Practical Debugging Course

Flavio Copes

Updated August 3, 2026

Contents

Debug code, browser requests, runtimes, and production incidents with reproducible evidence, focused tools, regression tests, and safe recovery actions.

What you'll learn: Move from symptom to first failing boundary, prove the cause, ship a regression-tested repair, and preserve a useful incident record.

Debugging method

Reproduce the problem, define expected behavior, minimize the case, and test one hypothesis.

Make the bug reproducible

Turn a vague report into exact starting state, actions, actual result, expected result, and frequency.

Debugging begins with a repeatable observation. “Sometimes broken” is not enough to tell whether a change helped. If the bug fires 5 times out of 5 before your fix and 0 times out of 5 after it, you learned something. If it was already random, you learned nothing.

Why a vague report is not a starting point

A typical report says ”login is broken”. That sentence hides everything you need: which account, which browser, which server, which build, what actually happened on screen. Two people can read it and imagine two different bugs.

Your first job is to turn the report into a procedure someone else can execute. A **reproduction card** is a short, exact recipe: starting state, actions, actual result, expected result, frequency, environment.

Write a reproduction card:

```
starting state: signed out, empty local storage
action: open /login, submit valid lab account
actual: spinner remains forever
expected: dashboard loads
frequency: 5 of 5 attempts
environment: Chrome, local server, commit abc123
```

Every line earns its place. `frequency: 5 of 5` tells you the bug is deterministic, so any future run that passes is real signal. `commit abc123` pins the code, so you can return to the exact broken version after experiments.

Verify the card actually reproduces

Have another person or a clean browser profile follow only the card. Don't stand behind them filling in missing steps. If they cannot trigger the bug, the card is incomplete — some precondition lives in your head or your browser. Add any missing precondition they discover.

The common failure mode: the bug reproduces only on your machine because of hidden state. A stale cookie, a local `.env` override, a cached response. When the clean profile works, that difference is your first clue, not a nuisance. Record it on the card.

Intermittent bugs

When you truly cannot make the failure deterministic, make the frequency measurable instead. ”Fails about 1 run in 10 under `npm test`” still supports experiments. You just need more runs per experiment before you trust the result.

Use disposable accounts and remove secrets, tokens, and private customer data from the reproduction. The card will get pasted into issues and chats, so it must be safe to share.

Separate facts from hypotheses

Write observed evidence apart from possible explanations and choose the next test that can disprove one.

The fastest way to waste a debugging session is to treat a guess as a conclusion. A hypothesis is useful only when a test can make it less likely. Do not turn the first plausible story into a fact.

Facts are things you observed: a status code, a log line, an error message you can paste. **Hypotheses** are stories that would explain the facts. Keep them physically apart, in writing, so your brain cannot quietly promote one to the other.

Create two columns:

```
facts:
- POST /login returns 500
- response request ID is r-42
- server log r-42 says connection refused

hypotheses:
- database process stopped
- database address is wrong
- firewall rejects the connection
```

Notice what the facts already rule out. ”connection refused” means the request reached the server, and the server tried to open a connection somewhere. The frontend is innocent. That is why writing facts down pays off: each one prunes the search space before you touch anything.

Test to disprove

Choose one cheap test for each hypothesis. For ”database process stopped”, check the process list on the database host. Ten seconds. For ”database address is wrong”, print the host and port the app actually resolved. Record the result before changing configuration.

The discipline matters more than the order: run the test, write down what happened, cross the hypothesis out or keep it. A crossed-out hypothesis is progress even though nothing is fixed yet.

Say it out loud

Explaining the two columns to a colleague — or to nobody, which is **rubber duck debugging** — forces you to state each assumption explicitly. Halfway through the sentence ”the app connects to the database on port 5432...” you go check the config and it says 5433. The duck did nothing. You did, by turning a silent assumption into a checkable claim.

Avoid making several fixes at once. If you restart the database, correct the address, and open the firewall in one step, the system may recover — and you lose the evidence that identifies the cause. Next time it happens, you start from zero.

Reduce to a minimal case

Remove unrelated code, data, requests, and environment until the failure has the smallest useful reproduction.

A bug inside a full application hides behind hundreds of moving parts. A smaller failing case exposes the responsible boundary and makes experiments faster. Every component you remove is

a component you no longer have to suspect.

The method: remove one dimension while preserving the bug. If the bug survives the cut, the removed part was innocent. If the bug disappears, you just found where it lives.

Reduce an application request:

```
full page -> one API call
one API call -> one input payload
production database -> disposable fixture
framework route -> direct function call
```

After every reduction, rerun the same assertion. The assertion is your anchor — "the total comes back as `NaN`" — and it must stay identical while everything around it shrinks.

A worked reduction

Say the checkout page shows `NaN` as the total. Start wide, then cut:

```
// step 1: call the API directly - still NaN, frontend is innocent
// step 2: call the function directly with the request payload
import { total } from './cart.mjs'

console.log(total([
  { name: 'notebook', price: 4.5 },
  { name: 'pen', price: undefined },
]))
// NaN
```

Two cuts, and the bug is one function plus one small payload. Keep cutting the payload. Remove the notebook line: the bug survives. Remove the pen line instead: the bug disappears. The minimal case is a single item with `price: undefined`, and the question has become sharp: why does that item exist in the cart data at all?

Stop removing when the behavior disappears, then compare the last two cases. The difference between the smallest case that fails and the largest case that passes is usually one line, and that line names the cause.

What not to minimize with

Do not minimize against live customer data. Copy only the smallest sanitized shape needed — above, two fake items reproduce what a real 40-item cart did. The minimal case also becomes your regression test input later, so it needs to be shareable and free of anything private.

Change one variable

Design a controlled experiment, predict both outcomes, and preserve the result before the next change.

If you change code, configuration, dependencies, and data together, success reveals almost nothing about which change mattered. The bug is gone, but you cannot say what was wrong, whether the other changes are safe, or whether the fix survives the next deploy.

Debugging experiments work like controlled experiments anywhere: one variable per run.

Write the experiment before you run it

Write the experiment first:

```
hypothesis: API base URL points at the old port
change: override only API_BASE_URL to port 4000
prediction if true: request connects and reaches authentication
prediction if false: connection failure remains
```

The two predictions are the important part. If you cannot say what ”true” and ”false” look like before the run, the experiment cannot teach you anything. You will read any outcome as vaguely supporting whatever you already believed.

Run it once in the same starting state. Same data, same browser profile, same commit. If your reproduction card says the bug fires 5 of 5 times, one clean run is enough to read the result.

Restore between experiments

Restore the variable before testing another hypothesis. This is the step everyone skips. After three unrestored experiments you are debugging a system nobody has ever seen: old port plus new flag plus edited config. When something changes now, you cannot attribute it to anything.

A practical pattern in a git repository:

```
git stash        # park the current experiment
git stash pop    # bring it back when you need it
```

One stash per hypothesis keeps each change isolated and reversible. Config overrides work the same way: set them on the command line for one run instead of editing files, and they clean themselves up.

When runs are expensive

In production you rarely get free experiments. Keep emergency production changes small, authorized, logged, and reversible — and still write the prediction first. A rollback with a written prediction (”error rate returns to baseline within five minutes”) is an experiment too. If the prediction fails, the deploy was not the cause, and you just saved yourself from ending the investigation too early.

Check your understanding: debugging method

Check the commands, decisions, safety boundaries, and failure cases from debugging method.

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

Take this interactive quiz online

Debug code

Read stack traces, add focused logs, pause execution, and convert the failure into a regression test.

Read the stack trace

Start at the error message and first relevant application frame instead of reacting to the longest library path.

A stack trace shows the chain of calls present when an error was created. The bottom frame started everything; the top frame is closest to the failure. Most developers' real mistake is not reading it at all — they see red text and start guessing.

Run a tiny failure:

```
function total(items) {
  return items.reduce((sum, item) => sum + item.price, 0)
}

total(undefined)
```

Node prints:

```
TypeError: Cannot read properties of undefined (reading 'reduce')
    at total (/srv/app/cart.mjs:2:16)
    at file:///srv/app/cart.mjs:5:1
```

Read it in this order. First the **exception type and message**: `TypeError`, something was `undefined` and we asked it for `reduce`. The message already names the broken value — the thing `reduce` was read from, so `items`. Then the first frame: `total` at `cart.mjs` line 2, where the error surfaced. Then the callers below it: line 5 is who passed `undefined`.

That means the fix belongs at the caller or at the function boundary, not on line 2. Fix the invalid caller or validate the function boundary rather than hiding the trace behind a try/catch that swallows it.

Library frames are noise, usually

In a real application the top frames often belong to a framework. The top frame is closest to the failure, but framework wrappers may appear before your code:

```
    at Array.reduce (<anonymous>)
    at total (/srv/app/cart.mjs:2:16)
    at handleCheckout (/srv/app/routes/checkout.mjs:14:20)
    at Layer.handle (/srv/app/node_modules/express/lib/router/layer.js:95:5)
```

Scan for the first frame in your own code: `cart.mjs`, then `checkout.mjs`. The `node_modules` frames explain how execution got there, not what went wrong. The longest, scariest path in the trace is almost never the interesting one — unless every frame is library code, which usually means you passed a library something invalid.

Do not return stack traces to public clients. They reveal file paths, dependency names, and internal structure. Keep the detail in protected logs, and send the client a generic error with a request ID that lets you find the full trace later.

Add focused diagnostic logs

Log event, request identity, state category, timing, and error without dumping secrets or entire objects.

When you cannot attach a debugger — production, a background worker, someone else's machine — logs are how the process tells you what happened. But `console.log('here')` and `console.log(user)` produce noise you cannot search and dumps you cannot share. A useful log answers what happened and connects related work. Random values without event names create noise.

Log events, not values

Give every diagnostic line an **event name** and the identifiers that tie it to one request:

```
console.error({
  event: 'invoice_load_failed',
  requestId,
  invoiceId,
  error: error.message,
})
```

This one line is searchable (`invoice_load_failed`), joinable (the same `requestId` appears in the access log and in downstream calls), and specific (which invoice). Compare that to `console.log('error!', error)`: you cannot grep for it, you cannot tell which request produced it, and it might print an entire response object.

Place logs at boundaries

The highest-value log points sit where data crosses a boundary: request received, external call started, external call finished with its duration, decision taken. A before/after pair around a suspicious call answers the two questions diagnostic logging is actually good at — did we get here, and with what:

```
console.error({ event: 'invoice_fetch_start', requestId, invoiceId })
const invoice = await fetchInvoice(invoiceId)
console.error({ event: 'invoice_fetch_done', requestId, ms: Date.now() - started })
```

Trigger one failure and locate the event by request ID. If you can follow one request through your logs from entry to error, the instrumentation works. If you find yourself scrolling, you logged too much or connected too little.

What must never be logged

Confirm the output avoids tokens, cookies, full customer records, and stack duplication. Log `error.message` and identifiers, not whole objects — a full user record in a log file is a data leak with a retention period. Production logs require access and retention controls, same as the database they describe.

Temporary logs need removal or deliberate promotion. Before closing the bug, either delete the diagnostic lines or keep them intentionally as permanent, named events. A codebase littered with forgotten `debug 2` lines makes the next investigation harder, not easier.

Pause with a debugger

Set a breakpoint, inspect values and scope, step through control flow, and change no code until the state is understood.

Print debugging costs one edit-and-rerun cycle per question. A debugger lets you stop execution before the wrong result spreads, then ask as many questions as you want about that frozen moment. Inspect actual runtime state instead of adding many speculative logs.

Neither tool is always right. Logs win in production and across many requests. A debugger wins when you have one reproducible failure and need to see state you did not think to print.

Start Node with the inspector

Start a Node process with the inspector:

```
node --inspect-brk app.mjs
```

`--inspect-brk` pauses on the first line, so you have time to set breakpoints before any code runs. Plain `--inspect` starts executing immediately. Open `chrome://inspect` in Chrome and click your process under Remote Target — DevTools attaches to the paused process.

Attach DevTools, pause at the relevant function, then work a fixed loop:

```
1. set a breakpoint on the line before the suspected failure
2. resume, wait for the pause
3. read local values and the call stack - is the state what you assumed?
4. step over (next line) or step into (enter the call), deliberately
```

The payoff moment: a variable holds a value you were sure it couldn't. That is the assumption the whole bug was hiding behind, and staring at source code would never have caught it. You can also drop a `debugger` statement in the code; any attached inspector pauses there.

Conditional breakpoints

When a function runs 500 times and fails once, a plain breakpoint is torture. Right-click the breakpoint and add a condition like `item.price === undefined`. Execution pauses only on the failing iteration, replacing the whole log-and-grep cycle.

The safety rule

Do not attach an unauthenticated remote debugger to a public interface. The inspector protocol can evaluate arbitrary code — debuggers can execute code with process authority. Keep the inspector bound to `127.0.0.1`, and tunnel over SSH when you must debug a remote machine.

And change no code until the state is understood. The debugger's value is observing the system exactly as it is; the moment you start editing mid-session, you are experimenting, and that belongs in the change-one-variable workflow.

Write a regression test

Capture the smallest failing input as a test, see it fail, apply the repair, and keep the test permanently.

A fix without a test is temporary. A regression test proves the old behavior is present before the fix and prevents the same failure from returning unnoticed. It is also the cheapest artifact to produce right now: the minimal failing case from your reduction work is the test input.

Fail first

The order is what makes it a regression test. Write it, watch it fail, then fix.

Express the expected behavior:

```
import assert from 'node:assert/strict'
import test from 'node:test'

test('empty cart totals zero', () => {
  assert.equal(total([]), 0)
})
```

Run the test before the code change:

```
node --test cart.test.mjs
#   empty cart totals zero
# AssertionError: Expected 0, got NaN
```

That failing run is evidence the test actually exercises the bug. Now apply the repair and run again. The pass means the fix works, and the red-to-green transition means the test guards the right thing.

Skip the failing run and you risk a test that passes for the wrong reason: it calls a different code path, or its assertion is too loose to catch anything. A test that never failed proves nothing.

Assert the contract, not the bug

Do not write a test that passes because it reproduces the bug's wrong result. `assert.equal(total([]), NaN)` would pass against today's broken code and permanently enshrine the failure. Ask what the function should do — the intended contract — and assert that.

Name the test after the behavior, not the ticket. `'empty cart totals zero'` tells the next reader what is protected. `'fixes bug #4521'` tells them to go read a ticket that may not exist anymore.

Keep it, and keep it small

Add a nearby edge case only when it protects a distinct boundary. An item with a missing price is a different contract than an empty cart, so it earns its own test. Resist adding ten speculative cases; each one you write now is one somebody maintains forever.

The test stays in the suite permanently. Six months from now someone refactors `total()`, and this test is the only thing standing between them and reintroducing your bug.

Check your understanding: debug code

Check the commands, decisions, safety boundaries, and failure cases from debug code.

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

Take this interactive quiz online

Debug the web path

Use browser and curl evidence across requests, responses, CORS, TLS, storage, and frontend state.

Inspect the Network panel

Find the failing request, inspect URL, initiator, timing, headers, body, status, and response.

When a page misbehaves, the Network panel shows what the browser actually sent and received — not what your code intended. The browser Network panel records requests while it is open, so the first move is always: open the panel, then reload.

Find the request that matters

Start from the first failed or unexpected request, not every red line at once. Failures cascade. One failed script load produces twenty downstream errors, and only the first one is interesting. Sort by time and find the earliest anomaly.

Enable **Preserve log** when the bug involves navigation or redirects. Without it the panel clears on every page load, and the evidence disappears with it.

Record one request as a compact report:

```
method and final URL
initiator
status or browser error
request headers and body shape
response headers and body
timing waterfall
redirect chain
```

Each line answers a distinct question. The final URL catches requests that landed somewhere unexpected after redirects. The initiator column names the code that triggered the request — click it to jump to the exact line. The status distinguishes a server that answered with an error from a request that never completed, which shows a browser error like `net::ERR_CONNECTION_REFUSED` and no status at all.

Read the timing waterfall

Click a slow request and open the Timing tab. The phases tell different stories:

```
Stalled            queueing, connection limits
Waiting (TTFB)     the server thinking - backend problem
Content Download   large response or slow network
```

A request with eight seconds of Waiting is a server-side investigation. Eight seconds of Content Download on a 40 MB JSON response is a payload problem. Same ”slow request” symptom, opposite fixes — the waterfall picks the direction.

Sharing what you found

Export a sanitized HAR only when needed: right-click, ”Save all as HAR”. HAR files can contain authorization headers, cookies, query secrets, and private response bodies for every recorded request, not just the one you care about. Sanitize before sharing, or share the compact report above instead of the raw capture.

Reproduce the request with curl

Copy the essential HTTP request outside the browser and remove headers until only the failing contract remains.

A failing request seen in the browser has three suspects: your frontend code, the browser itself, and

the server. curl separates server behavior from browser UI and frontend code. If curl gets the same failure, the frontend is innocent, and you just saved hours in the wrong layer.

Build the request from scratch

DevTools offers "Copy as cURL", but the copied command drags along a dozen headers and every cookie. A copied request can still include unnecessary or secret browser state. My advice is to go the other way: start minimal and add only what the failure needs.

Start with a minimal request:

```
curl --verbose --fail-with-body \
  --header 'Content-Type: application/json' \
  --data '{"email":"student@lab.test"}' \
  http://127.0.0.1:3000/api/login
```

`--verbose` prints the full exchange — request headers out, response headers in — so you see exactly what the server received and answered. `--fail-with-body` makes curl exit non-zero on HTTP errors while still printing the error body, which is where the useful message usually lives.

Compare status, headers, and body with the browser request. Three outcomes, three conclusions:

```
curl fails the same way   -> server-side bug, debug the backend
curl succeeds             -> the difference is browser state: cookies,
                             auth, Origin, or frontend code
curl fails differently    -> compare the two requests header by header
```

Add differences one at a time

When curl succeeds but the browser fails, the cause is in whatever the browser sends that you did not. Add the browser's headers one at a time — the session cookie, then `Origin`, then any custom header — rerunning after each. The header that flips curl from success to failure names the broken contract.

Add authentication only through a disposable credential when the endpoint requires it. A lab account token is fine; your own production session is not.

Remove copied cookies and tokens before saving or sharing the command. A curl command pasted into a ticket is effectively a stored credential if it embeds a live session — treat it like one.

Classify CORS and TLS errors

Distinguish a browser CORS policy block, a server HTTP response, and a TLS identity failure.

Browser console errors mix three different failures that need three different fixes. Classifying the error correctly is most of the work.

CORS is enforced by browsers after or around an HTTP exchange: the server may have responded fine, but the response lacked the `Access-Control-Allow-Origin` header your page's origin needs, so the browser withheld it from your JavaScript. **TLS failure** happens before protected HTTP — the connection never became trustworthy, so no request went through at all. A plain HTTP error (a 403, a 500) means everything worked except the application.

The key tool fact: curl does not enforce browser CORS policy. That difference is diagnostic.

Compare evidence:

```
curl -v https://api.lab.test/data
# browser: inspect Console and Network panels for CORS details
# TLS: openssl s_client -connect api.lab.test:443 -servername api.lab.test
```

If curl gets a normal response while the browser reports CORS, the server and network are fine. Inspect the `Origin` request header and the `Access-Control-*` response headers — the server is not authorizing your page's origin, or the preflight `OPTIONS` request fails.

If curl also fails with a certificate complaint, it is TLS, and the `openssl s_client` output names which kind:

```
certificate has expired       -> validity window, renew it
hostname mismatch             -> cert issued for another name
unable to get local issuer    -> server sent an incomplete chain
```

If TLS fails, repair certificates first. CORS symptoms on top of broken TLS mean nothing — the browser never got far enough for CORS to matter.

The fix that is worse than the bug

Do not fix CORS by reflecting every Origin with credentials. Echoing whatever origin arrives into `Access-Control-Allow-Origin`, combined with `Access-Control-Allow-Credentials: true`, lets any website make authenticated requests as your logged-in users. Authorize the exact browser origins required, as a fixed list.

The same discipline applies to TLS: never "fix" it by disabling verification with `curl -k` in a saved script or `NODE_TLS_REJECT_UNAUTHORIZED=0`. That turns a loud error into a silent opening for a man-in-the-middle.

Inspect browser state

Check cookies, local storage, IndexedDB, cache, service workers, and an isolated profile without clearing evidence first.

"It fails for me but works for everyone else" is usually not the code. It is the state the browser accumulated. Persistent browser state can make one user fail while a clean profile works: an old session cookie, a service worker serving last week's build, a cached API response.

The reflex to resist is clearing everything immediately. Compare state before deleting it — a wiped profile that suddenly works tells you state was involved, but destroys the evidence of which state.

Compare, then bisect the state

Open a private window or a fresh profile and run the exact reproduction from your card. Record a controlled comparison:

```
normal profile: failure
private profile: works
cookie difference: old session cookie
service worker: active version 12
cache: response served from disk
```

Now you have a bounded suspect list. Use the DevTools **Application panel** to inspect each category: Cookies with per-cookie expiry, Local storage, IndexedDB, Cache storage, Service workers.

Check ownership and expiry — a cookie scoped to the parent domain, set long ago by a different subdomain, is a classic invisible culprit.

Remove one state category at a time and rerun the exact reproduction:

```
1. unregister the service worker  -> still fails
2. clear cache storage            -> still fails
3. delete the session cookie      -> works
```

The bug is now specific: the app mishandles a stale session cookie instead of redirecting to login. That is a fixable code path. "Clear your cookies" was never the fix — it was the symptom eraser, and the next user with an old cookie hits the same wall.

Service workers deserve extra suspicion

An active service worker can serve every asset and API response from its own cache, making the site immune to your deploys. The Application panel shows the active version; check "Update on reload" to force fresh fetches while you test. If the failing user runs worker version 12 and everyone else runs 14, you have found the mechanism.

Clearing all site data may hide the cause and log the user out. Preserve evidence, and use a disposable account when the reproduction needs real login state.

Check your understanding: debug the web path

Check the commands, decisions, safety boundaries, and failure cases from debug the web path.

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

Take this interactive quiz online

Debug the runtime

Inspect configuration, permissions, processes, ports, dependencies, and build artifacts.

Compare effective configuration

Print safe configuration names and sources, then compare working and failing environments without exposing values.

The code is identical, but it works locally and fails deployed. Most of the time the difference is configuration: a missing variable, wrong scope, stale deployment, or unexpected default. The trap is debugging the config files. What matters is the **effective configuration** — what the process actually received at startup.

A variable can be set in `.env`, overridden by the shell, replaced by the deploy platform, or absent because nobody added it to production. Inspect what the process received, not what the files say it should receive.

Log a redacted startup summary

Log a redacted startup summary:

```
console.log({
```

```
  nodeEnv: process.env.NODE_ENV,
  apiHostConfigured: Boolean(process.env.API_HOST),
  databaseUrlConfigured: Boolean(process.env.DATABASE_URL),
})
```

The `Boolean()` wrapping is deliberate. It answers "is it set?" without printing the value — `DATABASE_URL` contains a password, and this line will land in log storage. Never print secret values. Presence, source, version, and a safe fingerprint are usually enough.

Diff the environments

Run the same summary everywhere and put the outputs side by side:

```
local: { nodeEnv: 'development', apiHostConfigured: true,  databaseUrlConfigured: true }
prod:  { nodeEnv: 'production',  apiHostConfigured: false, databaseUrlConfigured: true }
```

`apiHostConfigured: false` in production ends the investigation. The app fell back to a default host, and the default points somewhere wrong. Compare local, test, and deployed startup evidence this way for any config-shaped bug — the diff is the diagnosis.

The restart trap

Verify the process was restarted after changing environment configuration. Environment variables are read at process start. Editing the platform's settings screen changes nothing for the process already running. This produces the most common false conclusion in config debugging: "I set the variable and it still fails" — because the failing process has never seen it.

When a value must be compared but not shown, log a fingerprint instead: the first 8 characters of its SHA-256 hash. Two environments with different fingerprints hold different values, and nobody learned the secret.

Diagnose file and permission errors

Resolve the effective process user, path, ownership, mode, parent directories, and mount state before changing permissions.

`EACCES`, `EPERM`, `ENOENT` — file errors look alike from inside the application, but the causes live in different places. Before touching any permission, establish four facts: who the process runs as, which path it actually opened, what the permissions along that path are, and which filesystem the path lands on.

The rule most people miss: a process needs permission on the file and traversal permission on every parent directory. A world-readable file inside a `700` directory owned by someone else is unreachable, and the error message blames the file.

Inspect one failing path

Inspect one failing path:

```
id
namei -l /srv/app/data/report.json
stat /srv/app/data/report.json
findmnt /srv/app/data
```

`id` shows the effective user and groups — check it for the service account, not for yourself. `namei -l` is the underused gem: it walks the path one component at a time and prints the owner and mode of each:

```
f: /srv/app/data/report.json
drwxr-xr-x root root /
drwxr-xr-x root root srv
drwxr-x--- root deploy app        <- www-data is not in "deploy"
drwxr-xr-x app  app  data
-rw-r--r-- app  app  report.json
```

The file itself is readable, but the `app` directory blocks anyone outside the `deploy` group. That is the whole diagnosis, in one command.

`findmnt` catches the other class: the path sits on a read-only mount, a container volume that never got mounted, or a network filesystem that remaps ownership. `stat` confirms the exact mode and owner, and whether the ”file” is actually a symlink pointing somewhere else.

Reproduce as the real user

Run the failing operation as the actual service account:

```
sudo -u www-data cat /srv/app/data/report.json
```

If that reproduces the application error, you have a minimal case. Fix it narrowly: change only the ownership or mode needed by that operation — add the service user to the right group, or grant group read on one directory.

Do not use recursive `chmod 777`. It makes the error vanish by destroying the ownership model, it exposes data to every account on the machine, and it guarantees the real misconfiguration is still there, now invisible.

Diagnose processes and ports

Identify the listener, address, protocol, process, and duplicate instance behind a port failure.

“Port in use” and “connection refused” are opposite clues. `EADDRINUSE` means something already holds the address you tried to bind. `ECONNREFUSED` means you reached the machine and nothing was listening there. One indicates an existing bind; the other often indicates no listener at the reached address. Both get diagnosed the same way: find out who is actually listening, and where.

Find the listener

Inspect TCP port 3000:

```
ss -lntp | grep :3000
# macOS: lsof -nP -iTCP:3000 -sTCP:LISTEN
```

A typical line:

```
LISTEN 0 511 127.0.0.1:3000 0.0.0.0:* users:(("node",pid=41763,fd=19))
```

That answers three questions at once. Something is listening on 3000. It is a `node` process, PID 41763. And it bound `127.0.0.1` — loopback only.

Match PID to command and configuration:

```
ps -p 41763 -o pid,user,command
```

The command line often reveals the surprise: yesterday's dev server you forgot about, or a second copy of the same app started by a process manager that auto-restarts it.

Loopback vs. every interface

The bind address explains a whole family of "works here, refused there" bugs. Check whether it listens on loopback, one interface, or every interface:

```
127.0.0.1:3000   reachable only from the same machine
0.0.0.0:3000     reachable on every interface
```

A server bound to `127.0.0.1` inside a container or on a remote host answers curl locally and refuses everyone else. Nothing crashed — the listener and the client just disagree about which address to meet on.

If `ss` shows no listener at all, "connection refused" is fully explained: the service crashed, is still starting, or listens on a different port than the client was told. Go read that service's logs and its effective configuration.

Before you reach for kill

Do not kill an unknown process just to free a port. `EADDRINUSE` plus an unfamiliar PID means something is running that you do not yet understand, and it may be serving real traffic. Identify owner and service impact first, then stop it through its manager — `systemctl stop`, your process manager — rather than a bare `kill -9` that its supervisor will immediately undo anyway.

Verify dependencies and build output

Compare lockfile, runtime version, installed dependency tree, generated artifact, and source revision.

You fixed the bug, deployed, and the error is still there. Before doubting the fix, doubt the artifact: a local source fix does not help if production runs another artifact, runtime, or dependency resolution. This whole class of "but I fixed that!" bugs is an identity problem — the code you are reading is not the code that is running.

Record deploy identity

Capture the same four facts in every environment. Record deploy identity:

```
node --version
npm ls --depth=0
git rev-parse HEAD
shasum -a 256 dist/app.js
```

Each line closes one gap. `node --version` catches runtime drift: code using a newer API works on your Node 24 and throws on the server's Node 18. `npm ls --depth=0` shows the dependency tree as installed, not as declared — a missing lockfile can resolve different versions from the same `package.json` on different days. `git rev-parse HEAD` pins the source revision. The `shasum` of the built artifact is the ground truth: two machines with the same hash run the same bytes.

Compare, don't assume

Compare the working and failing environments line by line:

```
local:  v24.2.0  abc123f  dist/app.js sha256 9f2c...
prod:   v24.2.0  abc123f  dist/app.js sha256 4a71...
```

Same runtime, same commit, different artifact hash. Everything narrows to the build step: a stale build cache, a CI job that built another branch, or a deploy that copied an old `dist/`. Rebuild from a clean checkout and verify the deployed hash matches the tested artifact. If the hashes now agree and the bug persists, the fix itself is wrong — also useful to know.

The lockfile deserves a specific check: confirm it is committed, and that deployment installs with `npm ci` rather than `npm install`. `npm ci` installs exactly what the lockfile says and fails loudly when the lockfile and `package.json` disagree, which is exactly the noise you want.

Keep the variables still

Do not run broad dependency upgrades during diagnosis. An `npm update` mid-investigation changes dozens of variables at once, can erase the original evidence, and may fix the symptom while hiding the cause. Pin everything, find the bug, then upgrade deliberately with its own tests.

Check your understanding: debug the runtime

Check the commands, decisions, safety boundaries, and failure cases from debug the runtime.

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

Take this interactive quiz online

Debug production safely

Use timelines, observability, bounded mitigations, rollback, and an incident record to recover without destroying evidence.

Build a change timeline

Align symptom start with deployments, configuration, traffic, dependency, certificate, and infrastructure events.

Systems that worked yesterday and fail today were changed — by a deploy, a config edit, a certificate expiry, a dependency, or the traffic itself. Time correlation narrows the search without proving causation. Build the timeline from recorded events rather than memory, because memory reorders things to fit whatever story you already believe.

Assemble the timeline

Pull timestamps from systems that record them: deploy logs, monitoring dashboards, cron schedules, certificate validity dates. Create a UTC incident timeline:

```
10:02 deploy abc123 completed
10:04 error rate rose from 0.1% to 18%
10:05 database connections reached limit
10:07 rollback started
10:09 error rate returned to baseline
```

Add evidence links and the observer for each entry. "Error rate rose (dashboard link, seen by Ana)" is verifiable; "the site got slow around ten" is not.

Two minutes between deploy and error spike is suggestive. The rollback restoring baseline strengthens it: the change tracks the symptom in both directions. Still correlation — but correlation strong enough to direct where you look next, inside the diff of `abc123`.

Test whether the suspected change explains all affected and unaffected paths. If the deploy only touched image processing but checkout also degraded, either the connection limit links them, or your suspect is wrong. A cause that explains half the symptoms is usually half the story.

Bisect when the timeline is long

When the symptom appeared "sometime in the last two weeks" and fifty commits are candidates, stop reading diffs and bisect. `git bisect` runs a binary search over the commit history:

```
git bisect start
git bisect bad                 # the current commit fails
git bisect good v2.4.0         # this release was fine
# git checks out the midpoint; run your reproduction, then answer:
git bisect good   # or: git bisect bad
```

Each answer halves the range, so fifty commits take about six tests. At the end git names the exact commit that introduced the failure, collapsing your timeline to a single entry. The reproduction card from earlier in this course is what makes each test cheap and trustworthy.

Keep clocks synchronized and use one timezone in incident notes. A timeline mixing local time, UTC, and an unsynced server clock will convince you of an order of events that never happened.

Use metrics, logs, and traces together

Move from user impact to a failing component and one request without searching every signal blindly.

Production observability gives you three signal types, and each answers a different question. **Metrics** show scope and timing: how many, how slow, since when. **Traces** follow selected requests across boundaries: where one request spent its time. **Logs** explain discrete events inside those components: what exactly happened at that moment.

Debugging goes wrong when you start in the wrong one. Grepping all logs for "error" during an incident returns thousands of lines, most of them normal background failure. The efficient path runs in one direction: metrics to scope it, traces to localize it, logs to explain it.

Follow one path

Follow one path:

```
metric: checkout latency rose at 10:04
trace: waiting 8 seconds on inventory call
log: inventory pool timeout for request r-42
```

Each step shrinks the search. The metric says checkout, since 10:04 — not the whole system, not all day. The trace for one slow checkout request shows a waterfall of spans, and one span dominates: eight seconds waiting on the inventory service. Now you know the component and the dependency.

Only then open logs, scoped to the inventory service around 10:04, filtered by the request ID the trace handed you:

```
grep 'r-42' inventory.log
# {"event":"pool_timeout","requestId":"r-42","waitedMs":8000}
```

One log line, found in seconds, because two other signals aimed you at it.

Compare healthy and failing

Compare a healthy and failing request side by side: pull one trace from before 10:04 and one from after. The difference between the two waterfalls is the incident — 40 ms on inventory before, 8 seconds after. Everything the two traces share is background you can ignore.

Confirm the chosen signal represents user impact rather than background noise. A latency percentile on an endpoint nobody calls, or an error count dominated by one retrying bot, will have you debugging a non-incident while the real one continues.

Protect telemetry like the production data it describes. Traces and logs carry URLs, identifiers, and sometimes payloads — keep secrets and customer data out of them, and restrict access and retention.

Mitigate and roll back safely

Choose the smallest reversible action that reduces user harm while preserving evidence for diagnosis.

During an incident, recovery may matter before complete root cause. Users are failing now; the full diagnosis can take hours. **Mitigation** is the deliberate middle move: reduce harm first with the smallest reversible action, while keeping the evidence you will need to find the actual cause.

The menu is usually short: a feature flag, traffic shift, capacity limit, or rollback can contain impact. Prefer the option smallest in scope and easiest to undo. Disabling one feature beats rolling back a deploy that carried ten unrelated changes. A rollback beats restarting everything and hoping.

Write the gate before you act

Even under pressure — especially under pressure — write down what you are about to do and how you will know it worked. Write the mitigation gate:

```
action: disable new image processing
expected impact: uploads queue instead of failing
verification: error rate and queue depth
rollback of mitigation: re-enable after worker fix
evidence to preserve: failed job IDs and deploy logs
```

This takes ninety seconds and prevents the two classic incident mistakes: an action nobody can undo, and an action nobody verified. The `expected impact` line is a prediction. If you disable image processing and errors don't drop, your theory about the cause was wrong — and you learned that without wasting an hour.

Apply it first in the safest available scope — one region, a small traffic percentage, a canary — monitor the predicted signals, and stop if new harm appears. A mitigation that makes things worse and keeps running is a second incident.

Preserve the crime scene

The evidence line is the one people skip. Restarting a process clears its memory state. Rolling back removes the failing code path. Clearing a queue deletes the poisoned messages that would have explained everything. Before each action, capture what it will destroy: failed job IDs, a copy of current logs, the deploy identifier.

Do not delete logs, restart every system, or clear all queues before capturing evidence and understanding recovery consequences. Mass restarts create their own problems too — every service reconnecting at once can take down the database that was healthy.

Mitigation is temporary by definition. The gate's rollback line — "re-enable after worker fix" — needs an owner, or the workaround quietly becomes permanent and the feature stays off for a year.

Finish with a debugging record

Document symptom, scope, evidence, cause, repair, verification, cleanup, and prevention after the system recovers.

The bug is fixed and everything works. The temptation is to close the ticket and move on — which throws away the most valuable output of the whole investigation. A debugging record turns one repair into reusable operational knowledge: the next person hitting a similar symptom searches, finds your record, and skips the first three hours.

Write it while the evidence is fresh. A week later the details blur, and the record becomes fiction.

The template

Use a compact template:

```
symptom and user impact
start and end time
first failed boundary
evidence
root cause
mitigation and permanent repair
verification
follow-up owner and deadline
```

Keep it factual: every claim should point at evidence — a log line, a graph, a commit. And distinguish contributing conditions from root cause. "The pool was small" contributed; "the new image worker held a connection open for the length of each job" is the cause. Mixing the two produces fixes that only raise limits, then wait for the same failure at the new limit.

Separating **mitigation** from **permanent repair** matters for the same reason. "Rolled back at 10:07" restored service. "Connection now released before processing starts, commit def456" fixed the bug. If the record cannot name the permanent repair, the incident is not actually finished.

Test the record

Have someone unfamiliar with the incident reproduce the reasoning from the evidence. If they read it and ask "but how did you know it was the pool?", the evidence section has a hole. Remove speculative claims that were disproved along the way — half-remembered wrong hypotheses presented as fact are how folklore starts. "The database is flaky", forever, in every future incident.

The `follow-up owner and deadline` line keeps the record honest. Prevention items without an owner — "add pool metrics", "alert on queue depth" — are wishes, not follow-ups.

Do not use the review to blame individuals. A record that names the engineer who deployed teaches everyone to hide details next time, and the details are the whole point. Improve systems, checks, ownership, and recovery paths — the bad deploy was possible because nothing prevented it, and that is the fixable part.

Check your understanding: debug production safely

Check the commands, decisions, safety boundaries, and failure cases from debug production safely.

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/practical-debugging/.