When Does a Permission Actually Expire?

A guided tour of a real authorization bug in nsecbunkerd — and the design decision it forces

aiolabs — nsecbunkerd (issue #24)

2026-06-19

Contents

How to read this document

You're an engineer who can read code but hasn't necessarily seen this codebase, Nostr, or this *class* of bug before. This document is written for you. It does three things, in order:

1. **Orients you** — what this software is, and what it's protecting.
2. **Walks the bug** — line by line, what goes wrong and why.
3. **Frames the decision** — the team isn't just "fixing a bug." We've found that one design choice is generating a *family* of bugs, and we're deciding how to redesign it so the family stops appearing.

Every term in **bold** is defined in the **Glossary** at the end. If a sentence loses you, jump there and come back. You are not expected to already know what a "materialized grant" or a "NIP-46 bunker" is.

Part 1 — What is this thing protecting?

Nostr keys, in one minute

Nostr is a protocol where your identity *is* a cryptographic key pair: a **public key** (`npub…`, safe to share — it's your username) and a **private key** (`nsec…`, secret — it's your password *and* your signature stamp combined). Anything you publish to the network is **signed** with the private key. Anyone can verify, using your public key, that you really wrote it. There is no central server that owns your account; the key is the account.

This is powerful and dangerous for the same reason: **whoever holds the private key IS you.** If it leaks, an attacker can post as you, drain funds tied to you, impersonate you forever. You cannot "reset your password" — there is no password, only the key.

So where do you keep the key?

If every app that wants to sign on your behalf needs a copy of your private key, every one of those apps is a place it can leak from. That's unacceptable for anything serious — an ATM, a point-of-sale device, a web app on a phone.

The fix is a **remote signer**, also called a **bunker**. The idea:

- The private key lives in **one** hardened place (the bunker) and *never leaves it*.
- Apps that want something signed don't get the key. They send a *request* — "please sign this message for me" — to the bunker.
- The bunker decides whether to allow it, signs, and sends back only the signature.

This is exactly what a bank vault is for. The teller (the app) never holds the gold; they pass a slip to the vault, and the vault decides.

nsecbunkerd is one such bunker. The "d" is for *daemon* — a long-running background program. The standard that defines how apps talk to a bunker is called **NIP-46** (think of NIPs as the numbered chapters of the Nostr rulebook). Our particular consumer is **LNbits**, a Lightning/Bitcoin server that uses the bunker so it never has to store Nostr private keys on disk.

The job that matters here: deciding "is this allowed?"

Because the bunker signs on behalf of others, its single most important job is the gatekeeping decision:

> A request just arrived from app X asking me to sign a message of type Y. **Should I?**

That decision function is the **ACL** — Access Control List. In this codebase it's one function, `checkIfPubkeyAllowed`, in `src/daemon/lib/acl/index.ts`. Everything in this document orbits that function. If it says "yes" when it should say "no," the bunker signs something it shouldn't — the worst thing a vault can do.

Part 2 — The feature: time-limited access

Tokens and policies

When you want to let an app use your key through the bunker, you don't hand it raw permission forever. You issue it a **token** — a one-time secret string the app redeems to "pair" with the bunker, a bit like a hotel key card you're handed at check-in.

Attached to the token is a **policy**: the set of things this app is allowed to do (which request types, which message kinds). The key card analogy holds: your card opens *your* room and the gym, but not other rooms and not the manager's office.

The TTL — the feature at the center of the bug

A token can be issued with an **expiry** — a timestamp after which it should stop working. In the code this is the `expiresAt` field, set from a `durationInHours` parameter when the token is created (`create_new_token.ts:27`):

```
if (durationInHours)
  data.expiresAt = new Date(Date.now() + parseInt(durationInHours) * 60*60*1000);
```

This is a **TTL** — Time To Live. The product feature it powers is “per-device token expiry”: hand a kiosk a key card that *automatically stops working* after, say, 24 hours, so a forgotten or stolen device can’t sign forever. The whole *point* of a TTL is to bound how long access lasts.

Hold onto that sentence. It’s the promise the system makes. The bug is that the system doesn’t keep it.

Part 3 — The bug, in plain language

Here is the entire bug in one breath:

> The expiry is checked **once**, at the moment the app first connects. After that, it is **never looked at again**. An expired token keeps signing forever.

The key card that was supposed to deactivate after 24 hours? It checks the expiry date *as you walk in the front door* — and then the door is propped open behind you. Come back a week later and walk right in. Nobody re-checks the card.

Concretely, the sequence is:

1. App redeems a token that expires in 1 hour. The bunker checks: not expired yet → **OK, you’re paired.**
2. The bunker writes down the app’s permissions (we’ll see exactly how in Part 4).
3. An hour passes. The token is now expired.
4. The app asks the bunker to sign something.
5. The gatekeeper (`checkIfPubkeyAllowed`) says **yes** — because it consults the notes from step 2, which never mentioned an expiry date.

The expiry was real at step 1 and invisible by step 4.

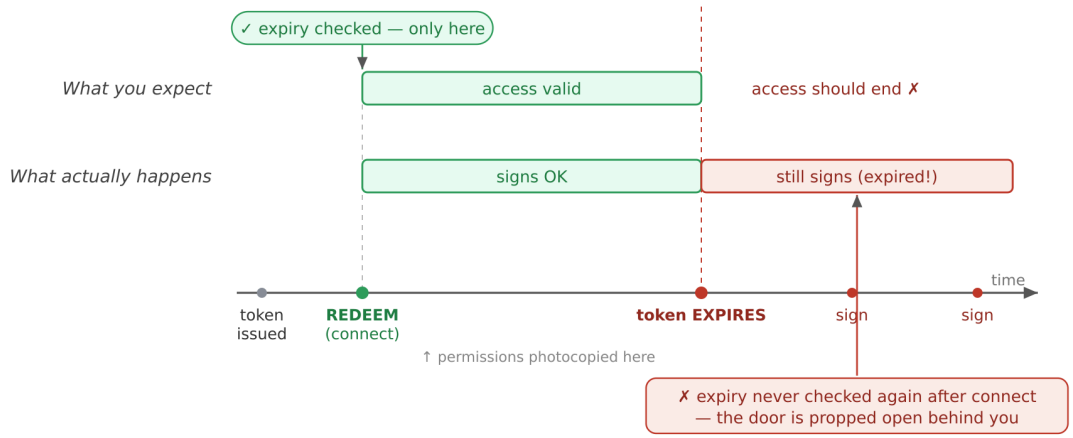

Figure 1: The two clocks. Expiry is checked once — at connect — and never again. The "valid" window the operator expects (top) and what the bunker actually does (bottom) diverge the moment the token lapses.

Part 4 — Walking the code

You don't need to be fluent in TypeScript or this database layer to follow this. There are two moments that matter: **redeem time** (when the app first connects) and **sign time** (every later request). The bug is the gap between them.

> A note on the database tooling: this code talks to its database through **Prisma**, an **ORM** — a library that lets you read/write database rows as if they were normal objects (`prisma.token.findUnique(...)`) instead of writing raw SQL. When you see `prisma.something`, read it as "go look in the `something` table." That's all you need.

Redeem time — expiry *is* checked (`backend/index.ts:84`)

When the app first pairs, `validateToken` runs. Note the last guard:

```
private async validateToken(token: string) {
  const tokenRecord = await prisma.token.findUnique({ where: { token }, ... });
  if (!tokenRecord)            throw new Error("Token not found");
  if (tokenRecord.redeemedAt)  throw new Error("Token already redeemed");
  if (!tokenRecord.policy)     throw new Error("Policy not found");
  if (tokenRecord.expiresAt && tokenRecord.expiresAt < new Date())
    throw new Error("Token expired");   // <-- the ONLY expiry check in the
system
  return tokenRecord;
}
```

That last line is the *only place in the entire codebase* that compares `expiresAt` against the current time. It runs exactly once per token, at pairing.

What pairing writes down (`applyToken`, `backend/index.ts:99`)

Right after validating, `applyToken` *copies* the policy's rules into a second table called `SigningCondition` — one row per permission:

```
for (const rule of tokenRecord.policy.rules) {
  await prisma.signingCondition.create({
    data: { keyUserId: upsertedUser.id, method: rule.method,
            allowed: true, kind: rule.kind?.toString() },
  });
}
```

This act of copying-permissions-into-a-fast-local-table is called **materialization**. Remember it — it is the source of the entire problem. The copied rows say *what* is allowed (`method`, `kind`, `allowed: true`). They say **nothing about when it expires.** Look at the table's definition (`prisma/schema.prisma:59`):

```
model SigningCondition {
  id        Int
  method    String?
  kind      String?
  allowed   Boolean?
  keyUserId Int?
  // <-- there is no expiresAt column here. None.
}
```

The expiry lived on the **token**. The copy was made from the **policy**. The expiry was simply **not on the photocopy.**

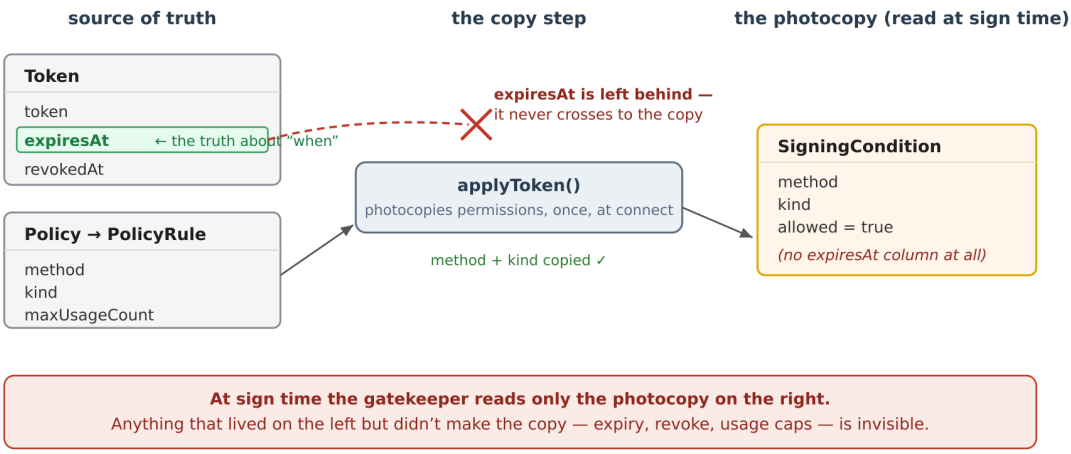

Figure 2: Materialization, and the drift it causes. `applyToken` copies `method` and `kind` across, but `expiresAt` lives on the token and never makes the copy. At sign time the gatekeeper reads only the right-hand box — so expiry, revoke, and usage caps are all invisible to it.

Sign time — expiry is *not* checked (`acl/index.ts:23`)

Now every later request runs `checkIfPubkeyAllowed`. It checks things in a deliberate order (the order is load-bearing — denials must beat grants). Simplified:

```
Step 1: Is this app even paired (does a KeyUser row exist)?   no -> ask a human
Step 2: Has the whole user been hard-revoked?                 yes -> DENY
Step 3: Is there a matching SigningCondition row?             yes -> return its
allowed flag  <-- (!)
Step 4: Otherwise, check the live policy join.
Step 5: Nothing matched -> ask a human
```

The bug lives at **Step 3** (`acl/index.ts:60-70`):

```
const signingCondition = await prisma.signingCondition.findFirst({
  where: { keyUserId: keyUser.id, ...signingConditionQuery },
});
if (signingCondition && (signingCondition.allowed === true || ...)) {
  return signingCondition.allowed;   // returns true. Never looked at any clock.
}
```

It finds the photocopied row from pairing, sees `allowed: true`, and returns. There is no date anywhere in this path to compare against the clock — because, as we saw, the photocopy never carried one. Step 3 also **short-circuits**: it returns immediately, so even the more careful Step 4 below it never runs.

And Step 4 (`acl/index.ts:93`), the "live" check that reads the real token, *also* misses it — it only filters out **revoked** tokens, never **expired** ones:

```
const policyAllowance = await prisma.token.findFirst({
  where: { keyUserId: keyUser.id, revokedAt: null, /* expiresAt: NOT CHECKED */
policy: {...} },
});
```

So both possible paths are blind to expiry. That's the bug, fully located.

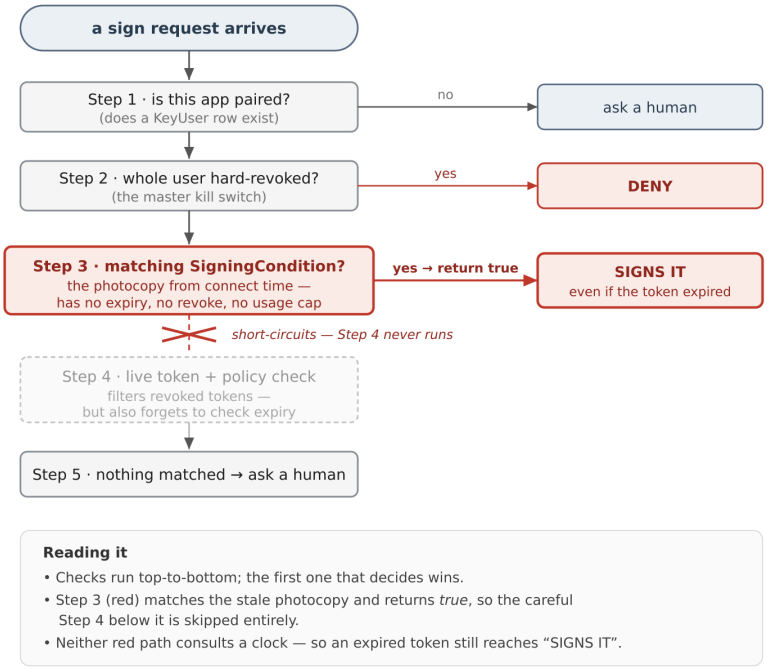

Figure 3: The gatekeeper's decision order. The first check that decides wins. Step 3 matches the stale photocopy, returns `true`, and short-circuits — so Step 4 never runs. Neither red path looks at a clock, so an expired token still reaches "SIGNS IT".

Part 5 — The deeper pattern (this is the real lesson)

It would be easy to "fix" this by adding an expiry check in a couple of spots and moving on. **Resist that.** Step back and notice *why* the bug was possible, because the shape will recur for the rest of your career.

Source of truth vs. a copy of it

Every fact in a system has a **source of truth** — the one authoritative place it lives. Here, the truth about "when does this access end?" is the token's `expiresAt`.

`applyToken` made a **copy** (the `SigningCondition` rows) for convenience and speed — a local, pre-chewed version the gatekeeper can read quickly (this is exactly the copy step drawn back in Figure 2). In computing this is a **cache**: a fast copy of something whose real home is elsewhere.

Caches have one famous, unavoidable hazard, summarized in a programmer's proverb:

> "There are only two hard things in computer science: cache invalidation and naming things."

Cache invalidation is the problem of *noticing when the original changed and updating (or throwing away) the copy.* Our cache was made at pairing time and then **never invalidated**. The original (the token) quietly expired; the copy never got the memo and kept saying "allowed."

That's the whole bug, abstractly: **a cache with no invalidation.** The expiry didn't fail to fire because someone wrote `<` instead of `>`. It failed because the value that *was* checked at sign time was a stale copy that never carried the expiry in the first place.

It isn't one bug. It's three (so far).

Once you see the shape, you find its siblings. Anything that lives on the **token or policy** but gets left off the **photocopy** will be ignored at sign time:

The feature	Where the truth lives	Enforced at sign time?
Token **expiry** (TTL)	`Token.expiresAt`	**No** (this issue, #24)
Token **revoke**	`Token.revokedAt`	**No** (sibling, spire-keeper #22)
Usage limits	`PolicyRule.maxUsageCount`	**No** (a capped policy signs forever)

Three "features" that each look implemented — there are fields, there's UI showing them — but none actually constrains a signing request, because the gatekeeper reads the photocopy, and the photocopy carries none of them. They are not three unrelated bugs to swat one at a time. They are **one design decision** (materialize permissions at pairing) producing one symptom per life-cycle rule. Swat them individually and the *next* lifecycle feature anyone adds will arrive broken too.

This is why the team is treating #24 as a design decision, not a one-line patch. The goal is to make the whole *family* impossible, not to fix today's instance of it.

Part 6 — The decision at hand

There are exactly two principled ways to make a cached system correct. Both are legitimate. We have to choose one as the foundation.

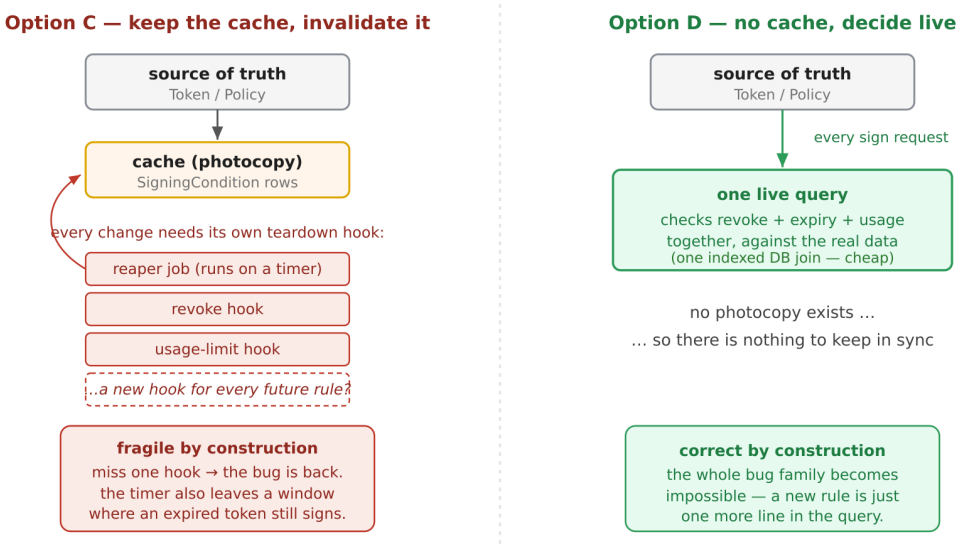

Figure 4: The two foundations. Option C keeps the fast copy but must actively tear it down on every change — one hook per rule, forever, and a timer gap besides. Option D throws the copy away and decides live, so there is nothing to keep in sync and the bug family cannot recur.

Option C — Keep the cache, but invalidate it

Keep materializing permissions at pairing for speed, but add machinery that *actively tears down* the copy whenever the original changes:

- a background **reaper** job that periodically wakes up, finds expired tokens, and deletes their photocopied rows;
- a hook on revoke that deletes the copies;
- a hook on usage-limit that deletes the copies;
- …and a new hook for every future lifecycle rule anyone invents.

Why it's tempting: it's the smallest change to today's code, and it reuses a revoke path that already partly works.

Why it's dangerous as a foundation: correctness now depends on *remembering to add an invalidation hook every single time*. Miss one — or add a new feature and forget — and you've silently recreated this exact bug. The reaper also runs on a timer, so there's always a window where an expired token still signs. And it's coarse: tearing down a user's copies can knock out *other* still-valid permissions they hold. This is the design that *generated* the bug family; doubling down on it keeps the family alive.

Option D — Don't cache. Decide live, every time.

Stop photocopying. At pairing, just record the *relationship* ("this app is paired via this token") and **nothing else**. Then, on every signing request, compute the answer **fresh** from the real tokens and policies — checking revoke **and** expiry **and** usage in one query.

Why it's the forward-looking choice: there is no copy, so there is nothing to keep in sync, so the entire bug family becomes **impossible by construction.** Add a new lifecycle rule in the future and it's just one more condition in the live query — it can't be "forgotten on the photocopy," because there is no photocopy. New correctness comes for free.

The usual objection to "decide live" is performance — re-computing every time sounds expensive. Here it genuinely isn't:

- The live check is a single indexed database join (fast).
- Every signing request already does **5–10 milliseconds** of cryptography over the network. One small database query is *noise* next to that.
- The software **hasn't been released yet.** We are free to make the clean choice now, with no users to migrate and no compatibility debt.

The recommendation on the table is **Option D**, with one extra discipline that prevents the original drift from ever returning:

> **Define "is this token valid right now?" in exactly one place**, and use that same definition at *both* redeem time and sign time. The original bug was possible because redeem-time checked expiry and sign-time didn't — two definitions of "valid" that disagreed. Make it one definition and they *cannot* disagree.

A clean result that falls out of Option D

Designing it this way also forces a useful clarification — the difference between two words people use loosely:

- **Revoke** = a *subject-level ban.* "This app/user is cut off." It's sticky and beats everything; un-banning must be a deliberate admin action. Trying to re-pair a banned user should fail loudly.
- **Expiry** = a *grant-level lapse.* "This particular key card ran out." Not a ban. Re-pairing with a fresh token simply adds a new valid card; the user was never banned.

In the cached design these two blur together (both end up as "delete the photocopy"). In the live design they stay cleanly distinct, which is exactly what you want when reasoning about security.

The sub-decision still open

If we take Option D, one modelling question remains — the one genuinely hard-to-reverse choice, which is why it wants a human decision:

- **D1 — Two typed sources, one shared rule.** Keep two kinds of permission — those derived from a token's policy, and those a human admin grants by hand ("just allow this app, manually"). Evaluate both live, both carrying a proper lifecycle, both run through the *one shared* "is it valid now?" rule. Lower ceremony; clean audit trail.
- **D2 — Everything is a token+policy.** Even a manual "allow this" becomes a tiny auto-generated policy. Exactly one code path for everything — maximally uniform — at the cost of more ceremony on the manual path and a lot of trivial one-rule policies cluttering the data.

Both are defensible. The robustness win (killing the bug family) comes from Option D itself, not from this sub-choice — so this one is a matter of taste about uniformity vs. simplicity, and it's the question we want a human to weigh in on.

Part 7 — What to take away

1. **The visible bug:** a token's expiry is checked when an app first connects, then never again, so expired tokens sign forever (#24).
2. **The real cause:** permissions are *photocopied* at connect time (**materialized**), and the photocopy never carried the expiry — a **cache that is never invalidated.**
3. **The pattern, not the instance:** the same cause already produces two siblings (revoke, usage-limits) and will produce more. That's why it's a design decision, not a patch.
4. **The choice:** keep the cache and bolt on invalidation machinery (**Option C**, fragile, recreates the bug on the next feature) vs. stop caching and **decide**

live every time (**Option D**, makes the whole family impossible, cheap enough here, and we're pre-release so it's free to do right).

5. **The discipline that seals it:** define "valid right now" in *one* place used by both connect-time and sign-time, so the two can never drift apart again.

Glossary

ACL (Access Control List). The logic that answers "is this request allowed?" Here it's the function `checkIfPubkeyAllowed`.

Bunker / remote signer. A hardened service that holds a private key and signs on the owner's behalf, so apps never touch the key itself. nsecbunkerd is one.

Cache. A fast, convenient copy of data whose real home ("source of truth") is somewhere else. Fast to read, but can go stale.

Cache invalidation. Noticing that the original data changed and updating or discarding the stale copy. Famously hard; the omission of it is the root of this bug.

Daemon. A long-running background program (the "d" in nsecbunkerd).

Grant. A permission: a thing an app is allowed to do. May come from a token's policy or from a manual admin decision.

Kind. A number tagging the *type* of a Nostr message (a post, a reaction, a payment receipt, etc.). Policies can allow some kinds and not others.

Lifecycle (of a permission). The rules about *when* a permission is valid — when it starts, when it expires, when it's revoked, how many times it can be used. The bug is that lifecycle rules are ignored at sign time.

LNbits. A Lightning/Bitcoin server; the consumer that uses this bunker so it doesn't store Nostr private keys.

Materialization. Pre-computing a result and storing it as rows for fast reading later — making a "photocopy." Here, copying a policy's rules into `SigningCondition` rows at pairing time. The convenience that caused the bug.

NIP / NIP-46. A NIP is a numbered chapter of the Nostr spec ("Nostr Implementation Possibility"). NIP-46 is the one defining how an app talks to a remote signer/bunker.

Nostr. A decentralized protocol where your identity is a cryptographic key pair and everything you publish is signed by your private key.

ORM (Object-Relational Mapper). A library that lets you use database rows as ordinary code objects instead of writing raw SQL. **Prisma** is the ORM here; `prisma.token.findUnique(...)` means "find a row in the `token` table."

Policy. The set of permissions attached to a token — what the app may do. Made of **PolicyRules**.

Public key (`npub`). The shareable half of a Nostr identity — your username. Used to verify signatures.

Private key (`nsec`). The secret half — password and signature stamp in one. Whoever has it *is* you. The thing the bunker exists to protect.

Reaper. A background job that periodically cleans up expired/stale data. One possible (but fragile) fix, "Option C."

Redeem (a token). The moment an app first uses its token to pair with the bunker. Expiry is (correctly) checked here — and, today, only here.

Revoke. To cut off access deliberately. We distinguish a *subject-level* revoke (ban the whole app/user, sticky) from a token simply *expiring* (a single grant lapsing).

Short-circuit. When a check returns early so later checks never run. Step 3 of the ACL short-circuits, which is why the more careful Step 4 never gets a chance to catch the expiry.

Sign / signature. Producing the cryptographic proof, using the private key, that a message genuinely came from this identity. The bunker's whole job, and what the ACL gates.

Sign time vs. redeem time. *Redeem time* = the one-off moment of pairing. *Sign time* = every later request to actually sign something. The bug is that expiry is enforced at redeem time but not sign time.

Source of truth. The single authoritative place a fact lives. For token expiry, that's `Token.expiresAt`. Bugs breed when code trusts a *copy* instead of the source.

Token. A one-time secret an app redeems to pair with the bunker — like a hotel key card. Carries a policy and, optionally, an expiry.

TTL (Time To Live). How long something stays valid before it should expire. The feature at the heart of #24.