directive: "script-src 'self'".

### Troubleshooting

CSP policies can be long and complex. Use report-uri/report-to to collect violations before enforcement.

Some external services (analytics, CDNs) may require allowlisting.

# 5) Referrer-Policy

**Purpose:** control what is sent in the Referer header to other sites (prevents leaking full URLs or tokens).

### A — Presence check

curl -I -s https://example.com/ | grep -i referrer-policy || echo "Referrer-Policy missing"

Common secure values:

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

Referrer-Policy: no-referrer

### B — Functional test (server to capture Referer)

You need a receiver server to capture incoming requests (or use a request bin service you control).

Host a test page on the target that links to your capture endpoint:

<!-- hosted on https://example.com/ref-test.html -->

<a href="https://collector.example/capture">send ref</a>

Click the link from https://example.com/ref-test.html. On your collector server check the Referer header that arrived.

With Referrer-Policy: no-referrer, no Referer header should be present.

With strict-origin-when-cross-origin, cross-origin requests will send only origin (e.g., https://example.com) not full path.

Alternatively, test with JavaScript fetch from a page on the site that hits your collector and inspect network requests.

### Evidence example (response headers):

HTTP/1.1 200 OK

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

Collector request received:

GET /capture HTTP/1.1

Host: collector.example

Referer: https://example.com/ # only origin, path omitted

**Expected result:** Referer content matches the policy.

### Troubleshooting

Some browsers may implement older/referrer behaviors; test across browsers.

Third-party libraries may break if they relied on full referer paths.

# 6) Permissions-Policy (Feature-Policy)

**Purpose:** control which features (geolocation, camera, microphone, payment, fullscreen, etc.) are allowed in the document or in framed contexts.

### A — Presence check

curl -I -s https://example.com/ | egrep -i 'permissions-policy|feature-policy' || echo "Permissions-Policy missing"

Example:

Permissions-Policy: geolocation=(), camera=(), microphone=()

### B — Functional tests (JS attempts)

Create a small test page (host on the target or locally framed to test frame-scoped behavior):

<!-- permissions-test.html -->

<!doctype html>

<html>

<body>

<h1>Permissions test</h1>

<button id="geo">Try geolocation</button>

<button id="cam">Try camera</button>

<script>

document.getElementById('geo').onclick = () => {

navigator.geolocation.getCurrentPosition(

p => console.log('geo success', p),

e => console.error('geo error', e)

);

};

document.getElementById('cam').onclick = async () => {

try {

const s = await navigator.mediaDevices.getUserMedia({video:true});

console.log('camera success', s);

s.getTracks().forEach(t => t.stop());

} catch (err) {

console.error('camera error', err);

}

};

</script>

</body>

</html>

Open the page and click the buttons:

If Permissions-Policy denies the feature (geolocation=()), getCurrentPosition should fail or be blocked without prompting.

For camera/microphone, getUserMedia should throw or be blocked.

### Evidence example (response headers)

HTTP/1.1 200 OK

Permissions-Policy: geolocation=(), camera=(), microphone=()

Browser console expected messages:

camera error DOMException: Permission denied

or feature-specific blocking messages.

### Troubleshooting

Some browsers vary in error messages; check browser compatibility for the exact directive names.

If permission prompt still appears, verify header syntax and that no iframe or document policy overrides apply.

# Automation snippets

### Quick single-endpoint checker (bash)

URL="https://example.com"

curl -I -s "$URL" | egrep -i 'X-Content-Type-Options|Strict-Transport-Security|X-Frame-Options|Content-Security-Policy|Referrer-Policy|Permissions-Policy|Feature-Policy' || echo "One or more headers missing"

### Bulk checker (reuse earlier script)

Use the check\_headers.sh I provided earlier to scan many endpoints and exported CSV results for review.

# Final notes & expected workflow

Run the **curl presence checks** for all routes (root, login, dashboard, API endpoints, static assets/CDN).

For sensitive endpoints (login, payment, profile upload) perform the **functional tests** above.

Collect evidence: for each tested URL include the request and full response headers (curl -i or -I) and browser console screenshots/logs for functional failures.

If a header is missing on static assets or CDN, fix configuration at the source (web server, CDN rules).

If a header is present but ineffective, confirm it is not being overwritten downstream (reverse proxy, load balancer).

If you want, I can:

provide a ready-to-run test kit (zip) containing the local HTML test pages and a small Node.js collector to capture Referer/CSP reports, or

produce a per-route test plan (a checklist you can paste into your issue tracker) that lists which tests to run per endpoint.

Tell me which of those you want and I’ll produce it now (all runnable locally).