Security Fundamentals Course

Flavio Copes

Updated August 6, 2026

Contents

Security thinking **2**
- Start with assets and harm . . . 2
- Model threats and attackers . . . 3
- Draw trust boundaries . . . 4
- Prioritize real risk . . . 4
- Check your understanding: security thinking . . . 5

Protect by default **6**
- Apply least privilege . . . 6
- Choose secure defaults . . . 7
- Use defense in depth . . . 8
- Minimize data and authority . . . 9
- Check your understanding: protect by default . . . 10

Data and secrets **10**
- Separate encoding, hashing, and encryption . . . 10
- Protect data in transit . . . 11
- Manage application secrets . . . 12
- Use secure randomness . . . 13
- Check your understanding: data and secrets . . . 14

Find and fix weaknesses **14**
- Validate at the boundary . . . 14
- Patch with context . . . 15
- Write security tests . . . 16
- Use security tools with judgment . . . 17
- Check your understanding: find and fix weaknesses . . . 18

Detect, respond, and recover **18**
- Log useful security events . . . 18
- Prepare an incident plan . . . 20
- Contain without destroying evidence . . . 21
- Restore and learn . . . 22
- Check your understanding: detect, respond, and recover . . . 23

Learn to threat-model software, choose practical controls, protect data and secrets, test important failure paths, and prepare for incidents.

What you'll learn: Review a small product from assets and trust boundaries through prevention, detection, response, and tested recovery.

Security thinking

Identify assets, attackers, trust boundaries, and the risks worth addressing first.

Start with assets and harm

Identify what a product must protect and describe the concrete harm that follows if confidentiality, integrity, or availability fails.

Security starts with what matters. A generic instruction like ”make the app secure” gives us nothing we can test.

Before you think about attackers or tools, list the **assets**: the data, actions, services, money, and reputation the product depends on. Then describe what happens if someone reads, changes, destroys, or blocks each one. This turns security from fear into a set of product requirements.

The three kinds of harm

Every asset can fail in three distinct ways:

- **Confidentiality** fails when someone reads data they should not see.
- **Integrity** fails when someone changes or deletes data without permission.
- **Availability** fails when legitimate users cannot reach the data or the operation.

Imagine a shared notes app that stores private drafts and recovery email addresses. A leaked draft harms confidentiality. An overwritten draft harms integrity. An author locked out of the editor before a deadline is an availability failure, and it hurts even though nothing leaked.

Build the asset table

Write the assets down. A short table forces you to be concrete:

```
asset             owner         location            worst believable harm
private drafts     note author   postgres "notes"    competitor reads unreleased work
recovery emails   account team  postgres "users"    attacker resets accounts at scale
publish action    note author   POST /api/publish   defaced public page under our brand
nightly backup    ops           S3 bucket           silent corruption blocks every restore
```

Each row names an owner and a location. That matters later, because ”who can rotate this credential” and ”where does this data live” are the first questions in an incident.

The mistake to avoid

A list that only says ”user data” hides these differences. It can lead us to protect database secrecy while missing deletion, account lockout, or a broken restore.

Check each row by asking for one observable failure per kind of harm:

```
asset: private drafts
read    -> another account downloads the draft      (confidentiality)
change  -> draft overwritten with defaced content   (integrity)
block   -> author cannot open the editor            (availability)
```

If you cannot describe what the failure would look like in a log, a screenshot, or a support ticket, the asset is still too vague to protect.

My advice: keep this table in the repository, next to the code. When a feature adds a new asset, add a row. This table is the input for every lesson that follows.

Model threats and attackers

Describe attacker goals, capabilities, entry points, and abuse cases without trying to predict every possible exploit.

A **threat** is a believable way an asset can be harmed. We do not need a movie villain or a list of every hacking tool.

A useful threat model answers one question: what can this attacker already do before the attack begins? Start from existing access, not from imagined superpowers.

Name the actors

For a typical web app, the cast is short:

- anonymous visitors, who can reach public pages and endpoints
- signed-in users, with a valid account of their own
- compromised accounts, where the credentials are real but the person behind them is not
- malicious dependencies, pulled in during a build
- insiders, with dashboard or database access
- automated bots, hammering forms and APIs

Give each actor a goal and an entry point. "A signed-in user wants to read other people's drafts through the notes API" is testable. "Hackers want to hack us" is not.

The most common attacker is ordinary

Here is the scenario that plays out constantly. A signed-in user opens their own note and looks at the request:

```
GET /api/notes/1842
Cookie: session=eyJhbGciOi...
```

They change `1842` to `1841` and read another person's draft. No special tool. The attacker needs only a valid account and a missing ownership check on the server.

This is why threat models become vague when every attacker is "advanced." Start from the access an ordinary user already has, then add stronger capabilities only when the design supports them.

Write attacker stories

Capture each threat as a short story with four parts:

```
actor:            signed-in user (free tier)
goal:             read drafts belonging to other accounts
entry point:      GET /api/notes/:id
expected control: server checks note owner against session user
```

Three of these stories are worth more than a fifty-row spreadsheet nobody reads.

Then test the cheapest one. Create two test accounts, sign in as the first, and request a note owned by the second. Save the request and the result. If data comes back, you found a real weakness

before writing any tooling. If you get a denial, keep that evidence too — it proves the expected control exists in the running system.

Draw trust boundaries

Mark where data or authority crosses between browsers, servers, databases, third parties, administrators, and local machines.

A **trust boundary** is a place where data or authority changes hands. Most security bugs appear because code trusts something too early.

The browser and server are different trust zones. So are your server and a webhook provider, or your application and a package downloaded during a build. Validate the crossing even when both systems belong to you.

Find the boundaries in a real request

Follow one note-creation request through a typical stack:

> **Diagram: Trust boundaries in a note creation request.** A browser crosses the public network with a session cookie and note data, a load balancer forwards the request, and the application uses database credentials to reach Postgres. Data crosses at least three boundaries here. The browser sends a title and body you must not trust. The session cookie must map to a real user before anything acts on it. The app talks to Postgres with credentials that prove it is the app, not some random process on the network.

Two requests, two proofs

Now add a webhook from your payment provider. It reaches the same server as a browser request, but it carries a provider signature instead of a user session:

```
POST /api/notes           proof: session cookie  -> user identity
POST /webhooks/payments   proof: HMAC signature  -> provider identity
```

Treating both requests alike can grant the webhook the wrong authority. A webhook has no user session, so code that reads "the current user" from it either crashes or, worse, falls back to a default account. Each boundary has its own kind of proof, and the server must check the right one.

Label what crosses, not just the boxes

A diagram with boxes but no crossing data is decoration. For each arrow, label three things: the identity, the payload, and the credential that cross. "Browser to server: anonymous or session user, JSON note body, session cookie" tells you exactly what to verify at that line.

Then test one crossing with the proof removed. Send the note-creation request without the session cookie. Send the webhook without its signature header. Both should fail with a clear denial. If either succeeds, you found code trusting a boundary it never verified — exactly the class of bug this exercise exists to catch.

Prioritize real risk

Compare likelihood, impact, exposure, and recovery cost so limited security work addresses the most meaningful risks first.

Not every possible weakness deserves the same response. We have limited time, so priorities matter.

Risk is not a feeling. For each weakness, estimate four things: how reachable it is, what an attacker gains, how many users are affected, and how hard recovery would be. Then fix easy, high-impact exposure first.

Compare two findings

Take two findings from the same notes app:

```
finding A: public note export endpoint has no ownership check
  reachable:  today, by any signed-in user with curl
  gain:       every draft in the database
  affected:   all accounts
  recovery:   painful - we cannot un-leak data

finding B: notes may persist in a server-side cache after deletion
  reachable:  requires server access first
  gain:       fragments of recently deleted notes
  affected:   some accounts
  recovery:   flush the cache
```

Both sound alarming in a report. But finding A is reachable today by anyone with an account, while finding B requires the attacker to already be on your server — and if they are, you have a bigger incident than the cache. A gets fixed this week. B goes on the backlog with a note.

Record assumptions, not scores

Teams like numeric scores because they look objective. Exact-looking scores can hide weak assumptions: a "7.4" says nothing about why, and it goes stale silently.

Keep the ranking comparative and write down what you assumed:

```
rank 1: export endpoint  - assumes the endpoint is enabled in production
rank 2: cache leak       - assumes attackers lack server access
reorder trigger: any sign of server compromise
```

That last line is the useful part. It records what evidence would make the priority change, so the ranking updates when reality does instead of when someone happens to remember it.

The failure mode

The common mistake is fixing whatever was reported most recently, or whatever a scanner painted red. A week spent on a theoretical issue is a week the reachable one stayed open.

Rank three risks from your own project using the four questions. Attach one piece of evidence to the top one — a request, a log line, a config value. Then change one assumption and check whether the order changes. If nothing ever changes it, your ranking is decoration.

Check your understanding: security thinking

Check assets, threats, trust boundaries, risk, and secure-by-design decisions.

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

Take this interactive quiz online

Protect by default

Apply least privilege, secure defaults, defense in depth, and data minimization.

Apply least privilege

Give people, processes, tokens, and services only the access they need for the task and only for as long as needed.

Every permission increases what a mistake or compromise can damage. **Least privilege** means you start with no access, then add the narrow capability the task needs.

The principle sounds easy until you look at real credentials. A reporting job may read invoices but should not delete them. A deployment token may publish one project but should not administer the account. Most systems drift the other way, because broad access is granted once and never reviewed.

Why the scope of a token matters

A nightly reporting job needs to read invoice totals. Someone creates it with the team's admin credential because that already works. Months later the credential appears in a CI log — logs leak constantly, through verbose error dumps or debug flags left on.

Now compare outcomes. With a read-only credential, the leak exposes invoice data. Bad, but bounded. With the admin credential, one leaked CI log turns a reporting failure into data loss, because whoever reads the log can delete every invoice.

The database version of this is two statements:

```
CREATE ROLE reporting LOGIN PASSWORD 'use-a-generated-value';
GRANT SELECT ON invoices TO reporting;
```

No `DELETE`, no `UPDATE`, no access to other tables. The blast radius of a compromise is now the same as the blast radius of the job itself.

Verify the denial, not just the success

A privilege model you never tested is a guess. Run one normal reporting task with the read-only credential and capture the successful result. Then attempt a delete with the same credential:

```
DELETE FROM invoices WHERE id = 4211;
-- ERROR: permission denied for table invoices
```

Keep that denied response as evidence. The error is the point: it proves the boundary exists in the running system, not just in a design document.

The convenience objection

Narrow access can make operations less convenient. One day someone genuinely needs to fix bad invoice rows, and the read-only role blocks them. The wrong answer is widening the routine credential "temporarily" — temporary grants become permanent because nothing forces their removal.

Keep a separate, audited recovery path instead: a break-glass role that requires a second person or writes a log entry every time it is used. And review privileges when roles, services, and projects change, because yesterday's necessary access is today's unnecessary risk.

Choose secure defaults

Make a new account, deployment, or feature safe before the user changes any optional security setting.

Defaults become reality for most users. A security control hidden behind an optional setting protects fewer people than you think.

When someone creates an account, a bucket, or a project, they accept whatever the system chose for them. Almost nobody reads the settings page first. So the defaults are a security decision you make on behalf of every user who never touches them.

What a secure default looks like

Four rules cover most cases:

- Keep new resources private until someone explicitly shares them.
- Deny operations you do not recognize instead of allowing them.
- Grant restrictive permissions and let users widen them deliberately.
- Require an explicit decision before anything becomes publicly reachable.

Cloud storage is the classic cautionary tale. For years, public buckets leaked data not because attackers were clever, but because ”public” was one unchecked box away and nothing forced a decision.

A quiet default becomes a repeated leak

Here is the failure in miniature. A project-creation endpoint treats visibility as optional:

```
const project = await db.projects.create({
  name: body.name,
  visibility: body.visibility ?? 'public', // the bug is this fallback
})
```

Every client that omits the field creates a public project. Most users never revisit the setting, so a quiet default becomes a repeated data leak — one new exposure per created project, until someone notices.

The fix is one word:

```
  visibility: body.visibility ?? 'private',
```

If compatibility genuinely needs a weaker mode, make the tradeoff visible and temporary: log it, surface it in the UI, and put an expiry date on the exception.

Test the default from outside

Do not trust the code review. Create a project without touching any optional security setting and record its resulting visibility and permissions. Then open a signed-out browser and try to reach it:

```
curl -i https://app.example.com/projects/8231
# HTTP/1.1 404 Not Found   <- what you want to see
```

Require a denial before calling the default safe. A `200` with project data means your default is doing the attacker's work for them.

A restrictive default may add one explicit publishing step for people who want public projects. That small cost is easier to explain than silently exposing every new project.

Use defense in depth

Combine independent preventive, detective, and recovery controls so one failed safeguard does not become a complete compromise.

One perfect defense does not exist. Good systems assume a control can fail, and arrange for the failure to be survivable.

Defense in depth means combining controls that catch different failures: input validation, database constraints, authorization checks, safe output handling, logs, and backups. Each solves a different part of the problem. Prevention stops the bad action, detection tells you it was attempted, and recovery limits the damage when both fall short.

Layers in a real endpoint

Take a notes API. Three controls guard the same cross-user read:

```
// layer 1: the route checks ownership
const note = await getNote(id)
if (note.ownerId !== session.userId) return res.status(403).end()
```

```
-- layer 2: the query itself is scoped
SELECT * FROM notes WHERE id = $1 AND owner_id = $2;
```

```
layer 3: an audit event records the denial
event=note.read.denied user=u_9412 note=n_1841 request=req_77af
```

Now suppose a refactor deletes the route check. The scoped query still returns zero rows, so the read stays blocked. And the audit event reveals the attempt, so you learn the first layer is gone before an attacker teaches you. That is what independence buys: the second control does not care that the first one vanished.

Fake depth

Layers help only when they are independent. Repeating the same assumption in five places is still one weakness.

The common version: the client sends an owner ID in the request body, and the route, the service layer, the query builder, and the audit log all use that client-supplied value. Five checks, one assumption — that the client tells the truth. One forged request defeats all of them at once. The server-side session is the only trustworthy source of identity here, so every layer must derive ownership from it.

Prove the depth exists

Pick one sensitive operation and name its preventive, detective, and recovery controls. If you cannot name all three, you have already found your gap.

Then rehearse the failure. In a test environment, disable the first control — comment out the route

check — and confirm the query still denies the read and the event still records the attempt. Depth you have never tested is depth you only hope you have.

Minimize data and authority

Collect less sensitive data, keep it for less time, and avoid building powerful operations the product does not need.

The safest sensitive record is the one you never collect. The safest administrative action is the one your product does not expose.

Every field, log line, export, token, and integration you add is something that can leak, be stolen, or be abused. Reducing stored data and powerful capabilities shrinks both the attack surface and the incident you may one day need to explain to your users.

Question every field

Ask why each piece of data exists, who owns it, and when it gets deleted:

```
field                why we store it           retention
email                login + receipts          account lifetime
date of birth        nobody remembers          none defined
full reset URL       "useful for debugging"    90 days in logs
last 4 card digits   support lookups           account lifetime
```

Two rows in that table should worry you. The date of birth has no owner and no purpose — delete the column. The full reset URL is worse, and it deserves its own story.

The reset-URL log

An analytics log stores complete password-reset URLs for debugging. Each URL contains a live reset token. The token later expires, but every log reader had account-reset authority during its lifetime. The logging pipeline, the dashboard, the third-party log vendor — all of them briefly held the keys to any account that requested a reset.

Nobody designed that. It happened because logging the full URL was the easy thing to do.

Removing the field can reduce debugging context during support work, and support will notice. Keep a safe event identifier or a token fingerprint instead of the reusable secret:

```
log.info('password_reset.requested', {
  user: user.id,
  tokenFingerprint: sha256(token).slice(0, 12), // correlates, cannot be redeemed
})
```

You can still match a support ticket to the exact reset event. You just cannot take over the account from the log.

Authority counts too

The same logic applies to what the product can do. A ”delete all users” admin endpoint that no real workflow needs is pure downside: it will never help you, and it might help an attacker.

Find one sensitive value kept in a field, log, or export and document why it exists. Remove or redact it, then prove the product still supports its real task and the raw value no longer appears. Repeat the audit every few months. Data grows back.

Check your understanding: protect by default

Check least privilege, secure defaults, defense in depth, data minimization, and failure behavior.

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

Take this interactive quiz online

Data and secrets

Separate cryptographic operations, protect transport, manage secrets, and generate tokens safely.

Separate encoding, hashing, and encryption

Choose the correct operation by distinguishing representation, one-way fingerprints, and reversible protection with a key.

Base64 is not encryption. A hash is not a way to hide data you need to recover later. Mixing these up is one of the most common security mistakes in real codebases.

Three different operations, three different jobs:

- **Encoding** changes representation so data survives a channel. Anyone can reverse it. No key, no secret.
- **Hashing** produces a fixed-size fingerprint and is designed to be one-way. Same input, same output, no way back.
- **Encryption** protects confidentiality using a key, and modern encryption must also protect integrity, so nobody can silently modify the ciphertext.

See the difference

```
echo -n "customer record" | base64
# Y3VzdG9tZXIgcmVjb3Jk        <- reversible by anyone

echo -n "customer record" | shasum -a 256
# 6cc3c... a fixed-size fingerprint, one-way
```

The Base64 output looks scrambled, which is exactly why it fools people. Scrambled-looking is not protected.

The mislabeled backup

A backup tool Base64-encodes customer records and labels the result encrypted. Anyone who downloads the file can decode it without a key — one `base64 -d` away from every record. The label created false confidence, which is worse than no protection, because nobody adds real encryption to something already marked ”encrypted.”

The other two operations would also be wrong here. Hashing the backup would prevent recovery: you cannot restore customers from fingerprints. Encryption without integrity could allow silent changes to the ciphertext. The correct operation follows the security goal we started with — recoverable confidentiality with tamper detection, which means authenticated encryption.

Pick by goal, not by function name

Classify by asking what you need afterwards:

```
need                                operation
send binary data inside JSON        encoding (Base64)
detect file corruption              hash (checksum)
check passwords at login            password hash (bcrypt/argon2, slow on purpose)
store data we must read back        authenticated encryption
```

Passwords get a special row. They use dedicated slow hashes, never plain SHA-256 and never encryption — you should be unable to recover a user's password even with every key you own.

Take a session token, a file checksum, a password record, and a recoverable backup from your own system and classify each by goal and operation. Then decode the encoded one, alter one byte of the protected one, and record which checks detect the change.

Protect data in transit

Use authenticated TLS for network connections and understand what transport encryption does and does not protect.

TLS protects a connection from passive reading and tampering, as long as certificate validation succeeds. That last clause carries all the weight.

Use HTTPS for the whole application, not only the login page. Every request carries the session cookie, so one plaintext page is enough to steal an authenticated session on a hostile network.

What TLS does not do

TLS is a transport guarantee, nothing more. It does not authorize the signed-in user, validate application input, or protect data after the endpoint decrypts and processes it. A perfectly encrypted connection will happily deliver a forged request or a malicious payload.

And it authenticates endpoints, not intentions. TLS proves you are talking to the server named by the certificate. It does not prove the signed-in user may reset another account. Authorization is a separate control for a reason.

The second hop

Here is where real systems leak. The browser uses HTTPS, but the application sends reset requests to an internal email service over plaintext HTTP:

```
browser --HTTPS--> app server --HTTP--> internal email service
                                ^ reset tokens cross here in plaintext
```

The secure browser connection does not protect that second hop. Anyone who can observe internal traffic — a compromised container on the same network, a misconfigured switch port — reads live reset tokens. Internal service traffic can carry sensitive data too. ”It's inside our network” is an assumption, not a control.

Trace one sensitive flow, like a password reset, across every network hop. For each hop, capture the scheme and the certificate result:

```
curl -sv https://mail.internal.example.com/send 2>&1 | grep "SSL certificate verify"
# SSL certificate verify ok.
```

Fail closed, never fall back

The most dangerous TLS bug is the quiet fallback: code that retries over plain HTTP when the handshake fails, or an HTTP client configured to skip certificate verification "temporarily." Both convert a visible outage into an invisible leak.

Break certificate validation on purpose in a test environment — point the client at a host with a self-signed certificate — and require the request to fail closed with an error. If the data goes through anyway, you found a fallback path that an attacker with network position can trigger whenever they want.

Manage application secrets

Keep credentials out of source code and logs, scope them narrowly, rotate them, and make local and production access deliberate.

API keys, signing keys, database passwords, and webhook secrets are credentials. Treat them as temporary authority, not configuration text.

The difference matters. Configuration describes how the app behaves. A secret is power: whoever holds it can act as your application. Power needs an owner, a scope, and a revocation plan.

Where secrets live

Three rules cover storage:

- Production secrets belong in the platform secret manager, injected as environment variables or bindings at runtime, only into the services that need them.
- Local development uses a gitignored `.env` file with development-only values.
- Nothing secret ever ships to browser code. If JavaScript in the browser can read it, so can every visitor.

Source code is the classic leak. Once a key is committed, it lives in git history even after you delete it from the current file. Rotate any secret that was ever committed, then add scanning so it does not happen again:

```
echo ".env" >> .gitignore

npx gitleaks git .
# scans the repository history for committed credentials
```

One key, three consumers

One shared API key powers development, production, and a contractor script. It works fine until the provider flags abuse. Now you have two bad options: revoke the key and stop all three systems at once, or leave it live while you investigate. And the sharing means you cannot even tell which consumer caused the abuse.

Separate secrets add rotation work. They also let us revoke one integration without turning a small leak into a full outage. One consumer, one credential.

Rotation is a muscle

A secret you cannot rotate is a time bomb. Plan the leak before it happens: for each production secret, know the owner, the scope, every consumer, and the exact rotation procedure.

```
secret: STRIPE_WEBHOOK_SECRET
owner: payments team
scope: verify webhook signatures only
consumers: app server (production)
rotate: provider dashboard -> roll secret -> update secret manager -> deploy
```

Then rehearse it in a test environment. Rotate the secret, prove the new value works, and prove the old value fails. The second half is the part teams skip — and it is the only part that tells you revocation actually revokes.

Use secure randomness

Generate session identifiers, reset tokens, nonces, and keys with a cryptographically secure system source instead of predictable application randomness.

Security tokens must be difficult to guess. That sounds trivial until you see what real code uses: timestamps, counters, `Math.random()`, or UUID variants built from weak randomness. All of them look random. None of them are unpredictable.

The distinction matters because an attacker does not need to guess the whole token. They need to shrink the search space. A token derived from the clock has a few thousand plausible values in the window around "now." That is a for-loop, not a defense.

Use the platform generator

Every platform ships a cryptographically secure random source. Use it, and do not reimplement it:

```
// a random identifier with strong randomness built in
const token = crypto.randomUUID()

// or explicit bytes when you control the format
const bytes = crypto.getRandomValues(new Uint8Array(32))
const reset = Buffer.from(bytes).toString('base64url')
```

Request enough bytes — 16 is a floor, 32 is comfortable for a reset token — and encode them safely for the channel, like `base64url` for URLs. Then keep the raw token out of logs, because a perfectly random token in a log file is a perfectly readable one.

`Math.random()` exists for shuffling playlists. It is seeded from predictable state and was never designed to resist prediction.

The predictable reset link

A reset link is built from the user ID plus the current timestamp:

```
// looks random enough, is enumerable
const token = `${userId}-${Date.now()}`
```

An attacker who knows the target's email requests a reset, notes the time, and tries nearby timestamp values until one works. Account taken over, no exotic vulnerability needed, because the token was never secret — only obscure.

Length would not have saved it. A long token is not strong when its source is predictable. And a custom "mixing" step — hashing the timestamp, appending the user ID — adds obscurity, not randomness. The unpredictability of the output can never exceed the unpredictability of the inputs.

Verify your generator

Find where your app generates reset tokens and record which API it calls and how many bytes it requests. Generate a few thousand tokens and check format and uniqueness. Then freeze time in a test and generate more: with a mocked clock, a secure generator keeps producing unpredictable values, while a timestamp-derived one collapses into repeats. That one test separates real randomness from decoration.

Check your understanding: data and secrets

Check encoding, hashing, encryption, TLS, secure randomness, secrets, and data classification.

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

Take this interactive quiz online

Find and fix weaknesses

Validate boundaries, patch with context, write security tests, and review tool findings.

Validate at the boundary

Treat external data as untrusted, parse it into a known shape, and enforce authorization and invariants before use.

Every request, file, webhook, database record, and third-party response can be malformed or hostile. **Validate at the boundary**: parse external data into a known shape before it gains any authority inside your code.

The order matters. Validation happens first, at the edge, once. Code past the boundary should receive typed, bounded, checked values — never the raw request.

What to check

For each field: type, length, range, format, and allowed values. A schema library makes this declarative:

```
import { z } from 'zod'

const CreateNote = z.object({
  title: z.string().min(1).max(200),
  body: z.string().max(50_000),
  visibility: z.enum(['private', 'shared']),
}).strict() // reject unknown fields

const result = CreateNote.safeParse(req.body)
if (!result.success) return res.status(400).json({ error: 'invalid input' })
```

Reject unknown fields when they create risk. The classic failure is a client sending `role: "admin"` and a careless spread operator copying it straight into the database.

```
curl -s -o /dev/null -w "%{http_code}\n" -X POST http://localhost:3000/api/notes \
  -H "Content-Type: application/json" \
```

```
  -d '{"title":"ok","body":"hi","visibility":"private","role":"admin"}'
# 400 - the unknown field bounced at the boundary
```

Valid is not the same as safe

Here is the gap validation fills. A create-note endpoint accepts a valid string title with five million characters. The JSON parses fine — it is a string, after all. But the request consumes memory, bloats every log line that echoes it, and fills logs before the database rejects it. Type-correct, and still a denial-of-service vector. The `max(200)` bound kills it at the door.

Also know what validation does not do. It does not replace safe parameterized queries, output encoding, or authorization; each solves a different problem. A perfectly validated note title can still belong to another user, and a valid string can still break an HTML page that renders it unescaped.

Strictness meets rollouts

A strict schema may reject a new client field during a rollout: the mobile app v2 sends `tags`, the server does not know it yet, and every v2 request bounces. That is not a reason to accept every unknown value forever. Version the contract deliberately — add the field to the schema before the client ships.

Exercise one boundary with five requests: valid, missing field, wrong type, oversized, and extra field. Save the response and the database state for each, including proof that rejected input caused no write. A `400` that still inserted a row is two bugs, not one.

Patch with context

Track software versions, evaluate exposure, test updates, and ship security fixes without waiting for perfect certainty.

A known vulnerable component is not automatically exploitable in your product. It is still a signal you must investigate, and ”investigate” has concrete steps.

Dependency scanners hand you a steady stream of advisories. Panic-patching everything wastes the week; ignoring everything loses the one advisory that matters. Context is how you tell them apart.

Four questions per advisory

- The affected version: is the vulnerable range what you actually run?
- The reachable feature: does your code call the vulnerable path?
- The privilege required: anonymous visitor, or authenticated admin?
- The possible impact: code execution and data exposure outrank a crash.

Start with what is installed:

```
npm audit
npm ls sharp   # who pulls this in, and which version resolves
```

`npm ls` matters because lockfiles resolve transitive versions you never chose. The advisory applies to what is installed, not to what your `package.json` mentions.

Reachability is the honest test

An advisory affects image parsing, and the application processes user uploads through that exact function. That is reachable: anonymous users feed bytes directly into the vulnerable code. Patch

it now.

The tempting dodge is "the package is indirect." Indirect does not make the path unreachable — your direct dependency calls it for you, with your users' input. Trace the call path from your route handler down to the vulnerable function and save that evidence. It justifies the urgency today and explains the decision next quarter.

Then apply the smallest supported fix and test the important path:

```
npm install sharp@0.33.5   # the patched release, not a new major
npm test
```

A rushed major upgrade can break production, and a security patch that causes an outage teaches the team to fear patching. Patch releases exist so the fix and the redesign travel separately. If a safe reproducer for the malformed input exists, run it against the fixed version too.

When you genuinely cannot patch yet

Sometimes the fixed version needs a migration you cannot ship today. Deferring is legitimate. Silent deferring is not. Record the reason, a compensating control, an owner, and a deadline:

```
advisory: GHSA-xxxx (image parser overflow)
deferred: fix requires Node 22, migration ticket INFRA-412
mitigation: uploads disabled for unverified accounts
owner: platform team    review by: 2026-09-01
```

A dated mitigation with a name on it gets revisited. An unwritten one becomes permanent.

Write security tests

Turn important abuse cases into repeatable tests for authorization, validation, secret handling, rate limits, and safe failure.

A security requirement becomes much stronger when a test proves it. "Users cannot read each other's notes" is a sentence. A test that fails the day someone breaks it is a control.

Start with the failures that would hurt most — usually authorization — and work down. Keep these tests beside the behavior they protect, in the same suite, running on every commit.

The two-user pattern

Most authorization bugs are invisible to single-user tests. A route test signs in Alice and confirms she can read her note. Green. It never asks whether Bob can read the same note, so the missing ownership check stays invisible for as long as nobody thinks to ask.

Every ownership test needs two identities:

```
test('users cannot read notes they do not own', async () => {
  const note = await createNote(alice, { title: 'draft' })

  const res = await request(app)
    .get(`/api/notes/${note.id}`)
    .set('Cookie', bob.sessionCookie)

  assert.equal(res.status, 403)
```

```
})
```

A 403 is not enough

For mutations, the status code is weak evidence on its own. A `403` alone proves little if the unauthorized update still reached a background job, or the row changed before the check fired. Assert three layers: the response, the stored state, and the emitted event.

```
test('denied update changes nothing', async () => {
  const note = await createNote(alice, { title: 'original' })

  const res = await request(app)
    .patch(`/api/notes/${note.id}`)
    .set('Cookie', bob.sessionCookie)
    .send({ title: 'hijacked' })

  assert.equal(res.status, 403)

  const stored = await db.notes.find(note.id)
  assert.equal(stored.title, 'original') // state unchanged

  const events = await capturedEvents('note.update.denied')
  assert.equal(events.length, 1) // the attempt was recorded
})
```

Widen the net

The same discipline covers the rest of your requirements. Send malformed and oversized input and assert clean rejection. Verify secrets never appear in public output — a test that greps API responses for token prefixes is cheap and catches painful bugs. Assert that a rate limit engages after repeated failed logins. And exercise expired and revoked credentials, not only missing ones: ”signed out yesterday” is a different code path from ”never signed in.”

Add the two-user test for one read and one mutation in your own project, then run the mutation test again with an expired session. Each test turns a security promise into something the build enforces.

Use security tools with judgment

Combine automated scanning with manual review, reproduce findings, and fix root causes instead of chasing unverified alerts.

Scanners are useful assistants. They do not understand the complete product or business rule, so their output is a list of signals, not a list of vulnerabilities.

Dependency scanners, static analysis, and dynamic scanners each see one slice: manifests without runtime, code paths without context, running behavior without source. Combine them with manual review, because each one is blind where the others see.

One rule comes before everything else: run tools only against systems you own or have permission to test. A dynamic scanner throws real attack traffic. Pointing one at a production system you do not own is an attack, whatever your intentions.

Reproduce before you rank

A finding you have not reproduced is a rumor. For each important one, reproduce it safely, inspect the affected path, and rank it using context the scanner does not have: is the endpoint reachable anonymously, what data sits behind it, what would recovery cost?

Here is why the manual step matters. A scanner reports a reflected value in a JSON response as XSS:

```
finding: reflected-xss (medium confidence)
source:  GET /api/notes?q=...
sink:    JSON response field "title"
```

You trace the path. The frontend renders that field with `textContent`, which never interprets HTML, so the reported path does not execute script:

```
noteTitle.textContent = data.title  // inert - no script execution
// noteTitle.innerHTML = data.title // this version would be vulnerable
```

But notice the trap on the other side: calling this a false positive without reproducing the browser flow is also a mistake. If some other template renders the same field with `innerHTML`, the finding is real. Tools find signals; the complete data path from source to sink determines impact. You only know after you looked.

Tune, do not mute

A noisy scanner trains people to ignore it, which is worse than not running it. Tune rules that consistently misfire on your stack, and write down why each rule was tuned. What you may not do is hide a result because it is inconvenient — suppressions with no recorded reason are how known vulnerabilities survive three audits in a row.

Take one real finding from any scanner and walk the full source-to-sink path in an authorized environment. Save the request, the response, the execution result, and a regression test, whether you confirm or reject the finding. The rejected ones matter too: the note explaining why the path is safe saves the next person a day of re-investigation.

Check your understanding: find and fix weaknesses

Check input boundaries, patching, security tests, scanners, disclosure, and remediation decisions.

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

Take this interactive quiz online

Detect, respond, and recover

Create useful evidence, prepare an incident plan, contain harm, restore, and learn.

Log useful security events

Record authentication, authorization, configuration, data-access, and administrative events without leaking credentials or sensitive payloads.

If an incident happens, we need evidence. A security log should answer who did what, when, from where, and whether it succeeded — for every event that matters.

The events that matter are stable across most products: authentication (logins, failures, resets), authorization denials, configuration changes, sensitive data access, and administrative actions. If a support engineer exported all users yesterday, the log should say so.

The shape of a useful event

Structure beats prose. Use stable event names, timestamps, actor and target identifiers, outcomes, and a request correlation ID:

```
{
  "event": "auth.login.failed",
  "time": "2026-08-03T14:12:09Z",
  "actor": "user_9412",
  "source_ip": "203.0.113.7",
  "outcome": "failure",
  "reason": "bad_password",
  "request_id": "req_77af0c"
}
```

The `request_id` is the thread that ties one user action together across services. The stable `event` name is what makes queries possible:

```
grep '"event":"auth.login.failed"' app.log | wc -l
# counting guesses is a one-liner when the name never changes
```

Under-logging and over-logging fail together

Watch the failure from both sides. A login event that records ”failed” but no safe account identifier, source, or request ID gives responders nothing during an attack — they cannot separate one mistyped password from thousands of guesses across hundreds of accounts. The event existed and answered no questions.

The reflex fix — log everything about the request — creates the opposite disaster. Logging the submitted password would add evidence by creating a second breach: every failed login stores a string that is probably almost the user's real password, readable by everyone with log access.

Never log passwords, raw session tokens, reset links, private keys, or personal data you do not need. Record enough context to correlate behavior without storing credentials. Identifiers and fingerprints, not secrets.

Logs are themselves a target

An attacker who can edit logs erases their own trail. An attacker who can read them harvests whatever leaked into them. Restrict who can read and modify logs, and define retention deliberately: long enough to investigate an incident you discover late, not so long that the archive itself becomes a liability.

Trigger one successful login, one failed login, and one denied cross-user read in your own app. Verify the three events share stable names and request IDs, then search the captured output for passwords, session tokens, and reset links. Empty search results are the pass condition.

Prepare an incident plan

Decide roles, communication paths, evidence handling, containment options, and recovery priorities before a security incident occurs.

An incident is the worst time to decide who can shut down production. Decisions made at 02:00, under pressure, with half the team asleep, are the worst decisions a team makes all year. Prepare the first moves while the system is healthy.

An incident plan for a small team is not a fifty-page policy. It is one page that answers the questions the first responder will actually have.

Names, not roles

Write down the incident lead, the technical contact per system, who holds decision authority (who may take production down, who may notify customers), the customer communication owner, and how to reach external providers. Names, with a fallback for each — the primary contact is on a plane exactly when you need them.

```
incident lead:    marta          fallback: jonas
prod shutdown:    marta or CTO
customer comms:   elena
cloud provider:   support plan 4412, portal + phone
```

Keep the supporting information current and reachable without production access: the architecture diagram, the asset table from module one, the secret inventory with owners, and the backup locations with restore steps. A plan that lives only behind the VPN that is down is not a plan.

The 02:00 test

Run the scenario in your head. A production database credential appears in a public build log at 02:00. The on-call developer can see it but does not know who may rotate it or whether rotation restarts the app. So they wait for the morning standup, and the credential stays live for seven hours.

That gap is what the plan removes. The relevant page looks like this:

```
secret: DATABASE_URL (production)
owner: marta (backup: jonas)
rotate: update in secret manager, redeploy - ~2 min downtime
authority: on-call may rotate WITHOUT approval at any hour
evidence: export the build log BEFORE rotating
```

Two lines carry the weight. ”Evidence first” preserves the timeline before containment erases it. And the authority line pre-approves the safe action, so nobody waits for permission at night. A long policy is useless if the first responder cannot find the credential owner; the first-hour plan needs names, authority, commands, and a fallback contact.

Rehearse one scenario

A plan that has never been exercised is fiction. Run a timed tabletop for the leaked credential: walk through detection, decision, rotation, and communication, and record every decision and delay. Confirm the contacts answer, the rotation works in a test environment, and the team knows how to preserve the relevant logs. Every delay found in rehearsal is a delay removed from the real incident.

Contain without destroying evidence

Limit ongoing harm, preserve useful evidence, and avoid impulsive changes that make the incident harder to understand.

Containment stops the damage from growing. It is not the same as immediately deleting every affected system — and the difference decides whether you can ever answer "what did they access?"

During an active incident, two jobs run at once. Containment limits ongoing harm. Investigation reconstructs what happened. Impulsive destructive actions serve the first job by killing the second.

The standard containment moves

- Revoke exposed credentials, so stolen authority stops working.
- Isolate compromised components — pull them from the load balancer or the network rather than wiping them.
- Block known abuse at the edge: IPs, tokens, endpoints.
- Preserve relevant logs and snapshots before anything changes or expires.

Record each action and its time as you go:

```
02:14 confirmed stolen token is creating unauthorized releases
02:17 exported provider audit events (before any changes)
02:19 revoked the deployment token
02:21 verified a deploy attempt with the old token fails
```

That timeline has two readers: the teammate joining mid-incident, and future-you writing the review next week.

The instinct that destroys the timeline

A stolen deployment token is still creating releases. The panicked instinct is to delete the malicious releases and the build history — make the bad thing disappear. Deleting the build history stops the visible activity but destroys the timeline and leaves the token valid. You erased your only record of what the attacker did, and they can still deploy.

The order that works: preserve the history first, revoke the credential second, clean up last. Revocation is the move that actually stops the attacker, and it destroys nothing.

Containment has costs, and that is acceptable. Revoking the deployment token may temporarily stop all deployments, including your urgent fix. Plan a separate recovery identity — a break-glass deploy credential kept offline — so containment does not lock the responders out too.

Coordinate the destructive steps

Any action that cannot be undone — wiping a server, deleting data, rotating every key at once — goes through the incident lead first. That is not bureaucracy. It is the mechanism that keeps recovery and investigation possible while individual responders move fast on the reversible actions.

Rehearse this with a disposable token in a test project. Order the containment actions, run them, and prove two things at the end: the token can no longer deploy, and its identifier and activity history are still available. Containment succeeded when the attacker's access is dead and the evidence is alive.

Restore and learn

Recover from trusted backups, verify the repaired system, communicate clearly, and turn incident evidence into lasting improvements.

Recovery is not "the server starts again." We need confidence that the attacker's access is gone, the original weakness is fixed, and the data we restored is complete. Skip any of those and you restart the incident with extra steps.

Restoring after a security incident

Restoring after a compromise carries a trap that ordinary disaster recovery does not: the backup may contain the compromise. Restore from known-good inputs — a snapshot from before the intrusion, checked against the timeline you preserved during containment.

Then, in order:

- Rotate every credential the attacker could have touched. A restored system that still accepts stolen passwords is not recovered.
- Patch the root cause before going live. Restoring the same vulnerable code invites the same visitor back.
- Monitor for recurrence with specific checks — the indicators from this incident, not generic dashboards.

Trust no backup you have not restored

The nastiest failure hides here. A backup job reports success every night, but the archive misses uploaded files referenced by the database. The restore runs cleanly, the service starts, and every attachment in the product is broken — discovered only now, when it is too late to fix the backup job.

The defense is dull and non-negotiable: test backups before an incident.

```
pg_restore --dbname=restore_drill backup_2026-08-01.dump

# verify relationships, not just "it started"
psql restore_drill -c "SELECT count(*) FROM attachments a
  LEFT JOIN files f ON f.id = a.file_id WHERE f.id IS NULL;"
# 0 means no attachment points at a missing file
```

Time the whole drill. Recovery time includes validation, credential rotation, and chasing missing dependencies. Measuring only download time gives the team false confidence about a number someone may one day promise a customer.

The learning half

Afterward, write a blameless timeline: what happened, when it was detected, what each response step cost. Blameless is not politeness — people who fear blame omit the details you need most.

End with concrete improvements, each with an owner and a date:

```
action: alert on >20 failed logins/hour per account
owner: marta    due: 2026-09-12
action: move build logs to private storage
owner: jonas    due: 2026-08-21
```

”Improve monitoring” fixes nothing; a named, dated action might. Then close the loop: restore one dated backup into an isolated environment, verify record counts, relationships, files, and one real user flow, and test one intentionally corrupted or incomplete backup so you know what failure looks like before it matters.

Check your understanding: detect, respond, and recover

Check security events, incident preparation, containment, recovery, backups, and learning after incidents.

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/security-fundamentals/.