How We Chose the Fix

From one expired-token bug to a design direction — by reading everyone else's homework first (a follow-up to "When Does a Permission Actually Expire?")

aiolabs — nsecbunkerd (issue #25, following #24)

2026-06-19

Contents

How to read this document

This is **Part 2.** Part 1 ("When Does a Permission Actually Expire?") explained a specific bug and *why* it happens. This part is about something you'll do far more often in a career than fix any single bug: **making a design decision well.** We had found a bug; the interesting work was deciding what to *do* about it — and we did that by going and reading how everyone else in the world had already solved (or failed to solve) the same problem.

So this document is half “here’s where we landed and why,” and half a worked example of a method: **don’t design in a vacuum — go read the field first.**

As before, every **bold** term is in the **Glossary** at the end, and you’re not expected to already know what “prior art” or a “threshold signature” is. If you haven’t read Part 1, the next section catches you up in one page.

Part 1 recap — the bug, in one page

If you read Part 1, skim this. If you didn’t, this is all you need.

nsecbunkerd is a **bunker**: a hardened service that holds a Nostr **private key** (the secret that *is* your identity — whoever holds it can sign as you) and signs messages on your behalf, so the apps that want signatures never have to touch the key itself. Apps pair with the bunker by redeeming a **token**, which carries a **policy** (what that app is allowed to do). A token can be given an **expiry** — a “**TTL**,” time-to-live — so a kiosk or a forgotten device automatically loses access after, say, 24 hours.

The bug (issue #24): that expiry is checked **once**, the moment the app first connects — and then **never again.** An expired token keeps signing forever. The cause is that, at connect time, the code makes a fast local **copy** of the app’s permissions (this copying is called **materialization**), and the copy doesn’t carry the expiry date. At sign time the gatekeeper reads the copy, not the original token, so the expiry is invisible. It’s a **cache** (a fast copy of data that really lives elsewhere) that is **never invalidated** (never updated when the original changes).

We also found the same cause produces two siblings: **revoke** (cutting off an app) and **usage caps** (limiting how many times a token may sign) are *both* ignored at sign time, for the identical reason.

The decision we faced (issue #25):

- **Option C** — keep the fast copy, but add machinery to tear it down whenever the original changes (a cleanup job, plus a hook for every rule, forever).
- **Option D** — stop copying; decide fresh from the real data on every request.

…and, if Option D, a sub-choice **D1 vs D2** about whether to keep two kinds of permission (token-granted and admin-granted) as distinct things or collapse everything into one. Part 1 recommended **D**, leaning **D1**, but flagged it as a real decision wanting evidence.

This document is how we got the evidence.

Part 2 — The move: read the field before you commit

It is very tempting, when you've understood a bug, to immediately start writing the fix. Resist that for any decision that's expensive to reverse — and a data-model decision in a security-critical service is exactly that.

There's a faster, humbler move first: **assume you're not the first person to hit this.** A remote signer that has to honor "this permission expires" is not an exotic problem; half a dozen projects in the Nostr world have shipped one. Each of them already made *our* decision, in production, and we can read the result. That's **prior art** — existing work that informs your own — and reading it converts a debate into an evidence-gathering exercise.

The question we took to the field was sharp and falsifiable:

> When a permission has a deadline, does this signer check the deadline on **every** signing request — or does it check once and cache the answer?

That single question sorts every implementation into "decides live" (Option D) or "trusts a copy" (Option C-ish), and tells us which camp *works.*

Part 3 — The field guide

We cloned and read the actual source of every NIP-46 signer worth learning from. (A signer is just a program that holds a key and signs on request; **NIP-46** is the shared rulebook chapter for how an app talks to a remote one.) Here is the landscape we found.

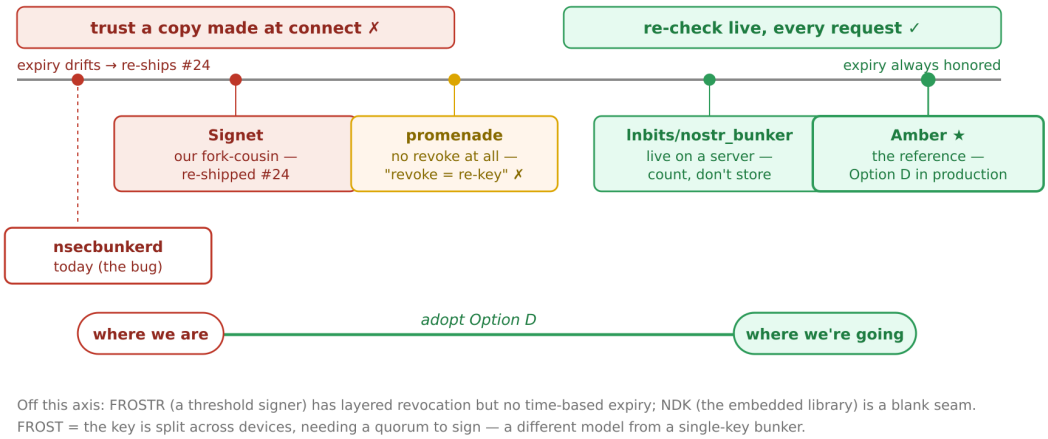

Figure 1: The field, on one axis: does the signer re-check a permission's deadline on every request, or trust a copy made earlier? The two that decide live (Amber, lnbits/nostr_bunker) work. The ones that trust a copy re-ship our exact bug. Our job is to move nsecbunkerd from the left to the right.

Amber — the positive template

Amber is a popular Android signer app: your key lives on your phone, and when an app wants a signature, Amber asks you to approve and *remembers* your approval with a deadline attached. The crucial detail, which we verified in its source: the function that decides "is this still approved?" **recomputes the deadline against the current time on every single request.** There's also a once-a-day cleanup sweep that deletes long-dead approvals — but that sweep is **cosmetic**, not load-bearing: if it never ran, nothing would be wrongly allowed, because the real check happens live each time.

That is **Option D, shipping in production, on phones, today.** It is the direct rebuttal to the natural worry that "deciding live every time will be too slow or too fiddly." Somebody already runs it at scale.

Amber also taught us two refinements we hadn't named:

- **Denials get a deadline too.** Amber can remember "reject for 5 minutes," which then decays back into asking again — instead of a permanent, forgotten "no." Symmetry worth copying.
- **Cache the *row*, never the *verdict*.** Amber keeps a fast copy of the approval *record* (with its deadline) for speed — but it still recomputes the yes/no against the clock on every cache hit. It never stores the *answer*. This is the

precise distinction our bug missed, and it's the single most portable lesson of the whole survey.

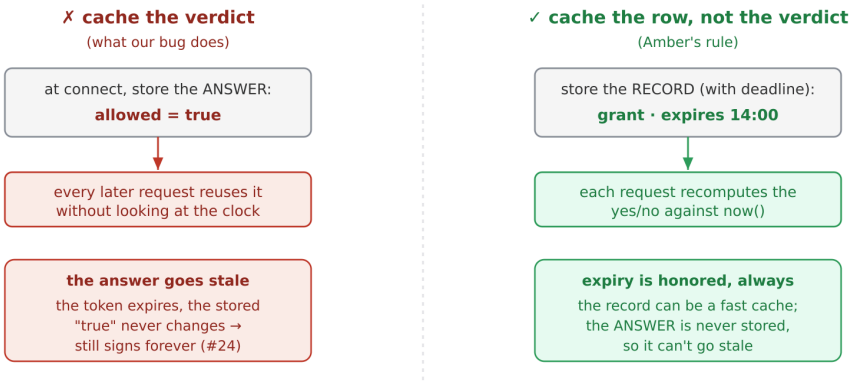

Figure 2: Amber's lesson, and the one rule that would have prevented our bug: cache the grant record if you want speed, but never cache the yes/no answer. Always recompute the verdict against the clock on each request.

lnbits/nostr_bunker — Option D on a server

`lnbits/nostr_bunker` is a server-side bunker, like ours. Its design is the stripped-down limit case: the permission *is* the connection record, and every signing request re-reads that live record — checking expiry and a rate limit each time, with **zero** copy-invalidation machinery. A second independent existence-proof that live evaluation is the normal, workable choice on a server, not just on a phone.

It also handed us a concrete fix for the **usage-cap** sibling: **count, don't keep a counter.** Instead of storing a "used 37 times" number that you have to remember to increment (which is just another copy that can drift), it *counts the signing-request records you already wrote* in the relevant window. Nothing to increment, nothing to invalidate — the count is derived from the truth on demand. We'll adopt exactly this and **delete our drift-prone counter.**

Signet — the cautionary mirror

Signet is the unsettling one: it's an extensive, actively-maintained **fork of our own codebase**, re-architected by another team around this very ACL problem. You'd hope they solved it for us. We read their source to find out, and the result is more useful than a solution would have been:

- They **independently re-shipped our exact #24.** Their connect-time code makes the same lifecycle-free **photocopy** ours does; their sign-time path

never re-reads the token. Token expiry and usage caps are dead in Signet for the identical reason.
- **But** they *did* fix one half cleanly: **revoke.** They added subject-level state — is this *app/user* banned or suspended? — on the account record, read live every request. And they put revoke on the **subject** (the app), not on the token, which independently confirms a distinction we'd already drawn (more on this in Part 5).

Two competent teams falling into the same materialization trap, on the same file, is strong evidence the trap is **structural** — a property of the design, not a careless oversight. It also tells us precisely which line to *not* copy from Signet, even as we borrow their schema for the parts they got right.

FROSTR and promenade — different world, two more lessons

These two are built on **threshold signatures** (**FROST**): instead of one secret key, the key is mathematically *split* across several devices, and some quorum (say 2 of 3) must cooperate to sign. Different security model, but they still face the "how do you revoke / expire access?" question, and their answers are instructive in opposite directions:

- **FROSTR** has a clean, layered idea of revocation — it separates "revoke this app's grant" from "revoke this device's transport" from "rotate the underlying key." Good influence: these really are three different operations and shouldn't be conflated.
- **promenade** has **no revoke at all** — to take away access, you **re-key** (rotate the master secret). This is the **anti-pattern** we want to consciously avoid: dropping one app's single capability should never force you to touch the master key, which is the most dangerous, most disruptive operation in the whole system.

NDK — the blank seam (and a bug we found in passing)

NDK is the off-the-shelf library we embed to speak NIP-46. It deliberately provides only a **blank seam** — a callback it invokes to ask "is this allowed?" — and leaves *all* the actual policy to us. Good: it means our redesign has nothing fighting it; we own 100% of the logic.

But reading it closely surfaced a separate problem, now filed as **issue #26**: NDK routes almost every request through that "is-this-allowed?" callback — *except* `get_public_key`, which it answers without ever asking us. In plain terms: **disclosing which identity a bunker holds is currently ungated and unaudited** through our permission layer, while every other action is gated.

That’s a real gap, found only because we went and read the seam we sit behind. It needs a deliberate “do we accept this, or override it?” decision as part of our “one check on every request” goal.

Part 4 — A note on rigor: trust the source, not the summary

One honest detail from how this survey actually went, because it’s a lesson in itself. Our *first* pass was a quick automated skim, and it **got several things wrong** — it claimed Signet enforced lifecycle live (it doesn’t), mis-stated what one threshold library could and couldn’t do, and got a couple of security parameters off by a factor of three.

Every claim in Part 3 was then re-checked **against the actual code**, at specific files and line numbers, and the errors corrected in the open. The takeaway for any engineer: **a confident summary is a hypothesis, not a fact.** Prior art is only worth as much as your willingness to open the file and verify it. A wrong “X already does this” can send a whole decision down the wrong road.

Part 5 — How the evidence added up to a direction

Lay the findings side by side and they point one way.

On Option C vs D: two production signers (Amber on phones, nostr_bunker on servers) decide **live** and enforce the exact trio we drop, with no invalidation machinery — so D is clearly *workable*. Meanwhile the one project that *kept the copy* (Signet) re-shipped our exact bug. The evidence isn’t merely that D is elegant; it’s that the copy-based approach has a track record of failing at this specific task, twice, independently. **Direction: Option D.**

But the survey also sharpened D with a distinction we now hold firmly — the one Signet’s split confirmed:

- **Subject-level** facts (“is this *app* banned or suspended?”) live on the app’s account record. There’s exactly **one** such row per app, and we *already read it* on every request as the first step of the permission check. So these are essentially free to check live — and, in Signet’s experience, the one place a small, carefully-invalidated cache is even reasonable.
- **Grant-level** facts (“has this *token* expired? used up its quota?”) live on the token and its policy. These are the ones that **cannot** be safely copied — proven by everyone who tried. They get read live, fresh, every request.

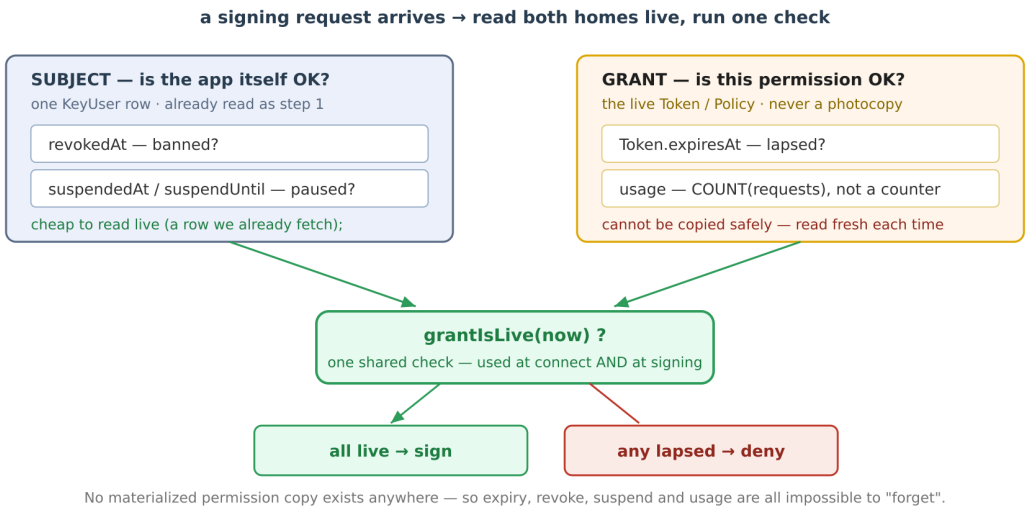

Figure 3: Where each rule lives, and how we enforce it. Subject facts (revoke, suspend) sit on the one app-record we already read each request. Grant facts (expiry, usage) are read live from the token/policy — usage by counting records we already keep, never a stored counter. Everything funnels through one shared "is this valid right now?" check.

This is why the answer isn't a flat "cache everything" or "cache nothing." It's: **cache the one subject row if you like (carefully); but never materialize grant lifecycle — read it live.** That precision is the gift of the survey.

On D1 vs D2: the two cleanest live implementations (Amber, nostr_bunker) collapse everything into one permission type — but they can *only* do that because they deleted the thing we can't delete. They have **no separate admin-grant path**; one app, one grant kind. We genuinely need *both* redeemable per-device tokens **and** interactive admin "just allow this" grants. So their uniformity (the spirit of **D2**) isn't evidence we should unify — it's evidence that unifying is free only when you've already thrown away one of the two things we must keep. Signet, which *does* keep two sources (a one-time connection token vs. a durable policy token), is the closer model. **Direction: D1** — keep token-granted and admin-granted permissions as two clearly-typed things, give both a real lifecycle, and run both through one shared "is this valid right now?" check.

Part 6 — The direction, in one page

Reiterating the target first: **we are fixing issue #24 — a token's expiry (and, by the same root cause, its revoke and usage cap) is honored**

only at connect time and ignored on every signature afterward. Here is the direction the evidence chose:

1. **Stop materializing grant permissions.** At connect time, record only that the app is paired — *not* a copy of what it may do. This is the one line of Signet's design we explicitly do **not** copy.
2. **Decide live on every request (Option D).** Compute the answer fresh from the real token and policy each time, the way Amber and nostr_bunker already do in production.
3. **One shared "is this valid right now?" check**, used at *both* connect time and signing time, so the two can never again disagree (the disagreement between them was the original bug). It checks expiry, revoke, suspension, and usage together.
4. **Usage caps by counting, not by a counter.** Delete the stored "used N times" number; count the signing records we already keep.
5. **Two homes, two tools.** Subject-level facts (revoke/suspend) on the one app-record we already read each request — cheap to check live, and the only place a small careful cache is even justified. Grant-level facts (expiry/usage) read live from token/policy — never copied.
6. **Keep two typed permission sources (D1):** redeemable tokens *and* interactive admin grants, both lifecycle-bearing, both funneled through the one shared check.
7. **Borrow the good, avoid the trap:** Amber's deadline-on-the-record + recompute-vs-clock and its time-boxed denials; nostr_bunker's count-don't-store; Signet's two-source schema — while avoiding promenade's "revoke means re-key" and Signet's materialization line.
8. **A bonus to resolve (#26):** close the `get_public_key` gap so *every* request — with no exceptions — passes the one check.

Because all three findings (expiry, revoke, usage) share one root, this direction closes the whole family at a single seam, rather than patching #24 alone and waiting for the next sibling to surface.

Part 7 — What to take away

1. **The bug we're fixing (#24):** a token's expiry is enforced at connect time and ignored at every signature after — and the same root cause silently breaks revoke and usage caps too.
2. **The method:** before committing to an expensive design decision, go read how the field already solved it. Prior art turns an argument into evidence.

3. **What the field showed:** deciding **live** (Option D) is proven and practical (Amber, nostr_bunker); keeping a **copy** has a track record of re-creating exactly our bug (Signet, twice now, independently).
4. **The refinement:** it's not "cache everything" vs "cache nothing" — it's *cache the record if you must, but never the verdict;* read grant lifecycle live; only subject state is even a cache candidate.
5. **The direction:** Option D, leaning D1 — read live, one shared check, count-don't-store, keep two typed sources, fix #26 — closing the whole bug family at one seam.
6. **The discipline underneath all of it:** trust the source, not the summary. Every claim here was checked against real code, and the first pass was wrong until it was.

Glossary

(Terms from Part 1 are repeated briefly so this document stands on its own.)

Amber. A widely-used Android Nostr signer app. Our positive reference: it decides permissions live, recomputing each approval's deadline against the clock on every request.

Anti-pattern. A common "solution" that looks reasonable but reliably causes trouble — something to recognize and avoid. promenade's "revoke = re-key" is one.

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

Cache. A fast copy of data whose real home is elsewhere. Fast to read, but can go **stale** if the original changes and the copy isn't updated.

Deadline (acceptUntil / rejectUntil). A timestamp attached to an approval (or a denial) saying when it stops being valid. Amber stores the deadline on the record and re-checks it live.

ECDH. A cryptographic operation used to encrypt/decrypt private messages between two keys. Relevant only to a side-note about threshold signers; not central here.

FROST / threshold signature. A scheme where a key is mathematically split across several devices and a quorum (e.g. 2 of 3) must cooperate to sign. FROSTR and promenade are built on it.

Grant. A permission an app holds — *what* it may do. May come from a redeemed **token** (token-granted) or from an admin decision (admin-granted).

Grant-level vs subject-level. A *subject* fact is about the whole app ("is this app banned?"). A *grant* fact is about one specific permission ("has this token expired? used its quota?"). The two belong in different places and are enforced differently.

Invalidation (cache invalidation). Noticing the original changed and updating or discarding the stale copy. The thing our bug never did.

Materialization. Pre-computing a result and storing it as rows for fast reading later — making a "photocopy." Here: copying a policy's rules into permission rows at connect time. The copy is lossy (it drops the expiry), which is the root of issue #24.

NDK (Nostr Dev Kit). The off-the-shelf library we embed to speak NIP-46. It provides a blank **seam** (a callback) and leaves all policy to us.

NIP-46. The chapter of the Nostr spec defining how an app talks to a remote signer / bunker.

Option C / Option D. The two ways to make a cached system correct. **C:** keep the copy, add machinery to tear it down on every change. **D:** don't copy — decide fresh from the real data every time. We chose D.

D1 / D2. Within Option D, a sub-choice. **D1:** keep two typed kinds of permission (token-granted and admin-granted). **D2:** collapse everything into one kind. We lean D1, because we genuinely need both.

Policy. The set of permissions attached to a token — what an app may do.

Prior art. Existing work (here, other people's signers) that already addresses your problem, which you read before designing your own.

Private key (`nsec`). The secret half of a Nostr identity. Whoever holds it *is* you. The thing the bunker exists to protect.

Predicate. A reusable yes/no test. Our planned `grantIsLive(now)` is one: "given the clock, is this permission valid right now?" — the single check every request runs.

Rate limit vs lifetime cap. Two different meanings of a usage limit: a *rate limit* is "N times per day" (a rolling window); a *lifetime cap* is "N times ever." Both can be enforced by counting records rather than keeping a counter — but you must decide which one you mean.

Reference implementation. A real, working example you can read and measure your design against. Amber is ours for live evaluation; Signet's schema is ours for structure.

Revoke. To cut off access deliberately. A *subject-level* revoke bans the whole app; it's sticky and beats everything.

Seam. A deliberate extension point in someone else's code (often a callback) where your logic plugs in. NDK gives us the permission seam.

Source-verified. Confirmed by reading the actual code at specific file/line, not taken from a summary. The standard every claim in the survey was held to.

Stale. A copy that no longer matches the original it was made from. A stale permission copy is what keeps signing after a token expires.

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 issue #24.

Verdict. The yes/no *answer* to "is this allowed?" The key lesson: cache the permission *record* if you want speed, but recompute the *verdict* live every time — never store the answer.