API Security Course

Flavio Copes

Updated August 3, 2026

Contents

Protect APIs with complete inventories, object and function authorization, bounded inputs, abuse controls, verified integrations, and operational testing.

What you'll learn: Review and secure an invoices API across authorization, resource use, credentials, third-party calls, monitoring, and incident response.

Map the API

Inventory assets, actors, versions, contracts, and trust boundaries.

Identify API assets and actors

List the data, operations, identities, integrations, and business flows an API exposes before choosing security controls.

An API exposes application capabilities directly. Before you pick a single security control, list what callers can read, change, trigger, or spend. That list is the map every later decision depends on.

List the actors

Include anonymous callers, users, administrators, mobile apps, background jobs, partners, and third-party services. Each one authenticates differently and deserves different limits.

Here is a starting point for an invoices API:

```
actors:
  - name: customer
    auth: session cookie or OAuth token
    operations: [list own invoices, download own PDF]
  - name: support-staff
    auth: SSO with staff role
    operations: [search invoices, issue refunds]
  - name: accounting-job
    auth: scoped API key, runs nightly
    operations: [export all invoices]
  - name: payment-provider
    auth: signed webhook
    operations: [mark invoice paid]
```

For each actor, record the intended operations and sensitive data. This becomes the authorization and testing map.

Attach impact to each operation

Not every operation carries the same risk. Record data sensitivity and maximum loss beside each operation:

```
GET  /invoices/:id          customer data       loss: one record leaked
POST /invoices/:id/refund   money movement      loss: unbounded without a cap
GET  /exports/invoices      all customer data   loss: the full dataset
```

That evidence separates a harmless invoice search from a refund path that needs stronger approval, logging, and recovery controls. A support account that can issue refunds turns a useful integration into a money-moving security boundary.

Test the map with a compromised actor

A good map answers a concrete question: what happens if this actor is compromised? Pick the support role and walk the table with its real credentials:

```
curl -i -X POST https://api.example.com/invoices/inv_00193/refund \
```

```
  -H "Authorization: Bearer $SUPPORT_TOKEN" \
  -d '{"amount": 490000}'
# HTTP/1.1 403 Forbidden - refunds above the per-day cap need approval
```

If the honest answer is "a stolen support token can refund every invoice, with no cap and no alert", the map found your first real risk before you wrote any code. That is the point of doing this first.

Revisit the table when a new integration lands. Most scope creep arrives one "quick partner endpoint" at a time, and the map only helps while it matches what production serves.

Create an actor-to-operation table for the invoices API. Prove it is useful by adding a compromised support account and recording which reads, writes, and refunds must fail.

Keep an API inventory

Track deployed hosts, routes, methods, versions, owners, data classes, and retirement dates so forgotten endpoints do not remain exposed.

You cannot protect an endpoint nobody remembers. Old API versions often keep old weaknesses alive, and attackers hunt for them because they know teams forget.

Build the inventory from more than one source

Documentation drifts. Build the inventory from source, gateway, deployment, and observed traffic, then compare them against each other.

```
# routes the code declares
grep -rE "router\.(get|post|put|patch|delete)" src/routes/ | sort -u

# routes production actually serves
awk '{print $6, $7}' /var/log/nginx/access.log | sort | uniq -c | sort -rn | head -20
```

Observed traffic is stronger evidence than documentation alone. A mobile client may still call `/v1/invoices` months after the web app moved to `/v2`. Removing the route too early breaks customers, but leaving it ownerless preserves old authorization bugs.

Record ownership and lifecycle

Give each API an owner and lifecycle state. A useful inventory row looks like this:

```
host: api.example.com
route: GET /v1/invoices
owner: billing-team
state: deprecated
deployment: invoices-v1 (build 2026-05-02)
last-seen: 2026-07-29T14:22:10Z
retirement: 2026-10-01
```

Include the deployment identifier and last-seen timestamp so the owner can distinguish a live route from stale gateway configuration. Remove unused routes and credentials instead of leaving "temporary" compatibility forever.

Shadow and zombie APIs are the two failure modes this catches: routes that shipped without ever entering a review, and routes that outlived their owners. The inventory exists to make both visible before an attacker does.

Verify the inventory catches drift

An inventory earns trust when it flags a route you did not expect:

```
curl -i https://api.example.com/v1/debug/config
# HTTP/1.1 404 Not Found  <- good, it was removed
# HTTP/1.1 200 OK         <- bad: undocumented, no owner, still serving
```

A common failure mode: the gateway routes `/v1/*` to an old deployment as a catch-all, so routes deleted from the code still resolve in production. Comparing gateway rules against source routes is the only way to catch that, because neither source alone shows the mismatch.

Compare source routes, gateway routes, and access logs for one environment. Save the mismatch list, then request one undocumented route and prove it is removed or has an owner and retirement date.

Define request and response contracts

Specify allowed methods, parameters, bodies, content types, responses, and errors so unknown input and accidental data exposure are visible.

A clear contract gives both clients and the server a boundary. Open-ended objects create room for accidental authority: any field the server accepts is a field an attacker will try.

Specify the request side

Describe required fields, limits, formats, allowed values, and response shapes. Reject unsupported methods and content types. Here is a request schema for creating an invoice:

```
{
  "type": "object",
  "required": ["customerId", "items"],
  "additionalProperties": false,
  "properties": {
    "customerId": { "type": "string", "pattern": "^cus_[a-z0-9]{12}$" },
    "items": { "type": "array", "minItems": 1, "maxItems": 100 },
    "notes": { "type": "string", "maxLength": 500 }
  }
}
```

`additionalProperties: false` is doing real security work here. Any field outside the contract gets rejected instead of silently flowing into your database layer.

Publish the schema — OpenAPI works well — so clients, server validation, and contract tests all share one source of truth instead of three diverging opinions.

Verify the rejection actually happens:

```
curl -i -X POST https://api.example.com/invoices \
  -H "Content-Type: application/json" \
  -d '{"customerId":"cus_ab12cd34ef56","items":[{"sku":"PLAN-PRO"}],"isPaid":true}'
# HTTP/1.1 400 Bad Request
# {"error":"unknown field: isPaid"}
```

Select response fields deliberately

Select response fields deliberately rather than serializing database models. An invoice response may accidentally expose `internalNote` when code serializes the database row. A strict response schema catches that leak without forcing every internal field into the public contract.

```
function invoiceResponse(row) {
  return {
    id: row.id,
    customerId: row.customer_id,
    total: row.total,
    status: row.status,
  }
}
```

New columns added by a migration never reach clients until someone adds them here on purpose.

Test the whole response

Contract tests should inspect the complete response, not only required fields. Assert that the set of returned keys equals the approved set, not just that the required ones exist.

This catches new database columns that appear after a migration and were never approved for API clients. The test fails the day the column ships, not the day a customer notices the leak in your JSON.

Write request and response schemas for creating one invoice. Add an unknown input field and an internal database field, then show both are rejected or absent from the response.

Draw API trust boundaries

Trace identity, data, and authority across clients, gateways, services, queues, databases, caches, and third-party APIs.

Internal traffic is not automatically trusted. A compromised service or leaked worker token is still an attacker path, and ”it sits behind the load balancer” is not an authorization policy.

Trace one request end to end

Follow an invoice request through every hop and mark three things: where authentication changes form, where authorization decisions happen, and where data leaves your control.

```
browser --TLS--> gateway --mTLS--> invoices-service ----> postgres
  |                 |                     |
cookie        verifies JWT,       trusts X-User-Id
              adds X-User-Id      from the gateway only
```

Record ambient credentials added by platforms and proxies along the way. Validate every boundary according to its own threat model.

Queues and caches are boundaries too. A message consumed by a worker and a cached response served to another user both carry authority across the diagram, and each hop deserves the same question: who proved this identity?

The forged header problem

A gateway may add `X-User-Id` after authentication. If the application also accepts that header directly from the internet, a caller can choose another identity:

```
# bypassing the gateway, talking to the service directly
curl -i http://invoices-service.internal:8080/invoices \
  -H "X-User-Id: 42"
# HTTP/1.1 401 Unauthorized   <- what you want
# HTTP/1.1 200 OK             <- identity spoofing, one header away
```

The fix is structural. The service accepts identity headers only over the mutually authenticated connection from the gateway, and rejects them on any other path. Network reachability alone must never imply identity.

Document header ownership

Document which component may create, replace, or remove every security header. A short table is enough:

```
X-User-Id         created by gateway, stripped if present on ingress
X-Forwarded-For   appended by gateway, never trusted from the client
Authorization     verified by gateway, not forwarded downstream
```

Then the direct-service test can prove the application rejects identity claims outside that trusted network path. Without the table, nobody can say whether a header arriving at the service is a platform feature or an attack in progress.

Trace one invoice request from client to database and label who creates each identity value. Bypass the gateway in a test and prove a forged identity header cannot authorize the request.

Check your understanding: map the api

Check API assets, inventory, contracts, trust boundaries, versions, and attack surface.

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

Take this interactive quiz online

Authorization

Protect objects, properties, functions, tenants, and bulk actions.

Enforce object-level authorization

Check the caller's relationship to every requested object instead of accepting an object identifier as proof of access.

An object ID tells the server what to load. It does not tell the server who may load it. Broken object level authorization — **BOLA**, also known as IDOR — sits at the top of the OWASP API Security Top 10 because it keeps working: change one identifier in a URL, read someone else's data.

Put ownership in the query

Resolve identity from trusted authentication state, then constrain the query by owner, tenant, membership, or policy:

```
SELECT id, total
FROM invoices
WHERE id = ? AND account_id = ?
```

The `account_id` value comes from the session, never from the request. If the row belongs to someone else, the query returns nothing and the API returns a plain `404`.

Customer 42 can be signed in correctly and still request customer 43's invoice. Random-looking IDs reduce guessing, but they never replace the ownership condition in the query. UUIDs leak through logs, referrer headers, and support tickets.

Test with two valid identities

Verification needs two real accounts, both authenticated, each trying to reach the other's data:

```
# customer 42's token requesting customer 43's invoice
curl -i https://api.example.com/invoices/inv_9f3c2a \
  -H "Authorization: Bearer $TOKEN_CUSTOMER_42"
# HTTP/1.1 404 Not Found   <- the ownership check is working
```

Apply the same rule to read, update, delete, export, and nested routes. A team that protects `GET /invoices/:id` but forgets the `DELETE` handler has only decorated the problem.

Check indirect paths

Check indirect paths too. An attachment, comment, or export nested below an authorized invoice can still load a foreign child object unless every relationship stays constrained:

```
curl -i https://api.example.com/invoices/inv_own1/attachments/att_foreign \
  -H "Authorization: Bearer $TOKEN_CUSTOMER_42"
# must be 404: the attachment belongs to a different invoice
```

The classic mistake: the handler authorizes the parent invoice, then loads the attachment by its ID alone. The child query needs its own relationship condition, joining back to the authorized parent.

Create invoices for two valid accounts and exercise read, update, delete, and export. Save the response and database state proving every cross-account attempt failed without changing the target.

Protect object properties

Allowlist fields callers may read or change so mass assignment and excessive data exposure cannot cross property-level permissions.

A user may access an object without being allowed to see or edit every property on it. Object-level authorization answers ”may you touch this invoice?”. Property-level authorization answers a second question: ”which fields of it?”.

Allowlist input fields

Parse request bodies into purpose-specific input types. Do not spread arbitrary JSON into a database update:

```
// mass assignment: everything in the body reaches the database
await db.invoices.update(id, req.body)

// allowlist: only fields this operation may change
const { label } = req.body
await db.invoices.update(id, { label })
```

The first version is a **mass assignment** vulnerability. A customer may change an invoice label but not `isPaid` or `accountId` — with the spread version, one crafted request flips both.

```
curl -i -X PATCH https://api.example.com/invoices/inv_9f3c2a \
  -H "Authorization: Bearer $CUSTOMER_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"label":"office chairs","isPaid":true,"accountId":"acc_admin"}'
# HTTP/1.1 400 Bad Request
# {"error":"fields not allowed: isPaid, accountId"}
```

Silently ignoring forbidden fields can preserve compatibility, while rejecting them makes client mistakes easier to detect. Choose one behavior and document it in the contract.

Allowlist output fields

The read side leaks the same way. Build response objects from approved fields and apply role-specific views where necessary. A customer view of an invoice never includes `internalNote`. A support view might, because support staff carry a different permission set over the same object.

ORMs make both leaks easy: `toJSON()` on a model serializes every column, and generated API clients happily send every field the schema exposes.

Verify the stored row, not the status code

The important evidence is the stored row and returned object, because a successful status alone can hide a partial unauthorized update:

```
SELECT label, is_paid, account_id
FROM invoices WHERE id = 'inv_9f3c2a';
-- label changed; is_paid and account_id untouched
```

Run this check after the malicious PATCH above. A `400` response paired with a flipped `is_paid` column means validation ran after the write — a bug the status code alone would never show you.

Send an update containing one allowed field and two protected fields. Prove the allowed value changed, the protected values did not, and the response never reveals `internalNote`.

Enforce function-level authorization

Protect administrative, export, billing, and maintenance operations with explicit server-side permissions on every route and method.

Changing a path or HTTP method must not turn an ordinary user into an administrator. Object-level checks ask ”whose invoice is this?”. **Function-level authorization** asks a different question: may this role call this operation at all?

Deny by default

Centralize policy where it stays visible, but verify coverage route by route. Every handler declares what it requires, and anything undeclared is denied:

```
app.post('/admin/refunds', requirePermission('refunds:create'), issueRefund)
app.get('/admin/exports', requirePermission('exports:read'), listExports)

// a route without requirePermission gets rejected by the
// outer middleware instead of running unprotected
```

The UI hides `/admin/refunds`, but an ordinary user can still send the request. Hiding a button is a UX decision, not an access control.

```
curl -i -X POST https://api.example.com/admin/refunds \
  -H "Authorization: Bearer $REGULAR_USER_TOKEN" \
  -d '{"invoiceId":"inv_9f3c2a","amount":4900}'
# HTTP/1.1 403 Forbidden
```

Methods and versions escape policies

A new `POST` route can also escape a policy that only covered the older `DELETE` method. Test sibling endpoints, alternate methods, batch actions, and older API versions with a low-privilege identity.

Include route aliases and versioned endpoints in the matrix. `/api/v1/admin/refunds` and `/admin/refunds` need the same rule, and a batch endpoint that wraps refunds needs it too.

Watch for the inverse bug as well: a policy applied to everything under `/admin/*` misses an administrative action mounted elsewhere, like `POST /invoices/:id/force-close`. Route prefixes are a convention, not a security boundary.

Prove coverage instead of assuming it

Central policy helps, but a route-level coverage test proves each handler actually invokes the decision before starting work:

```
for (const route of adminRoutes) {
  const res = await request(app)[route.method](route.path)
    .set('Authorization', `Bearer ${lowPrivilegeToken}`)
  assert.equal(res.status, 403, `${route.method} ${route.path} is unprotected`)
}
```

This loop fails the build the day someone adds an admin route and forgets the permission check. That is a much cheaper discovery than a bug bounty report.

Build a role-by-method matrix for refund, export, and maintenance routes. Run every denied cell with a low-privilege token and save evidence that no side effect or audit gap remains.

Isolate tenants and bulk actions

Carry trusted tenant context through queries, caches, jobs, and batch operations so one organization cannot affect another.

Multi-tenant software repeats one authorization boundary everywhere. One missing tenant condition can expose many records at once, which is why tenancy bugs produce the worst incident reports.

Carry tenant context everywhere

Derive tenant context from trusted membership, not request data alone. Then include it in storage queries, cache keys, object paths, queues, and logs.

The cache is the classic gap. A cache key such as `invoice:193` can leak the same record even when the database query is correct:

```
// wrong: tenant-blind key, first tenant to cache wins
const cached = await cache.get(`invoice:${id}`)

// right: tenant is part of the key
const cached = await cache.get(`tenant:${tenantId}:invoice:${id}`)
```

The same applies to object storage paths: `exports/acme/2026-08.csv`, never a shared `exports/2026-08.csv` that the next tenant's download link can reach.

Bulk operations check every item

Apply per-object authorization inside bulk operations instead of checking only the first item. A bulk export may check the first invoice, then load the remaining IDs without tenant filters:

```
-- every ID is constrained, not just validated up front
SELECT id, total FROM invoices
WHERE id = ANY(?) AND tenant_id = ?
```

If the query returns fewer rows than the IDs requested, something in the batch was foreign. Decide whether that is a silent skip or a hard failure, and log the event either way.

Background jobs inherit the boundary

Background jobs must carry immutable tenant context from creation through delivery. Letting a worker reconstruct it from user-controlled job data moves the authorization weakness into another process:

```
{ "job": "export-invoices", "tenantId": "acme", "requestedBy": "usr_8817" }
```

The worker uses `tenantId` from the job record the API wrote at enqueue time, never from anything the client sent alongside it.

Mix valid and foreign invoice IDs in one bulk export. Prove the job, cache, stored file, and download path expose no foreign record, including when the foreign ID appears first.

Check your understanding: authorization

Check object, property, function, tenant, and bulk-operation authorization.

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

Take this interactive quiz online

Input and resource controls

Validate shapes, limit expensive work, protect business flows, and return safe errors.

Validate every request shape

Parse parameters, headers, bodies, and content types into bounded application values before business logic uses them.

JSON syntax only proves the body is JSON. It says nothing about the values your operation can safely accept. A valid JSON body can contain 50,000 line items or a negative total.

Validate structure and bounds

Validate types, lengths, ranges, formats, nesting depth, array size, and allowed fields. A schema library keeps the rules declarative and testable:

```
import { z } from 'zod'

const createInvoice = z.object({
  customerId: z.string().regex(/^cus_[a-z0-9]{12}$/),
  items: z.array(z.object({
    sku: z.string().max(64),
    quantity: z.number().int().min(1).max(999),
  })).min(1).max(100),
  total: z.number().positive().max(1_000_000),
}).strict()
```

`.strict()` rejects unknown fields. Rejecting unknown fields is safer for authority-bearing input, but tolerant readers may be useful for harmless provider additions.

Reject unsupported content types and ambiguous duplicates too — two `total` fields in one body, or a `text/plain` request your framework helpfully parses as JSON anyway.

Verify rejection at the boundary

Test the exact limits, then one step beyond each:

```
curl -i -X POST https://api.example.com/invoices \
  -H "Content-Type: application/json" \
  -d '{"customerId":"cus_ab12cd34ef56","items":[],"total":-500}'
# HTTP/1.1 400 Bad Request
# {"error":"items: at least 1 entry; total: must be positive"}
```

Then confirm no invoice row was created. A `400` that still wrote a partial record is worse than no validation at all, because it looks safe from the outside.

Validate again off the HTTP path

Apply validation again when a message arrives from a queue or third party. Also verify the queue consumer, because an internal producer or old message may bypass the current HTTP schema entirely. Messages enqueued last year under an older schema will replay against today's assumptions.

Validate after decoding but before costly business work begins. Parsing 50,000 line items just to reject them late still costs you the parse, so put array-size limits as early as the body parser allows.

Record the accepted invoice schema and its numeric, size, and nesting limits. Test the exact boundaries plus one value beyond each limit, and prove rejected input creates no invoice.

Limit resource consumption

Bound request size, execution time, pagination, uploads, concurrency, and expensive downstream work before one caller can exhaust shared capacity.

An API can be valid and still be too expensive. Attackers look for operations where a small request creates large work — and so do badly written client retry loops, which cause most real capacity incidents.

Bound the obvious dimensions

Set limits at the edge and inside the application. Cap page size, query complexity, file size, processing time, retries, and concurrent jobs.

Pagination is where most APIs leak capacity first:

```
curl -i "https://api.example.com/invoices?per_page=100000"
# HTTP/1.1 400 Bad Request
# {"error":"per_page must be between 1 and 100"}
```

Clamping silently to 100 also works. Rejecting makes the limit visible to client developers. Either way, the database never sees an unbounded query.

Rate limiting handles frequency. Return a controlled response and observe repeated limit hits:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 30
RateLimit-Remaining: 0
```

A caller who hits the limit constantly is either broken or probing. Both deserve a look.

Price operations by their real cost

A 2 KB report request can trigger a minute-long database scan and a 200 MB export. One global rate limit misses this cost difference and may punish cheap requests.

Measure the expensive unit directly: rows scanned, worker seconds, memory, or provider calls. Per-operation budgets make the limit explainable and let ordinary invoice reads keep working:

```
GET  /invoices          600 requests/min per account
POST /reports           5 requests/min, max 2 concurrent per account
POST /invoices/import   1 running job per account
```

Verify the failure is controlled

Exceeding a limit must produce a clean rejection, not a half-finished job that already burned the resources:

```
# third concurrent report for the same account
curl -i -X POST https://api.example.com/reports \
  -H "Authorization: Bearer $TOKEN"
# HTTP/1.1 429 Too Many Requests
```

Check the worker queue after the `429`: no orphaned job should exist for the rejected request.

Measure database time, response bytes, and queued work for the report endpoint. Exceed one page, timeout, and concurrency limit, then prove the API fails predictably while a normal request still

succeeds.

Protect sensitive business flows

Defend high-value actions such as purchasing, invitations, reservations, and password reset from automation and logic abuse.

A perfectly valid request can still abuse the product. Buying all limited stock through automation is a business-flow attack: every individual request passes every technical check, and the damage lives in the aggregate.

Find the flows worth attacking

Identify scarce, valuable, or irreversible actions: purchasing limited inventory, sending invitations, reserving slots, triggering password reset emails, claiming referral rewards. For each one, ask what happens when a scripted client runs it a thousand times an hour.

An invitation endpoint can pass every schema check while sending 10,000 emails. The attacker pays nothing. You pay the provider bill and lose sending reputation for your whole domain.

Layer limits on the flow, not just the endpoint

Add per-identity and contextual limits, idempotency, confirmation, monitoring, and manual review where impact warrants it:

```
invitations:
  per-account:      50 / day
  per-destination:  3 / week    # the same inbox cannot be flooded
  per-signup-ip:    5 / hour    # fresh accounts do not reset limits
```

The per-destination limit matters more than it looks. Creating ten accounts defeats a per-account cap, but the target address stays the same across all of them.

A CAPTCHA alone may slow bots, but it does not cap cost after a valid session is automated. Avoid defenses that only slow honest users.

Verify the cap holds:

```
for i in $(seq 1 60); do
  curl -s -o /dev/null -w "%{http_code}\n" \
    -X POST https://api.example.com/invitations \
    -H "Authorization: Bearer $TOKEN" \
    -d '{"email":"person'$i'@example.com"}'
done
# 201 for the first 50, then 429 - and exactly 50 emails delivered
```

Count the actual deliveries in the email provider's log. A limiter that returns `429` after the send already happened protects nothing.

Leave a recovery path

Limits need a recovery path for legitimate spikes. A real customer launching a company-wide rollout will hit the invitation cap on day one. Record who can approve a larger campaign, how long the exception lasts, and which signal reveals continued abuse.

Set per-account and per-destination limits for invitations and record the expected email count. Replay requests across several sessions and prove delivery stops, an event is logged, and normal recovery remains possible.

Return safe errors

Use consistent status codes and problem details without exposing stack traces, queries, secrets, internal hosts, or authorization-sensitive existence.

API errors are part of the contract. They should guide the client without teaching an attacker about internals. Every stack trace, SQL fragment, or internal hostname in a response is free reconnaissance.

Use a stable, safe shape

Return a stable type, title, status, safe detail, and request ID. The problem details format from RFC 9457 gives you a conventional structure:

```
{
  "type": "https://api.example.com/errors/validation",
  "title": "Invalid request body",
  "status": 400,
  "detail": "items must contain between 1 and 100 entries",
  "requestId": "req_7f3ab910"
}
```

Keep private diagnostics in protected logs. A request ID connects the safe response to detailed evidence without copying sensitive diagnostics outside:

```
# internal log line, never in the response
req_7f3ab910 POST /invoices 400 ValidationError items.length=50000 user=usr_8817
```

Keep error identifiers stable enough for clients, but do not expose internal exception classes. A type of `validation` survives a refactor. `PostgresUniqueViolationError` does not, and it names your database for free.

Don't confirm what the caller may not know

Returning `invoice 193 belongs to another account` confirms the record exists. That is an enumeration oracle. Decide deliberately when authorization failures should be indistinguishable from missing resources:

```
curl -i https://api.example.com/invoices/inv_someone_elses \
  -H "Authorization: Bearer $TOKEN"
# HTTP/1.1 404 Not Found   <- same response as a truly missing invoice
```

Returning the same public result for missing and forbidden records hides that fact, but protected logs still need the real cause. The security team must be able to distinguish a typo from a probing campaign.

Verify every error path

Trigger each class of failure on purpose and inspect what leaks. A common miss: your handlers return clean problem details, but the framework's default 500 handler still prints a stack trace when

something throws outside your code. Send a request that breaks early — an unparseable body, an oversized header — and read the raw response.

Capture public and internal errors for missing, forbidden, invalid, limited, and failed requests. Prove public responses contain stable codes and request IDs while stack traces and object existence stay private.

Check your understanding: input and resource controls

Check schemas, output selection, limits, business-flow abuse, pagination, and safe failure.

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

Take this interactive quiz online

Credentials and integrations

Scope credentials, prevent replay, verify webhooks, and distrust upstream data.

Scope API credentials

Give API keys and tokens a narrow audience, permission set, lifetime, owner, storage location, and revocation path.

An API key is a bearer credential unless the protocol adds stronger proof. Anyone who obtains it may use its authority. The server cannot tell your nightly job from an attacker holding the same string.

The common credential types differ mostly in lifetime and where authority lives. A static **API key** is easy to issue and lives until you revoke it. An **OAuth access token** is short-lived and scoped by the authorization server that issued it. A **JWT** carries its claims — audience, scopes, expiry — inside the token, verifiable without a lookup. Whatever you use, the scoping discipline below is the same.

One credential per job

Create separate credentials per environment and integration. A shared production key used by billing and analytics makes revocation an outage for both. Separate credentials add inventory work, but they make ownership and blast radius visible.

```
key_billing_prod_a91x    owner: billing-team  scope: invoices:read charges:create
key_analytics_prod_covk  owner: data-team     scope: invoices:read
key_billing_stg_m2p      owner: billing-team  scope: invoices:read charges:create
```

Scope permissions to what the integration actually does. Avoid URLs and browser code — a key in a query string ends up in access logs, proxies, and browser history. Store only where needed.

Log the identifier, never the secret

Log an identifier rather than the secret. Usage evidence should identify the credential without recording its value:

```
2026-08-03T09:12:44Z key=key_analytics_prod_covk route="GET /invoices" status=200
```

A last-used timestamp and expected source help responders distinguish an abandoned key from one still serving production.

Make revocation routine

Make rotation and revocation routine, not an emergency procedure you run for the first time during an incident:

```
# after revoking the analytics key
curl -i https://api.example.com/invoices \
  -H "Authorization: Bearer $ANALYTICS_KEY"
# HTTP/1.1 401 Unauthorized

curl -i https://api.example.com/invoices \
  -H "Authorization: Bearer $BILLING_KEY"
# HTTP/1.1 200 OK   <- billing unaffected, blast radius contained
```

The second request is the real test. Revoking a credential is easy; proving the revocation touched nothing else is what scoping buys you.

Replace one shared key with two scoped credentials and record their owners and permissions. Revoke the analytics key, then prove billing still works and analytics receives a denied response.

Prevent replay and duplicate work

Use expirations, nonces, idempotency keys, and stored operation results where repeated requests could charge, send, or mutate twice.

Networks retry. Attackers replay. A sensitive operation needs a clear answer to "what if this exact request arrives again?" — because it will, whether from a flaky mobile connection or a captured request resent on purpose.

Idempotency keys for client retries

Use idempotency keys tied to the caller and operation, store the result for a bounded period, and reject conflicting reuse:

```
curl -X POST https://api.example.com/payments \
  -H "Authorization: Bearer $TOKEN" \
  -H "Idempotency-Key: pay-order-8912" \
  -d '{"invoiceId":"inv_9f3c2a","amount":4900}'
```

A client times out after payment succeeds and retries the same request. Returning the stored result prevents a second charge, while reusing the key with a different amount must fail:

```
# same key, different body
curl -i -X POST https://api.example.com/payments \
  -H "Authorization: Bearer $TOKEN" \
  -H "Idempotency-Key: pay-order-8912" \
  -d '{"invoiceId":"inv_9f3c2a","amount":9900}'
# HTTP/1.1 422 Unprocessable Entity
# {"error":"idempotency key reused with a different request body"}
```

Make the store atomic

Store the key and result atomically with the operation. Otherwise two workers can both see an unused key and perform the same side effect before either stores evidence:

```
-- the unique constraint is the actual lock
INSERT INTO idempotency_keys (key, account_id, request_hash)
VALUES ('pay-order-8912', 'acc_42', 'sha256:9d1e44…')
ON CONFLICT (key, account_id) DO NOTHING;
-- zero rows inserted -> another worker owns this key,
-- return its stored result instead of charging again
```

Checking with a `SELECT` first and inserting afterwards is the race condition, not the fix. Design database changes atomically so a rejected duplicate leaves nothing half-done.

Give stored keys a bounded lifetime, say 24 hours. That covers realistic client retries, keeps the table small, and prevents an old key from resurrecting a stale result months later.

Replay windows for signed requests

For signed requests, include a timestamp or nonce and enforce a replay window. A captured request older than a few minutes gets rejected even with a valid signature, and a stored nonce blocks the same request inside the window.

Send two concurrent payment requests with the same idempotency key and body. Prove one charge exists, both callers receive the same result, and conflicting key reuse is rejected.

Verify webhooks

Authenticate webhook bytes, validate freshness, prevent replay, acknowledge safely, and process events idempotently.

A webhook endpoint is public by design. The payload must prove which provider sent it, because anyone who discovers the URL can POST convincing-looking JSON at it.

Verify the signature over the raw bytes

Verify the signature over the exact raw body using the provider's current scheme and a protected secret or public key. Most providers send an HMAC in a header:

```
import { createHmac, timingSafeEqual } from 'node:crypto'

function verifyWebhook(rawBody, signatureHeader, secret) {
  const expected = createHmac('sha256', secret).update(rawBody).digest()
  const received = Buffer.from(signatureHeader, 'hex')
  return expected.length === received.length &&
    timingSafeEqual(expected, received)
}
```

The **raw body** detail is where implementations break. If your framework parses the JSON first and you re-serialize it, key order or whitespace changes and every signature fails — or you end up verifying bytes the provider never signed. Capture the body before any parser touches it.

`timingSafeEqual` matters too: a byte-by-byte comparison that returns early leaks the correct signature through response timing.

Freshness and replay

Check timestamp tolerance, store event IDs, and make processing idempotent. A provider can deliver the same event twice or retry after your response times out. Signature verification proves origin and integrity, but only stored event IDs prevent duplicate fulfillment:

```
INSERT INTO webhook_events (event_id) VALUES ('evt_1PqR8s')
ON CONFLICT DO NOTHING;
-- zero rows -> already processed: acknowledge and stop
```

Acknowledge, then work

Acknowledge only after durable acceptance, then process asynchronously when work is slow. This reduces provider retries without pretending an in-memory queue can survive a process crash.

Then test the paths that must fail:

```
# unsigned payload aimed straight at the endpoint
curl -i -X POST https://api.example.com/webhooks/payments \
  -H "Content-Type: application/json" \
  -d '{"type":"payment.succeeded","id":"evt_fake"}'
# HTTP/1.1 401 Unauthorized - and no order was fulfilled
```

Do not log signatures or complete sensitive payloads. The webhook log needs the event ID and outcome, not the customer data inside the event.

Save results for valid, changed, expired, duplicate, unsigned, and wrong-key payloads. Prove only one side effect occurs and a duplicate delivery returns a safe success response.

Treat third-party API data as untrusted

Validate upstream responses, constrain redirects and destinations, set timeouts and size limits, and avoid inheriting a provider's compromise.

A trusted provider can fail, change, or become compromised. Its response still crosses a trust boundary. TLS authenticates the connection, not the schema or business meaning of its response.

Validate the response like user input

Validate response status, content type, schema, size, and meaning. A tax provider may return HTML during an outage or redirect to a private address after compromise:

```
const res = await fetch('https://tax-api.example.net/v2/rates?country=IT', {
  redirect: 'error',              // an unexpected redirect is a failure
  signal: AbortSignal.timeout(3000),
})

if (!res.ok) throw new Error(`tax api returned ${res.status}`)
const contentType = res.headers.get('content-type') ?? ''
if (!contentType.includes('application/json')) {
  throw new Error('tax api returned non-JSON')  // the outage-page case
}
```

Then parse the body into a bounded schema, exactly as you would a client request. A tax rate of `-2` or `9999` should fail validation, not flow into invoice totals because "the provider is trusted".

Cap the size before reading everything into memory:

```
const length = res.headers.get('content-length')
if (length && Number(length) > 100_000) {
  throw new Error('tax api response too large')
}
```

For chunked responses without a length header, enforce the same cap while streaming.

Keep upstream text out of interpreters

Do not pass upstream text into SQL, HTML, shell commands, or AI tools without the same controls as user input. A provider's "company name" field rendered unescaped into your dashboard is stored XSS with an extra hop. Parameterize and escape regardless of origin — a paid API contract changes nothing about the bytes.

Bound the failure, not just the request

Set short timeouts and bounded retries. Bound retries as well as individual requests: five automatic retries can multiply an outage into resource exhaustion, while a controlled fallback keeps the failure visible to operators.

The realistic failure mode is quiet. The provider starts responding slowly, your retries stack up, and your own API exhausts its worker pool serving requests that were never going to succeed.

Stub normal, oversized, slow, redirected, malformed, and hostile provider responses. Record the timeout and size evidence, then prove no bad response reaches storage, HTML, SQL, or shell execution.

Check your understanding: credentials and integrations

Check API credentials, replay, idempotency, webhooks, outbound APIs, SSRF, and third-party trust.

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

Take this interactive quiz online

Test and operate APIs

Test authorization, harden configuration, monitor abuse, and retire versions safely.

Write an API security matrix

Test every route across identity, role, object relationship, input boundary, rate limit, and expected security event.

API security tests need more than one happy-path token. Build a matrix that changes one security condition at a time, so a failing cell points at exactly one broken control.

Choose the rows

Include anonymous, expired, revoked, low-privilege, wrong-tenant, and administrative identities. Test another user's object, hidden properties, alternate methods, batch requests, oversized input,

and repeated calls.

For one high-value route, the matrix reads like this:

```
route: POST /invoices/:id/refund
  anonymous              -> 401, no refund row
  expired token          -> 401, no refund row
  wrong tenant           -> 404, no refund row
  low-privilege role     -> 403, no refund row, denial event logged
  admin, valid           -> 200, one refund row, audit event logged
  admin, key replayed    -> 200 same body, still exactly one refund row
```

Each row is a real request with real credentials against a real database, not a mocked unit test. Authorization bugs live in the wiring between middleware, handler, and query — a mock skips exactly the layer you are trying to test.

Assert state, not just status

A `403` alone does not prove safety if the invoice changed before the response. The matrix must check identity, outcome, stored state, and the expected security event. Verify stored state and events, not only status codes:

```
const res = await api.post('/invoices/inv_9f3c2a/refund', body, wrongTenantToken)
assert.equal(res.status, 404)

const refunds = await db.query(
  'SELECT count(*) FROM refunds WHERE invoice_id = $1', ['inv_9f3c2a'])
assert.equal(refunds.rows[0].count, '0')   // no side effect happened

const events = await db.query(
  "SELECT count(*) FROM audit_events WHERE type = 'authz_denied'")
assert.equal(events.rows[0].count, '1')    // the denial left evidence
```

Prioritize by harm

Prioritize cells by potential harm instead of automating every permutation first. Cross-tenant writes and money movement deserve stronger evidence than a harmless malformed optional field. Forty high-value cells that run on every commit beat four thousand cells nobody maintains.

One extra step keeps the matrix honest: break a policy on purpose. Comment out one permission check locally and confirm the matching cell fails. A matrix that stays green through a real regression is measuring nothing.

Automate the highest-impact rows for two users, two tenants, an expired token, and an oversized body. Save assertions for response, database state, and audit event, then make one policy fail and prove the test catches it.

Harden API configuration

Disable unused methods and features, restrict content types and CORS, protect documentation, and keep production errors and credentials controlled.

Security misconfiguration can expose a sound codebase. Production deserves its own review, because the most dangerous defaults are development conveniences nobody remembered to turn off.

Inventory every layer

Inventory gateway, proxy, runtime, framework, storage, and cloud settings. Remove default accounts and sample routes. Keep diagnostics and internal schemas behind the access policy their content requires.

Default accounts and sample routes deserve a dedicated pass. They ship enabled, they are documented publicly, and scanners probe for them by name within hours of a deployment going live.

Development may expose stack traces, wildcard CORS, and interactive API docs. Copying those defaults into production can bypass careful application code: the stack trace names your framework and file paths, permissive CORS lets hostile pages read API responses, and a public Swagger UI hands over a complete map of the attack surface.

Probe the effective behavior

Inspect effective runtime behavior because environment variables, gateway rules, and framework defaults may override reviewed files:

```
# does production answer methods you never use?
curl -i -X TRACE https://api.example.com/invoices
# want: HTTP/1.1 405 Method Not Allowed

# does CORS reflect arbitrary origins?
curl -i -X OPTIONS https://api.example.com/invoices \
  -H "Origin: https://evil.example" \
  -H "Access-Control-Request-Method: GET"
# want: no Access-Control-Allow-Origin header for unknown origins

# are the interactive docs public?
curl -i https://api.example.com/docs
# want: 401 or 404 in production, or a deliberate decision otherwise
```

A middleware that reflects any `Origin` value back into `Access-Control-Allow-Origin` is wildcard CORS wearing a disguise. It passes a casual config review and fails this probe.

Check the downstream path too

Verify TLS across downstream connections too. The gateway terminating HTTPS means little if the hop to the application or database crosses a shared network in plaintext:

```
psql "host=db.internal dbname=invoices sslmode=require" -c "SHOW ssl;"
#  ssl
# -----
#  on
```

Save response headers and probe results beside the intended configuration after every production deployment. Configuration drifts quietly; the probes catch what review misses.

Capture effective production settings from the gateway and application, not only configuration files. Probe an unused method, foreign origin, debug route, and downstream plaintext connection, then record the denied result.

Monitor API abuse

Observe authorization failures, unusual object access, expensive requests, credential use, business-flow spikes, and configuration changes.

A single denied request may be noise. A pattern across accounts, objects, or regions may be an active attack. Monitoring is what turns individual log lines into that pattern.

Log the fields that make patterns visible

Log safe actor, credential, operation, target, result, cost, and correlation identifiers:

```
{
  "ts": "2026-08-03T09:14:02Z",
  "requestId": "req_7f3ab910",
  "actor": "usr_8817",
  "credential": "key_analytics_prod_covk",
  "op": "GET /invoices/:id",
  "target": "inv_00214",
  "result": 404,
  "dbMillis": 4
}
```

No tokens, no bodies with card numbers. The log itself must stay safe to read broadly, or access to it becomes the next incident.

Alert on deviations, not single events

One failed invoice lookup is normal. One token reading sequential invoice IDs across 40 accounts is a stronger object-access signal:

```
alert: object-enumeration
  when:  one credential produces > 30 distinct 404s
         on /invoices/:id within 5 minutes
  show:  actor, credential, sample targets, request IDs
  action: revoke-credential runbook (owner: platform-team)
```

Build alerts around meaningful deviations and high-impact actions: authorization failures clustering on one route, a refund spike, an admin permission granted at 3 AM, a configuration change nobody deployed.

Tune the signal with known traffic before paging anyone. Replay a week of production logs against a new rule. If it fires on your own nightly batch jobs, fix the rule now, because an on-call rotation learns to ignore a noisy alert within days.

Give responders a lever

Detection without containment is bad news delivered faster. Give responders a way to revoke credentials and limit operations quickly. The alert should show why the pattern is unusual and give the responder a reversible containment action with a named owner.

Reversible matters. Revoking one credential or lowering one rate limit can be undone in minutes if the alert was wrong. Blocking a whole IP range cannot, and false positives there hurt real customers.

Generate a normal access pattern and a cross-account enumeration pattern. Prove the second

creates one actionable alert with actor, targets, request IDs, owner, and a tested revocation step.

Retire and respond

Remove obsolete API versions, revoke old credentials, preserve evidence, and rehearse response to data exposure or authorization failure.

An API version is not retired when the documentation disappears. It is retired when callers and credentials can no longer reach it. Deleting `/v1` documentation does not stop an old token from calling it.

Retire with evidence, not hope

Measure real use, announce a deadline, migrate clients, block new credentials, disable the deployment, and monitor attempted calls.

Start with who still depends on the old version:

```
# which credentials still call /v1?
grep ' /v1/' /var/log/nginx/access.log | awk '{print $3}' | sort | uniq -c | sort -rn
#  4102 key_mobile_prod_x91b   <- this client needs a migration plan
#     7 key_partner_acme_v22
```

Immediate shutdown reduces exposure, but measured migration may be needed for clients you cannot update instantly. A mobile app version already in the field takes months to drain.

When the deadline arrives, make the block explicit and permanent:

```
curl -i https://api.example.com/v1/invoices \
  -H "Authorization: Bearer $OLD_TOKEN"
# HTTP/1.1 410 Gone
# {"error":"v1 retired 2026-10-01, see /docs/migration-v2"}
```

After blocking the route, watch denied calls for an agreed period. That evidence reveals forgotten clients without reopening the version or trusting an inventory that may already be incomplete.

Rehearse the incident before you have one

For an incident, identify affected objects and identities, contain the path, preserve evidence, fix related routes, and communicate from facts.

The tenant-boundary rehearsal is worth running on purpose. Given the claim ”tenant acme's export leaked”, can you list which invoices were touched, by which credential, in what time range?

```
SELECT target, credential, ts FROM audit_events
WHERE tenant = 'acme' AND op LIKE '%export%'
  AND ts BETWEEN '2026-07-01' AND '2026-08-01';
```

If that query cannot be answered from today's logs, you found the gap in a rehearsal instead of during a disclosure deadline. Fix the logging first; the response plan depends on it.

Record real `/v1` traffic, credential owners, migration dates, and the final block rule. Call the route after retirement and prove it cannot mutate data, then rehearse identifying invoices touched by a leaked tenant boundary.

Check your understanding: test and operate apis

Check security testing, monitoring, configuration, version retirement, incident response, and final review.

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/api-security/.