Web Application Security Course

Flavio Copes

Updated August 3, 2026

Contents

Protect browser-based applications from XSS, injection, CSRF, unsafe uploads, SSRF, broken access control, and insecure deployment.

What you'll learn: Threat-model, harden, test, and operate a small web application across its browser, server, storage, and deployment boundaries.

Browser security boundaries

Reason about origins, cookies, CORS, CSP, storage, and trust between sites.

Understand the origin boundary

Use scheme, host, and port to reason about which documents and requests can share browser authority.

The browser uses an **origin** as a major security boundary. An origin is the exact triple of scheme, host, and port. All three must match for two documents to share full browser authority.

That means `https://app.flaviocopes.com` and `http://app.flaviocopes.com` are different origins, because the scheme differs. `https://app.flaviocopes.com` and `https://app.flaviocopes.com:8443` are different origins, because the port differs. The host is compared exactly, so `flaviocopes.com` and `www.flaviocopes.com` do not match either.

Sending is not the same as reading

The **same-origin policy** restricts one origin from *reading* another origin's data. It does not stop every cross-origin request from being *sent*. This distinction is where most confusion starts.

```
// Runs on https://evil.example
// The request IS sent, and cookies for the target may ride along
fetch('https://api.flaviocopes.com/account', { credentials: 'include' })
  .then(res => res.json()) // ← the browser blocks reading this response
  .catch(err => console.log('read blocked:', err))
```

The `fetch()` above leaves the browser and hits your server. The browser only refuses to hand the *response* back to the calling script. An HTML form is even more permissive: it can POST to any origin and the request arrives at your server normally.

```
<!-- A plain form posts cross-origin with no restriction -->
<form action="https://api.flaviocopes.com/account/email" method="post">
  <input name="email" value="attacker@evil.example">
</form>
```

`https://app.flaviocopes.com` and `https://api.flaviocopes.com` belong to the same company but not the same origin. A browser may send a cross-origin form while refusing JavaScript access to its response.

Saying ”the browser blocks cross-origin requests” hides this important difference. Design separately for sending, reading, cookies, navigation, and other browser-controlled behavior.

Classify five URLs by scheme, host, and port, then test one cross-origin `fetch()` and one HTML form submission. Record whether each request was sent and whether the initiating page could read the response.

Protect session cookies

Use restrictive cookie attributes and understand why a cookie can be sent even when browser JavaScript cannot read it.

A session cookie carries authority. Whoever holds it is treated as the logged-in user until it expires. So the attributes you set on it decide how far that authority can leak.

`HttpOnly` keeps the cookie away from `document.cookie`, which blocks a script from reading it. But `HttpOnly` does not stop the browser from *attaching* the cookie to requests. Reading and sending are separate concerns, and each attribute controls a different one.

Set the attributes deliberately

Here is a session cookie set from an Express-style handler, with each attribute chosen on purpose.

```
res.cookie('session', sessionId, {
  httpOnly: true,          // no script access
  secure: true,            // only sent over HTTPS
  sameSite: 'lax',         // not sent on cross-site subrequests
  path: '/',               // scope to what the app needs
  maxAge: 1000 * 60 * 60,  // 1 hour
})
```

Notice there is no `Domain` attribute. Leaving it out scopes the cookie to the exact host that set it. Adding `Domain=.flaviocopes.com` would widen it to every subdomain, which is authority you rarely want to hand out.

When the rules fit, a `__Host-` prefix is the strongest baseline the browser enforces for you.

```
Set-Cookie: __Host-session=abc123; Secure; HttpOnly; SameSite=Lax; Path=/
```

The `__Host-` prefix *requires* `Secure`, `Path=/`, and no `Domain`, so the browser rejects the cookie if any of those is missing. That turns a policy into something the browser checks.

A session cookie is `HttpOnly` but lacks `Secure`, so a mistaken HTTP link can expose it before the redirect. Another cookie uses `Domain=.flaviocopes.com`, giving every subdomain unnecessary scope.

A strict `SameSite` value can break a legitimate external login return. Choose the narrowest setting that still supports the real flow, then test that flow.

Capture the session `Set-Cookie` header and justify `Secure`, `HttpOnly`, `SameSite`, `Path`, `Domain`, and expiry. Test an HTTP request, a cross-site form, and the real sign-in return path.

Configure CORS narrowly

Allow only the browser origins, methods, headers, and credential behavior an API actually needs.

CORS tells browsers which other origins may *read* a response. That is all it does. It is not authentication, and it does nothing to protect non-browser clients like `curl` or a mobile app, which ignore these headers entirely.

The dangerous pattern is reflecting whatever `Origin` the request carried and allowing credentials. That combination lets any site read a signed-in user's data.

```
// Vulnerable: echoes the caller's origin and allows cookies
app.use((req, res, next) => {
  res.set('Access-Control-Allow-Origin', req.headers.origin)
  res.set('Access-Control-Allow-Credentials', 'true')
  next()
})
```

Allow one origin at a time

The safe version keeps an explicit allowlist and answers with a single, exact origin. When you vary the header by caller, you must also send `Vary: Origin` so caches do not serve one origin's response to another.

```
const allowed = new Set([
  'https://app.flaviocopes.com',
  'http://localhost:3000',
])

app.use((req, res, next) => {
  const origin = req.headers.origin
  if (allowed.has(origin)) {
    res.set('Access-Control-Allow-Origin', origin)
    res.set('Access-Control-Allow-Credentials', 'true')
    res.set('Vary', 'Origin')
  }
  next()
})
```

Never return `Access-Control-Allow-Origin: *` together with credentials. The browser refuses that combination, and reaching for it is usually a sign the allowlist is missing.

An API reflects any `Origin` and allows credentials. A page on an attacker-controlled origin can read the signed-in user's private API response.

A broad wildcard feels convenient during development, but it can escape into production unnoticed. Keep local origins explicit and separate from the production allowlist.

Send preflight and credentialed requests from the production origin, the local origin, and an unlisted origin. Save the response headers and prove the unlisted page cannot read the protected response.

Add a Content Security Policy

Use CSP to restrict script and resource execution while keeping output handling and application design as the primary XSS defenses.

A **Content Security Policy** limits what a page is allowed to load and execute. It is a strong second layer. Treat it as defense in depth behind correct output handling, not as permission to render unsafe HTML.

The mistake is to enable enforcement on day one and break the site. Start in report-only mode instead, so violations are reported but nothing is blocked while you learn what the page really needs.

```
Content-Security-Policy-Report-Only:
  default-src 'self'; script-src 'self'; report-uri /csp-report
```

Point `report-uri` at an endpoint and read what arrives. Each report names the blocked resource and the directive that would have stopped it, so you build the real policy from evidence rather than guessing.

Move toward nonces, not exceptions

Once you understand the page, switch to enforcement and let inline scripts run through a per-response **nonce** rather than opening `unsafe-inline`.

```
const nonce = crypto.randomBytes(16).toString('base64')
res.set('Content-Security-Policy',
  `default-src 'self'; script-src 'self' 'nonce-${nonce}'`)
```

```
<script nonce="RANDOM_PER_RESPONSE">/* only this inline script runs */</script>
```

A new analytics script requires `unsafe-inline`, so the team adds it to `script-src`. That exception also lets an injected inline script run across the site.

A strict policy can break checkout or sign-in if required sources are missed. Start in report-only mode, then replace broad exceptions with nonces, hashes, or removed scripts.

Deploy a report-only policy and exercise the home, sign-in, and checkout paths. Record required violations, remove one unnecessary source, then test that an injected inline script is blocked under enforcement.

Check your understanding: browser security boundaries

Check origins, cookies, CORS, CSP, browser storage, and trust between sites.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Injection and output

Prevent XSS, SQL injection, command injection, and path traversal.

Prevent cross-site scripting

Keep untrusted data as text, use framework escaping, and avoid browser sinks that interpret attacker-controlled strings as code or markup.

XSS happens when the browser interprets untrusted data as active page content instead of inert text. An attacker's string stops being a name and becomes running code. The safest default is to render text as text.

The two lines below look almost identical, but they behave completely differently.

```
message.textContent = userMessage  // rendered as text, always safe
message.innerHTML = userMessage     // parsed as HTML, a script sink
```

`textContent` writes characters. `innerHTML` parses markup. A **sink** is any place a value turns back into code, and the same variable can be safe in one sink and dangerous in another.

Match the encoding to the destination

Framework interpolation escapes for you in the normal case. Trust it, and avoid the bypasses.

```
<span>{userMessage}</span>          {/* React escapes this automatically */}
<span dangerouslySetInnerHTML={{ __html: userMessage }} /> {/* bypass, do not */}
```

If the product genuinely must accept HTML, run it through a maintained sanitizer with a narrow allowlist rather than writing your own filter.

```
import DOMPurify from 'dompurify'
preview.innerHTML = DOMPurify.sanitize(userMessage) // strips scripts and handlers
```

A profile name containing `<img src=x onerror=alert(1)>` is safe in escaped text. The same value becomes active when a preview feature assigns it to `innerHTML`.

Escaping once at input time is fragile because the value may later enter HTML, an attribute, or a URL. Keep the stored value unchanged and use a safe sink for its final context.

Trace one user-controlled value to every place it renders and record each sink. Test text, markup, attribute-breaking, and `javascript:` payloads, then prove none execute in the browser.

Use parameterized database queries

Keep SQL structure separate from values so user input cannot become executable database syntax.

SQL injection appears when data is concatenated into a query string. Once the value and the query share one channel, a crafted value can rewrite the query's logic. Parameters keep the value in the value channel where it can only ever be data.

Here is the pattern that fails. The search term is glued directly into the SQL text.

```
// Vulnerable: input becomes part of the query structure
const rows = await db.query(
  `SELECT id, title FROM notes WHERE owner_id = ${ownerId} AND title = '${term}'`
)
```

A term of `' OR '1'='1` closes the string early and changes the condition. The database now returns rows the user was never allowed to see.

Bind values, allowlist identifiers

The safe version sends the query text and the values separately. The driver never treats a value as syntax.

```
SELECT id, title
FROM notes
WHERE owner_id = ? AND id = ?
```

```
const rows = await db.query(
  'SELECT id, title FROM notes WHERE owner_id = ? AND id = ?',
  [ownerId, noteId],
)
```

Parameters cover values, not identifiers. A column name for sorting cannot be a bound parameter, so map it through a fixed lookup.

```
const sortColumns = { title: 'title', created: 'created_at' }
const orderBy = sortColumns[req.query.sort] ?? 'created_at' // reject anything else
```

A search query concatenates `owner_id` and the search term into SQL. A crafted quote changes the condition and returns another user's notes.

Parameters protect values, not dynamic table names or sort columns. Keep identifiers in a fixed

mapping instead of passing them through the value API.

Replace one concatenated query with parameters and capture its result for a title containing a quote. Test SQL control characters and an unsupported sort field, then verify no extra rows or schema changes appear.

Avoid command injection

Prefer library APIs, pass fixed executable arguments without a shell, and allowlist the small set of operations a feature supports.

Shells turn strings into programs. A shell reads characters like `;`, `|`, `&&`, and backticks as instructions. Passing user input through a shell gives that input the power to start new commands.

The vulnerable shape passes a whole string to the shell to parse.

```
const { exec } = require('node:child_process')
// Vulnerable: the filename is interpreted by /bin/sh
exec(`convert uploads/${name} output.png`)
```

An image tool runs `convert uploads/${name} output.png` through a shell. A filename containing shell syntax starts a second command with the server's permissions.

Pass arguments, not a command line

Prefer a native library when one exists. If you must spawn a process, choose the executable in code and pass arguments as a separate array. `execFile` does not invoke a shell, so shell metacharacters in an argument are just characters.

```
const { execFile } = require('node:child_process')
// Safe: fixed executable, arguments never parsed by a shell
execFile('convert', ['--', `uploads/${name}`, 'output.png'], (err) => {
  if (err) return handleError(err)
})
```

The `--` marks the end of options, so a filename like `-delete` cannot be read as a flag. Combine this with an allowlist of the operations the feature actually supports.

Hand-written escaping changes across shells and platforms, and one missed character is enough. A library call or direct argument array removes shell parsing from the boundary.

Run the conversion through a no-shell process API with a fixed executable and argument list. Test spaces, quotes, separators, and a leading dash in the filename, then prove only the intended output file changes.

Keep file paths inside a root

Generate storage names, resolve canonical paths, and prevent user-controlled traversal outside the directory an operation owns.

A filename like `../../secrets.env` is data to the user and a path instruction to the filesystem. The `..` segments walk up the directory tree. Treating those two meanings as equivalent is how path traversal starts.

The weak check compares strings before the path is resolved.

```
// Vulnerable: prefix check on unresolved text
const target = path.join('/srv/reports', req.query.file)
if (target.startsWith('/srv/reports')) fs.createReadStream(target)
```

A download route joins `/srv/reports` with `../../.env`. A prefix check on the unresolved text passes in one implementation, while a symlink inside the directory escapes later.

Resolve first, then check containment

Resolve the path to its canonical form with `fs.realpath`, which follows symlinks, and only then confirm it sits inside the root.

```
const root = await fs.promises.realpath('/srv/reports')
const resolved = await fs.promises.realpath(path.join(root, req.query.file))
if (resolved !== root && !resolved.startsWith(root + path.sep)) {
  return res.status(404).end() // outside the root: refuse, do not reveal
}
```

Better still, avoid interpreting user path input at all. Store files under generated IDs and map a requested ID to a server-owned path.

```
const files = { 'q1-report': '/srv/reports/2026-q1.pdf' }
const chosen = files[req.params.id] // unknown IDs simply do not resolve
```

The same discipline covers writes and deletes, not just downloads. A route that saves an upload or removes a temporary file is just as exposed if it joins user text onto a base directory without resolving and checking the result.

Normalizing dot segments does not solve symlink behavior by itself, because a symlink resolves to a path outside the root after the check has already passed. Server-generated IDs avoid most path interpretation and make the remaining checks smaller.

Test the file route with dot segments, absolute paths, encoded separators, and a symlink that points outside the root. Record the resolved target and prove every escape attempt is denied without revealing the external file.

Check your understanding: injection and output

Check XSS, SQL injection, command injection, path traversal, contextual output, and safe APIs.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Requests, files, and servers

Protect state changes, framing, uploads, and server-side network requests.

Prevent cross-site request forgery

Protect cookie-authenticated state changes with intentional methods, SameSite cookies, origin checks, and proven anti-CSRF tokens.

The browser attaches a user's cookies to a request based on where it is *going*, not where it started.

CSRF abuses that ambient authority: a hostile page triggers a request to your site, and the session cookie rides along automatically.

A signed-in user visits a hostile page containing an auto-submitted form to `/account/email`. The browser adds the session cookie even though the hostile page cannot read the response.

Layer the defenses

First, use non-GET methods for anything that changes state, and set `SameSite` on the session cookie so it is not sent on cross-site subrequests.

```
res.cookie('session', sessionId, { httpOnly: true, secure: true, sameSite: 'lax' })
```

Second, check request intent on the server. `Sec-Fetch-Site` (Fetch Metadata) or the `Origin` header let you reject cross-site writes.

```
function sameSiteWrite(req, res, next) {
  const site = req.headers['sec-fetch-site']
  if (site && site !== 'same-origin' && site !== 'none') {
    return res.status(403).end() // cross-site state change: refuse
  }
  next()
}
```

Third, add a per-session **anti-CSRF token** the server issues and verifies on each write. The hostile page cannot read the token, so it cannot forge a valid request.

```
if (req.body.csrf !== req.session.csrf) return res.status(403).end()
```

`SameSite=Lax` blocks many cross-site form posts, but legacy clients or required cross-site flows may need another design. Validate request intent on the server too.

Because XSS can read a token and defeat these controls, prevent both.

Submit a real state-changing request from a second origin and capture the server decision and unchanged account state. Repeat without the CSRF token or trusted origin, and test the legitimate form still works.

Stop clickjacking

Control which sites may frame a page and add confirmation for sensitive actions that should never be triggered through a disguised interface.

Clickjacking places your real page inside a frame on a hostile site, then overlays a decoy so the user clicks a genuine control they cannot see. The browser reports a legitimate authenticated click, because that is exactly what happened.

A transparent frame places the real ”Delete project” button under a fake ”Play” button. The click is genuine from the application's perspective.

Control who may frame you

The primary defense is the CSP `frame-ancestors` directive, which lists the origins allowed to embed the page. Keep the older `X-Frame-Options` header too, for clients that predate CSP.

```
Content-Security-Policy: frame-ancestors 'none'
X-Frame-Options: DENY
```

If one partner legitimately embeds a dashboard, name that origin instead of blocking everything.

```
Content-Security-Policy: frame-ancestors 'self' https://partner.example.com
```

You can set the header from the application for the routes that need it.

```
res.set('Content-Security-Policy', "frame-ancestors 'none'")
```

For destructive operations, add a second barrier that a hidden click cannot satisfy: require typed confirmation or recent authentication, so a single disguised click is never enough.

Blocking every frame can break a product that intentionally embeds a dashboard for one partner. Use a narrow `frame-ancestors` allowlist when embedding is required.

Build a local attacker page that frames the sensitive route and show the attack before the header change. Afterward, capture the browser refusal and confirm any approved embedding origin still works.

Handle file uploads

Validate upload purpose, size, type, content, storage, permissions, and delivery without trusting filenames or browser-provided metadata.

An upload is untrusted bytes plus untrusted metadata. The filename, the extension, and the `Content-Type` all come from the client and can all lie. Treat both parts as hostile until the server checks them.

A file named `avatar.png` carries HTML and script. If the server trusts the name and serves it inline from the application origin, the upload becomes active content.

Limit, inspect, rename, isolate

Set a hard size limit at the parser so a huge upload cannot exhaust memory or disk.

```
const upload = multer({ dest: '/srv/uploads', limits: { fileSize: 2 * 1024 * 1024 } })
```

Verify the *content*, not the claimed type. Sniff the real bytes and reject anything that is not on your narrow allowlist.

```
import { fileTypeFromFile } from 'file-type'
const kind = await fileTypeFromFile(req.file.path)
if (!kind || !['image/png', 'image/jpeg'].includes(kind.mime)) {
  return res.status(415).end() // not a real image: reject
}
```

Generate the storage name yourself and keep files outside any path the server will execute. Then serve them through a dedicated route that sets a safe type and forces download rather than inline rendering.

```
res.set('Content-Type', 'image/png')
res.set('Content-Disposition', 'attachment; filename="download.png"')
res.set('X-Content-Type-Options', 'nosniff')
```

Deep scanning adds time and can still miss new formats. Start with a narrow accepted format, strict size, verified parsing, generated storage names, and safe download headers.

Upload a valid image, an oversized image, HTML renamed as PNG, a malformed image, and two files with the same name. Record the decision, stored name, detected type, and response headers

for every case.

Control server-side requests

Prevent SSRF by restricting destinations, protocols, redirects, DNS behavior, credentials, and response sizes for server-side fetch features.

A server-side fetch runs from inside your network. It can reach internal services and cloud credentials the user's browser never could. **SSRF** turns a URL input into a path into that private space.

The naive importer trusts a user-supplied URL directly.

```
// Vulnerable: fetches wherever the user points
const image = await fetch(req.body.url)
```

An image importer accepts a public URL that redirects to `http://169.254.169.254`. The first host passes validation, but the server follows the redirect into cloud metadata.

Validate the destination you actually connect to

Prefer fixed provider endpoints. When you must accept a URL, allowlist the scheme, resolve the hostname, and reject private and metadata ranges. Because DNS can resolve differently on a second lookup, validate the resolved address.

```
import dns from 'node:dns/promises'
import ipaddr from 'ipaddr.js'

async function assertPublic(hostname) {
  const { address } = await dns.lookup(hostname)
  const range = ipaddr.parse(address).range()
  if (['private', 'loopback', 'linkLocal', 'uniqueLocal'].includes(range)) {
    throw new Error('blocked destination') // 169.254.x, 10.x, 127.x, etc.
  }
  return address
}
```

Then cap what the request can do: disable redirect following, bound the time, and limit the bytes read. Use a client with no ambient credentials attached.

```
const res = await fetch(url, { redirect: 'manual', signal: AbortSignal.timeout(3000) })
```

A DNS name can also resolve differently after an initial check. Validate the resolved destination used for each connection and cap redirects, time, and response bytes.

Test the fetcher with a public image, loopback, a private address, a metadata address, and a redirect to each blocked range. Capture the destination decision and prove the client sends no ambient credential.

Check your understanding: requests, files, and servers

Check CSRF, clickjacking, uploads, SSRF, request methods, redirects, and server-side fetches.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Access and configuration

Enforce access control and secure headers, errors, production settings, and admin paths.

Enforce object authorization

Use trusted identity and server-owned relationships for every read and write instead of treating object IDs or hidden controls as permission.

A signed-in user is not authorized for every record. Authentication proves *who* someone is; authorization decides *what* they may touch. Changing `/notes/123` to `/notes/124` must not cross an ownership boundary. This gap is called broken object-level authorization, and it is one of the most common web flaws.

The route below authenticates correctly and still leaks data.

```
// Vulnerable: checks the session but not ownership
app.get('/notes/:id', requireLogin, async (req, res) => {
  const note = await db.query('SELECT * FROM notes WHERE id = ?', [req.params.id])
  res.json(note)
})
```

Alice requests `/notes/42`, then changes the ID to Bob's note. The route sees a valid session and returns the record because the query checks only `id`.

Bind the query to the trusted owner

The fix scopes every query by the identity from the session, never from the request. The database returns nothing when the owner does not match.

```
const note = await db.query(
  'SELECT * FROM notes WHERE id = ? AND owner_id = ?',
  [req.params.id, req.user.id], // owner comes from the session, not the URL
)
if (!note) return res.status(404).end()
```

Apply the same rule to reads, updates, deletes, exports, search, and bulk operations. A bulk endpoint that checks only the first item is a common miss.

Hiding the note ID or using a UUID makes guessing harder but does not create permission. The query still needs the trusted owner or tenant relationship.

Use two valid accounts to test read, update, delete, export, and search for the same note. Save each response and verify Bob's stored data never appears or changes when Alice acts.

Set security headers

Add a deliberate baseline for framing, content types, referrers, browser features, transport, and content loading.

Security headers switch on browser protections and remove ambiguous behavior. Choose each one for your application rather than pasting an unexplained block from a blog post.

A useful baseline sets a few headers on every response.

```
Content-Security-Policy: default-src 'self'
```

```
X-Content-Type-Options: nosniff
Referrer-Policy: strict-origin-when-cross-origin
Permissions-Policy: camera=(), microphone=(), geolocation=()
```

`X-Content-Type-Options: nosniff` stops the browser from guessing a response's type, which blocks a text file from being run as script. `Referrer-Policy` trims what leaks in the `Referer` header. `Permissions-Policy` turns off browser features you do not use.

Add HSTS only after HTTPS is fully working, because it forces future requests onto HTTPS and cannot be undone quickly.

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

Cover every route, not just the home page

Apply headers in middleware so they reach errors and redirects too, then verify the routes people forget.

```
app.use((req, res, next) => {
  res.set('X-Content-Type-Options', 'nosniff')
  res.set('Referrer-Policy', 'strict-origin-when-cross-origin')
  next()
})
```

The home page has a strong header set, but framework-generated 404 and 500 responses omit it. An attacker targets the forgotten route where browser protections differ.

Copying every recommended header can break required features or advertise a false policy. Choose values from the application's real content, framing, and referrer needs.

Capture headers for a normal page, login, redirect, static file, 404, and forced 500. Compare them against the intended baseline and test one blocked framing or MIME-sniffing behavior in a browser.

Fail without leaking internals

Return useful public errors while keeping stack traces, queries, paths, tokens, and infrastructure details in protected diagnostics.

Errors help users recover and help attackers learn. A stack trace that reaches the browser can name your database, file paths, internal hostnames, and library versions. Separate the public message from the private diagnostic record.

A database outage returns a stack trace containing the SQL query, filesystem path, and internal host. The error helps the user no more than a safe failure would.

Log the detail, return a code

Catch failures at a central handler. Record everything internally, and send the client only a stable code and a request ID that ties the two together.

```
app.use((err, req, res, next) => {
  const requestId = req.id
  console.error({ requestId, message: err.message, stack: err.stack }) // stays server-side
  res.status(500).json({ error: 'internal_error', requestId })
})
```

The user can quote the request ID to support, and support can find the full context, but the response body reveals nothing about the internals.

Make sure the framework's development error page is off in production. In Express, `NODE_ENV=production` already stops the default handler from sending stack traces to the client.

```
NODE_ENV=production node server.js
```

Removing all detail can make support impossible. Return a stable public code and request ID while keeping the protected diagnostic record.

Test unexpected exceptions, not only clean validation failures, since those are the ones that leak.

Trigger validation, storage, and unexpected runtime failures in production mode. Save public responses and internal events, then prove the response contains no stack, query, path, host, or secret while the request ID connects both sides.

Protect administrative surfaces

Separate high-impact operations, require stronger proof, minimize exposure, log changes, and avoid hidden URLs as the main defense.

Administrative interfaces can change users, money, access, and production behavior. That impact makes them the highest-value target, and a secret-looking path like `/admin-x7f2` is not an authorization system. Anyone who learns the URL reaches it.

An administrator follows a saved link after leaving a privileged session open all day. A stolen browser session can now delete users without fresh proof.

Require role, then require freshness

Every admin operation needs an explicit role check on the server, not a hidden route and not a client-side menu that simply hides buttons.

```
function requireAdmin(req, res, next) {
  if (req.user?.role !== 'admin') return res.status(403).end()
  next()
}
```

High-impact changes need more than a valid session. Require recent authentication or MFA so a long-lived stolen session cannot perform them.

```
function requireRecentAuth(req, res, next) {
  const age = Date.now() - req.session.authAt
  if (age > 5 * 60 * 1000) return res.status(401).json({ reauth: true })
  next()
}

app.post('/admin/users/:id/delete', requireAdmin, requireRecentAuth, deleteUser)
```

Record a complete audit event for every attempt, keep normal accounts separate from admin use, and narrow network exposure where practical.

Requiring MFA for every small admin read can slow routine work. Reserve recent authentication and explicit confirmation for high-impact changes, while authorizing every operation.

Test one destructive admin action as an ordinary user, an administrator with a stale session, and

an administrator with fresh proof. Verify the first two fail, the third succeeds, and every attempt produces an audit event.

Check your understanding: access and configuration

Check access control, security headers, errors, production settings, administrative paths, and safe deployment.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Test and operate

Plan authorized tests, monitor abuse, review releases, and verify fixes.

Build a security test plan

Turn routes, roles, data flows, and threat stories into a repeatable plan that covers prevention and safe failure.

A generic checklist helps a little, but your application's own trust boundaries should drive the test plan. Start from what would actually hurt if it broke, not from a list someone else wrote for a different app.

Begin by mapping the surface: public routes, authenticated actions, roles, sensitive records, uploads, outbound requests, background workers, and admin functions. Each of those is a boundary worth a test.

Turn boundaries into concrete rows

For each feature, write a row that fixes the precondition, the request, and every expected outcome. A test is only repeatable when the expected state and expected log line are written down, not judged by eye.

```
Feature: update note
Precondition: Alice logged in, note 42 owned by Bob
Request:      PUT /notes/42  { "title": "x" }
Expected:     403, note 42 unchanged, one authorization-failure event
```

Test more than the happy path. For one feature, cover expected use, misuse, missing proof, alternate encodings, and limit boundaries.

```
- valid update by owner        -> 200, row changed
- update by non-owner          -> 403, row unchanged
- update with no session       -> 401
- oversized body               -> 413
- id as ../ or SQL characters  -> rejected, no error leak
```

A test plan covers login and form validation but misses the background export worker. That worker trusts a stored user ID and writes another tenant's data into the archive.

A huge generic checklist can hide this product-specific boundary. Start from assets, routes, roles, queues, files, and outbound requests. Keep evidence without storing live secrets.

Build a table with precondition, request, expected response, expected state, and expected event for one feature. Include a background or failure path, run the highest-impact rows, and attach their evidence.

Test only with permission

Keep active security testing inside an authorized scope and use local or intentionally vulnerable targets for destructive experiments.

Security tools send unusual and sometimes destructive traffic. Fuzzers, scanners, and enumeration probes can corrupt data, trigger real emails, or take a service down. Only test systems you own or have explicit, written permission to assess. This is a legal and ethical line, not a style preference.

Before running anything active, write the rules of engagement down.

```
Target:        staging.flaviocopes.com only
Window:        2026-08-10 02:00-04:00 UTC
Allowed:       read probes, auth checks, rate  5 req/s
Excluded:      production email provider, real customer records
Contact:       ops@flaviocopes.com
Stop when:     any 5xx spike or an excluded system is reached
```

Keep destructive experiments local

Owning the application does not make every connected system disposable. Payload experiments belong on a disposable local lab, not on shared staging.

```
# Run an intentionally vulnerable target locally for payload work
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

A scanner points at staging, but staging shares a production email provider. A harmless account-enumeration test sends thousands of real messages. The blast radius reached a system that was never in scope, because the scope was defined by "the app I own" instead of every integration behind it.

Scope the targets, techniques, data, integrations, rate, contacts, and stop conditions explicitly. Rate-limit your tools and preserve the availability of shared environments, since other people depend on them. When you find a real issue, report it through the agreed contact rather than exploiting it further.

Write an authorization for one staging test and have the system owner confirm it. Run a bounded harmless probe, verify rate and stop conditions, then test that an excluded target is never contacted.

Monitor web abuse

Create actionable signals for access failures, suspicious input, unusual data movement, configuration changes, and administrative actions.

A prevented request still tells you an attack is underway. A denied read is a normal event on its own, but a pattern of them is evidence. The goal is to capture that signal without recording the dangerous payload or the credential someone tried.

Log the security-relevant fact, and be careful what you put in the fields.

```
log.warn('authz.denied', {
```

```
  actorId: req.user?.id,        // safe identifier, not the raw token
  action: 'note.read',
  targetId: req.params.id,
  outcome: 'denied',
  ip: req.ip,
})
```

Alert on patterns, not on every denial

One failed object read is common. Five hundred failures across sequential note IDs from one session is evidence of enumeration. Aggregate before you alert.

```
alert: object-enumeration
when:  authz.denied grouped by actorId
       count > 100 within 5m
       across > 20 distinct sequential targetIds
route: security-oncall  (link: dashboard + runbook)
```

Aggregate repeated authorization failures, rate-limit decisions, upload rejections, outbound-fetch blocks, and admin changes. Alerting on every denial overwhelms responders and buries the useful pattern in routine noise.

Give responders enough context to act safely: a safe actor id, the operation, the target pattern, time, and outcome, plus an owner and a link to investigate. An alert with no owner and no next step becomes noise that people learn to ignore.

Watch the same way for unusual data movement and configuration changes, not only failed reads. A sudden large export, a new admin grant, or a burst of blocked outbound fetches each deserves its own aggregated signal.

Generate normal mistakes and a burst of sequential cross-user reads in a test environment. Show the normal traffic stays below the threshold, the burst creates one actionable alert, and the alert contains an owner and investigation link.

Complete a web security review

Review architecture, code, dependencies, configuration, tests, and recovery as one release decision instead of one final scanner run.

Web security lives across the whole delivery path, so the review is a release decision, not a final scanner run. A clean scan proves no known-bad dependency shipped. It says nothing about whether your business rules are correct.

A dependency scan is clean, but a new bulk-delete route checks authorization only for the first note. The release still contains a high-impact application flaw. No automated tool would flag that, because the code is valid; the *logic* is wrong.

Review the release as a whole

Walk the changed trust boundaries, not the whole app. For this release, check where new input enters, where authorization decisions live, and what dependencies and configuration changed.

```
[ ] threat model revisited for changed boundaries
[ ] authorization checked on every new read and write, including bulk
```

```
[ ] negative tests run (wrong user, missing proof, bad input)
[ ] dependency and deployment findings triaged
[ ] logs and alerts fire for the new paths
[ ] rollback rehearsed
```

Record each finding as a decision with an owner and a date, so nothing is silently forgotten.

```
Finding: bulk-delete authorizes only the first item
Path:    POST /notes/bulk-delete with mixed-owner ids
Impact:  high - cross-tenant deletion
Decision: fix before release   Owner: flavio   Date: 2026-08-03
```

Blocking every low-risk finding delays fixes that matter. Make each ship, fix, or accept decision from a reproducible path, impact, owner, and date. Block release when a credible high-impact path remains open.

Review one release across changed boundaries, authorization, input, dependencies, production configuration, monitoring, and rollback. Attach evidence for every finding and rerun one regression check after the deployed fix.

Check your understanding: test and operate

Check threat-driven testing, safe tools, logging, deployment review, regression tests, and incident response.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/web-application-security/.