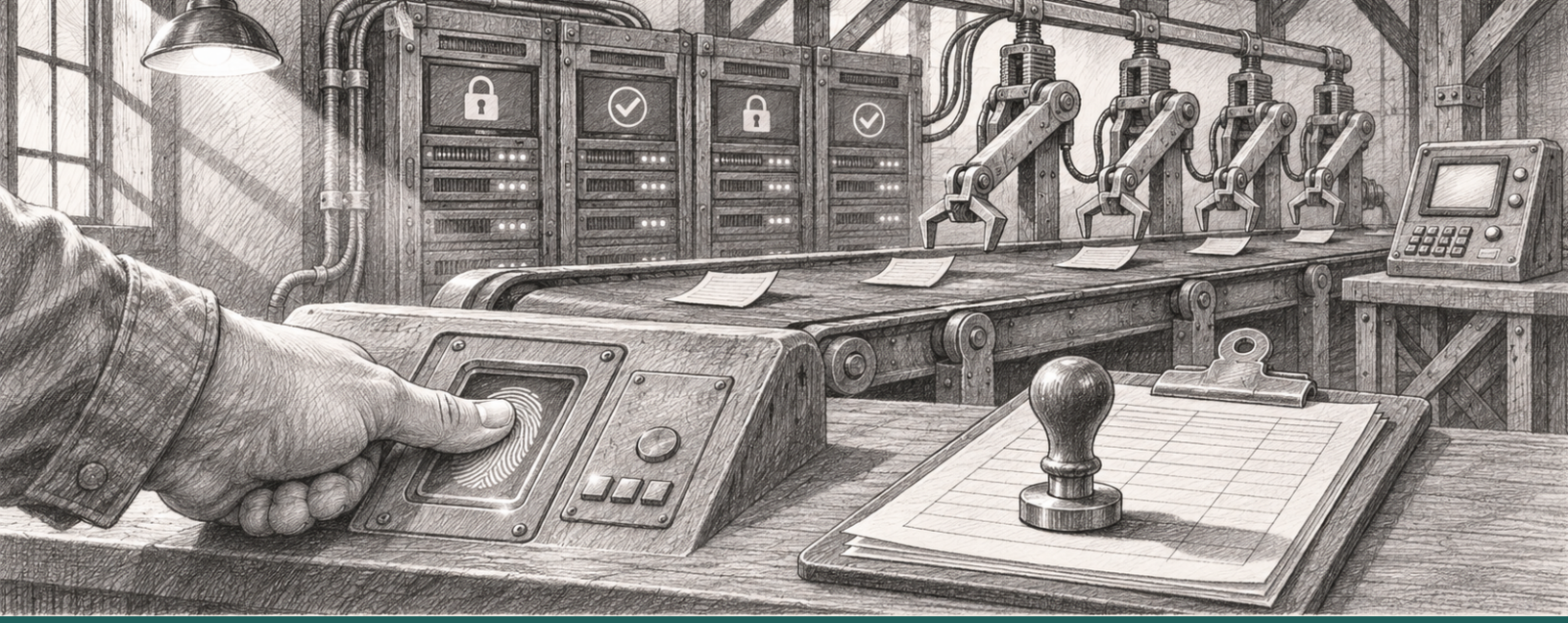

A HANDS-ON GUIDE

Transient Authority

Proof of Yes

Signed Human Approval for AI Agents with sjbis and ta

Asking • Proving • Running

November 22, 2026

Revision unknown

Stephen Barr

Contents

Preface 1
- 0.1 Who This Book Is For . . . 1
- 0.2 What You Need . . . 2
- 0.3 How This Book Is Organized . . . 2
- 0.4 Conventions Used in This Book . . . 2
  - 0.4.1 The approver's name . . . 3
  - 0.4.2 Example names . . . 3
- 0.5 Using the Examples . . . 3

I Asking a Human 4

1 Hearsay Is Not Permission 6
- 1.1 The Relay Problem . . . 6
- 1.2 Two Problems, Two Tools . . . 6
- 1.3 What a Grant Looks Like From the Outside . . . 7
- 1.4 Principles . . . 7
- 1.5 Summary . . . 8

2 Asking a Question with sjbis 9
- 2.1 One Binary, Two Roles . . . 9
- 2.2 Standing Up the Daemon . . . 9
- 2.3 Pointing the Client at the Daemon . . . 11
- 2.4 Your First Question . . . 11
  - 2.4.1 Agent names and instances . . . 12
- 2.5 Answer Shapes . . . 12
- 2.6 Blocking with a Deadline . . . 13
  - 2.6.1 Other reply channels . . . 13
- 2.7 The Working Agreement . . . 13
  - 2.7.1 The agent primer . . . 14
- 2.8 Managing Questions . . . 14
- 2.9 Summary . . . 15

3 Where the Human Answers 16
- 3.1 The Dashboard . . . 16
  - 3.1.1 How the dashboard shows an approval . . . 16

3.2 The Phone Web App . . . 17
3.2.1 Serving it over HTTPS . . . 17
3.2.2 Installing it . . . 17
3.2.3 Turning on push . . . 17
3.3 The Pi Console . . . 18
3.4 Rules: Keeping the Queue Short . . . 18
3.5 The Optional AI Router . . . 19
3.6 Summary . . . 19

II Proving the Answer **21**

4 Grants, Signatures and Short IDs **23**
4.1 Getting the Fixtures . . . 23
4.2 The Envelope . . . 24
4.3 The Record . . . 24
4.4 Canonical Bytes . . . 25
4.5 The Hash and the Short ID . . . 26
4.6 The Signature . . . 26
4.7 Verifying With ta . . . 26
4.8 Verifying Without ta . . . 28
4.9 Summary . . . 29

5 Setting Up ta **30**
5.1 Installing . . . 30
5.2 Configuration . . . 30
5.3 The Signer Registry . . . 31
5.4 Your First Signer . . . 32
5.5 Pinning Everywhere . . . 32
5.6 Checking the Setup . . . 33
5.7 The Agent Primer . . . 33
5.8 Summary . . . 35

6 Asking for and Verifying Authority **36**
6.1 Asking . . . 36
6.2 Answering . . . 37
6.3 What the Agent Gets Back . . . 37
6.3.1 Read the note . . . 38
6.4 Timeouts Carry No Authority . . . 38
6.5 Relaying . . . 39
6.6 Verifying: Show, Then Verify . . . 39
6.6.1 Why `--question` matters . . . 40
6.6.2 Verifying a denial . . . 40
6.7 Choices . . . 41
6.8 Summary . . . 41

7 Expiry, Renewal, Revocation and Single Use **42**
7.1 Lifetimes . . . 42

7.2 Renewal with `ta reauth` . . . 42
7.2.1 Lineage . . . 43
7.3 Revocation . . . 43
7.4 Single-Use Grants . . . 44
7.5 Exporting for Offline Verification . . . 45
7.6 Looking Back . . . 45
7.7 Summary . . . 46

III Running It **47**

8 Trusted Surfaces: Mac, Phone and Pi Console **49**
8.1 What Makes a Surface Trusted . . . 49
8.2 The Mac: Touch ID through `ta-se` . . . 49
8.2.1 Building and creating the key . . . 49
8.2.2 Pinning it and using it . . . 50
8.3 The Phone: Signing in the Web App . . . 51
8.4 The Pi Approval Console . . . 51
8.4.1 Setting it up . . . 52
8.4.2 The sidecar's contract . . . 52
8.4.3 The Stream Deck . . . 53
8.4.4 The signer ladder . . . 53
8.5 Comparing the Surfaces . . . 54
8.6 Summary . . . 54

9 The Threat Model, Honestly **55**
9.1 The Bar . . . 55
9.2 Who Can Do What . . . 55
9.3 What the Signature Proves, and What It Doesn't . . . 56
9.4 Accepted Gaps . . . 56
9.5 Things That Are Not Gaps . . . 57
9.6 Open Decisions . . . 57
9.7 Summary . . . 58

Command and Exit-Code Reference **59**
9.8 Exit Codes . . . 59
9.9 ta . . . 60
9.10 ta ask . . . 60
9.11 ta verify . . . 61
9.12 ta show . . . 62
9.13 ta reauth . . . 62
9.14 ta claim . . . 63
9.15 ta approve . . . 63
9.16 ta revoke . . . 64
9.17 ta console . . . 64
9.18 ta signers . . . 65
9.19 sjbis daemon endpoints for grants . . . 65

Wire-Format Reference **66**
9.20 Encodings 66
9.21 Envelope 66
9.22 Record Kinds 67
9.23 Daemon Errors on `POST /grants` 67

Preface

Agents now do real work while nobody watches: they open merge requests, migrate databases, rotate credentials and send mail. Most of the time that is the point. But some steps still need a person. Somebody has to say "yes, delete the staging database" or "no, not until the scan passes", and then the agent has to act on that answer, and often hand it to another agent who acts on it too.

That handoff is where things go wrong. One agent asks you a question in a terminal you are not looking at. Another agent tells a third that "we have permission". The third has no way to check. Permission travels as hearsay, and hearsay degrades at every hop.

This book is about two small tools that fix that, built and run by one person for his own fleet of agents:

- **sjbis** is a daemon that every agent can ask a question through. It queues the question, ranks it, puts it in front of you on a dashboard, a phone or a dedicated console, and routes your answer back to whatever process is waiting.
- **ta** (*transient authority*) is the attestation layer on top. When the question is one whose answer must be *provable*, you answer it on a trusted device that signs your decision. The agent gets back a short ID, such as `ta-ucfuud`, that any other agent can check for itself against public keys it has pinned. Relaying the ID is enough. Hearsay stops.

The book starts with one agent asking one question and ends with a fleet that relays signed, expiring, revocable approvals between agents, answered from an approval console that no agent can draw on.

Who This Book Is For

You run AI agents, or scripts that act on your behalf, and some of what they do needs your explicit permission. You are comfortable in a Unix shell, can read TOML and JSON, and know roughly what a public key is.

You will get the most from this book if one of these is true:

- Your agents ask you things in places you don't see in time, and you want one queue for every question from every agent.
- You have caught an agent claiming you approved something you did not, or approved something narrower than it claimed.
- You run several agents that hand work to each other and need a way for one to prove to another what you actually agreed to.

- You want a clear, small model of human approval that you can reason about: what is signed, who can forge what, and what happens when you do not answer.

You do not need to know Rust. Both tools are written in it, and a few later sections quote small pieces of the wire format, but every task in the book is a command you run.

What You Need

- A macOS or Linux machine with a recent stable Rust toolchain.
- An always-on Linux host for the sjbis daemon, with PostgreSQL reachable from it. A small VM is plenty.
- A private network between your machines, such as a WireGuard or Tailscale network. sjbis has no authentication of its own yet, so it must never face the public internet. Chapter 9 explains why that matters less for `ta` than it first appears.

Later chapters add optional hardware: a Mac with Touch ID, an iPhone, and a Raspberry Pi with a small screen and an Elgato Stream Deck. Each chapter that needs one says so at the start, and every earlier chapter works without it.

How This Book Is Organized

Part I, Asking a Human, covers sjbis. Chapter 1 describes the problem in detail and the principles that shape both tools. Chapter 2 installs sjbis and walks through asking: question types, blocking with a deadline, and what an agent should do when you don't answer. Chapter 3 is about the places you answer from: the dashboard, the phone web app, the Pi console, and the rules that keep your attention scarce.

Part II, Proving the Answer, covers `ta`. Chapter 4 takes a signed grant apart byte by byte, using real test vectors you can verify on your own machine. Chapter 5 installs `ta`, creates your first signer and pins it everywhere. Chapter 6 is the heart of the book: asking for authority, relaying it, and verifying it the right way. Chapter 7 covers what makes the authority *transient*: expiry, renewal, revocation and single use.

Part III, Running It, moves from mechanics to operation. Chapter 8 sets up each trusted surface: Touch ID on the Mac, the phone, and a Raspberry Pi approval console. Chapter 9 states the threat model plainly: what the system protects against, what it does not, and the decisions still open.

The appendices hold the command and exit-code reference and the wire-format reference.

Conventions Used in This Book

`Constant width` Commands, flags, file names, configuration keys, IDs and output.
`Constant width with CAPITALS, such as ASK_ID` A value you replace with your own.

> **Tip**
> A tip suggests a faster or safer way to do something.

> **Note**
> A note adds background that you can skip on a first read.

> **Warning**
> A warning marks something that can leak a key, grant authority you did not mean to grant, or let an agent act on a claim nobody checked.

The approver's name

Both tools were written for one person, and `ta`'s own messages name him: "Ask Stephen for a signed approval", "waiting for Stephen to answer". In this book *you* are the approver. Wherever a listing says Stephen, read it as you.

Example names

Every example uses placeholder infrastructure. Substitute your own values:

Placeholder	Stands for
`http://sjbis.example.com:7878`	The sjbis daemon's base URL
`https://sjbis.example.com:7443/phone`	The phone web app, served over HTTPS
`pg.example.com`	The PostgreSQL server the daemon uses
`deploybot`, `agent-x`, `agent-y`	Agent names
`touchid@mac`, `phone@iphone`, `console@pi4`	Signer IDs for your Mac, phone and Pi console
`sjbis-AbCdEfGh`	An sjbis ask (notification) ID

Using the Examples

Commands that only inspect, such as `ta verify --record`, `ta show`, `ta signers list` and `sjbis list`, never change anything. Commands that ask, approve, revoke or claim change state on the daemon, and approving creates authority that other agents will act on. Each chapter marks the boundary.

Listings that show `ta`'s own output are generated from the `ta` source and its test fixtures when the book is built, with the clock pinned so expiry times never drift. You can reproduce every one of them offline from a checkout of the repository. Where a listing needs a live daemon it is marked as an example, and the exact text on your machine will differ.

PART I

Asking a Human

Before an answer can be proved, it has to be asked for and given. This part covers sjbis, the daemon every agent asks through: how a question becomes a card in front of you, how your answer finds its way back, and what an agent should do when you are not there.

CHAPTER 1

Hearsay Is Not Permission

In this chapter you will see the problem that sjbis and `ta` solve, in the form it actually takes in a fleet of agents, and the handful of principles that shape every design decision in the rest of the book. There are no commands to run yet; the point is to know what you are building toward.

The Relay Problem

Here is a conversation that happens, in one form or another, every day in a fleet of agents:

> Agent X asks you whether it may delete the staging database. You say yes. X hands the cleanup to agent Y and tells it "we have permission". Y replies: "X says we have permission, but I want to hear it from you."

Y is right to be suspicious, and it has no way to stop being suspicious. The only evidence that you said yes is X's word. If Y accepts it, Y is trusting X's memory, X's honesty, and X's reading of what you meant. If Y refuses, it has to ask you again, and you answer the same question twice. Add a third agent and the claim is now "X told Y that you said yes", which is worth even less.

Each relay hop degrades the claim in a particular way:

- **It loses scope.** You said "yes, but only the two on the left". By the second hop it is "yes".
- **It loses time.** You said yes an hour ago, for the next hour. Nobody down the chain knows when the hour started.
- **It loses the question.** You approved "delete the staging database". The agent three hops away is about to delete the *production* database and believes it has the same permission.
- **It can't be withdrawn.** If you change your mind, there is no artifact to revoke, only a rumor to chase.

What is missing is an *artifact*: something an agent can hold up and say "this is the decision, check it yourself", that carries its own scope and expiry, and that cannot be altered on the way.

Two Problems, Two Tools

The relay problem is really two problems stacked on top of each other.

The first is **surfacing**. An agent that needs you has to reach you, at the right moment, without drowning you in noise, and your answer has to get back to the process that is waiting for it. That

problem has nothing to do with cryptography. It is about queues, deadlines, notifications, and keeping your attention scarce. sjbis solves it.

The second is **proof**. Once you have answered, the answer has to be something other agents can check without trusting the agent that asked, the daemon that carried it, or the database that stored it. That problem is almost entirely cryptographic, and it is small. `ta` solves it.

Keeping them apart is the most important design decision in this book. `ta` does not reinvent asking, routing, deadlines, push notifications or the dashboard; it uses sjbis for all of that. sjbis does not know how to sign or verify anything beyond rejecting obvious junk; it stores what `ta` gives it.

Concern	Owner
Transport, queueing, deadlines, push, dashboard	sjbis
Canonical record, ID, signing, verification, signer registry	ta
Storage of grants	sjbis's PostgreSQL database, as self-verifying rows
Trusted answer surfaces	Mac (Touch ID), phone, Pi console

What a Grant Looks Like From the Outside

You will take a grant apart in Chapter 4. From the outside it is three things:

1. **A short ID**, such as `ta-ucfuud`. Short enough for an agent to put in a sentence: "you approved the cleanup (ta-ucfuud)".
2. **A signed record** behind the ID that holds the verbatim question, your decision, your note, who asked, when you signed and when it expires.
3. **A verdict** that any agent computes for itself with `ta verify`, against a list of public keys it pinned in advance.

The conversation from the start of the chapter becomes:

> Agent X asks you through `ta ask`. You approve on your phone. X gets back `ta-ucfuud` and tells Y "approved, ta-ucfuud". Y runs `ta show ta-ucfuud`, sees the exact question you were asked, and runs `ta verify ta-ucfuud --question "..."`. Exit code 0. Y proceeds.

Y never had to trust X. It did not even have to trust the daemon that stored the grant: the daemon can lose or hide a grant, but it cannot forge one or alter one, because Y checks the signature against keys it holds itself.

Principles

These principles come up again and again. Each is short; each costs something, and later chapters show what.

The question is signed verbatim. The exact bytes the agent asked are inside the signed record. An agent can't later claim you approved something broader, and a verifier compares the question byte for byte (whitespace aside) rather than trusting a paraphrase.

Silence is never consent. sjbis's ordinary rule for a question you did not answer in time is "proceed on your best judgment". For an approval, that rule is switched off. If the deadline passes, there is no grant and no ID, and the agent must not claim it had permission.

Authority is transient. Every grant carries an expiry inside the signed record, so even an offline verifier enforces it. The default lifetime is one hour. Expired authority can be renewed, but it is never stretched.

Denials are first-class. "You explicitly said no, at ta-qo2xz7" is as useful as a yes. A signed denial is relayable, verifiable and permanent.

Verifiers trust only their own keys. A verifier never uses a key the daemon serves. The daemon keeps a copy of the registry only to reject junk early.

Presence, then signature. A grant is signed on a device you are holding or sitting at, after a physical act: a fingerprint, a face, a key press on a box no agent can reach. The agent's own terminal never renders the approval prompt.

The bar is "better than hearsay". None of this is meant to stop a nation-state. It is meant to make the honest path the easy one for cooperating agents, and to make a dishonest or confused claim fail loudly. You will see in Chapter 9 exactly where the line sits.

Why not just use the dashboard's answer?

sjbis already records every answer in PostgreSQL, with who answered and when. Why isn't that enough?

Because the record proves nothing once it leaves the process that asked. The daemon trusts any client that can reach it, so anything on your network could have written that row. An agent reading it from a log or a message has no way to distinguish your answer from one posted by another agent. And a row can be edited after the fact.

A signed grant changes what the evidence is. It is no longer "the daemon says you said yes" but "here are bytes that only your device could have signed". The daemon becomes a convenient place to find grants, not a party you have to trust.

Summary

Permission relayed between agents is hearsay: it loses scope, time and the question itself at every hop, and it can't be withdrawn. Fixing that takes two separate tools. sjbis surfaces questions and routes answers; `ta` turns an answer into a signed, short, expiring artifact that any agent verifies against keys it pinned itself. The question is signed verbatim, silence is never consent, and verifiers never trust the daemon's keys.

Chapter 2 starts on the surfacing half: installing sjbis and asking your first question.

CHAPTER 2

Asking a Question with sjbis

In this chapter you will stand up the sjbis daemon, point the `sjbis` client at it, and ask your first questions: fire-and-forget, blocking with a deadline, and in each of the answer shapes the dashboard can render. You will also learn the working agreement every agent in the fleet follows when you don't answer, because Chapter 6 is built on the one place where that agreement is switched off.

One Binary, Two Roles

sjbis is a single Rust binary that plays two roles:

- **The daemon** runs on an always-on host. It owns the PostgreSQL database, serves the dashboard, pushes live updates over Server-Sent Events, and exposes a small HTTP/JSON API. It is usually a systemd user service.
- **The client** is the same binary run from your workstation or from any agent. It talks to the daemon over HTTP and keeps no state of its own.

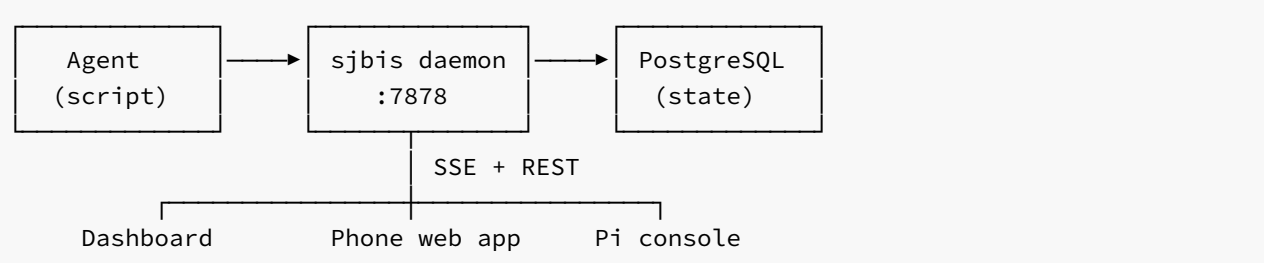

Figure 2-1. sjbis topology: one daemon, many askers, several answer surfaces.

The daemon is the only process boundary that matters. Everything else, the dashboard, the phone, the console and every agent, is a client of its API.

Standing Up the Daemon

You need an always-on Linux host and a PostgreSQL database it can reach. The daemon runs its own migrations on start, so an empty database is fine.

Build a static binary. On a Mac, cross-compile for Linux with `cargo-zigbuild`:

```
brew install zig
cargo install cargo-zigbuild
cargo zigbuild --target x86_64-unknown-linux-musl --release
```

The result, `target/x86_64-unknown-linux-musl/release/sjbis`, is a single statically linked file of about 8 MB with no runtime dependencies. Copy it and the dashboard's static files to the host:

```
HOST=sjbis.example.com
ssh "$HOST" 'mkdir -p ~/sjbis/static ~/.config/sjbis'
scp target/x86_64-unknown-linux-musl/release/sjbis "$HOST":~/sjbis/
scp static/*.js static/*.jsx static/*.css static/*.html "$HOST":~/sjbis/static/
```

Tell the daemon where its database is, in `~/.config/sjbis/database.toml` on the host:

```
[database]
dsn = "postgresql://sjbis:PASSWORD@pg.example.com:5432/sjbis"
```

Then run it as a systemd user service. Create `~/.config/systemd/user/sjbis.service`:

```
[Unit]
Description=sjbis daemon
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
WorkingDirectory=%h/sjbis
ExecStart=%h/sjbis/sjbis daemon start --port 7878
Restart=on-failure
RestartSec=5

[Install]
WantedBy=default.target
```

and start it:

```
systemctl --user daemon-reload
systemctl --user enable --now sjbis
loginctl enable-linger "$USER"     # keep it running when you log out
curl http://localhost:7878/health  # ok
```

> **Warning**
>
> sjbis has no authentication yet. Anything that can reach port 7878 can ask questions, answer ordinary ones, and write rules. Keep the daemon on a private network, such as a Tailscale or

WireGuard network, and never expose it to the internet. Approval asks, the subject of Part II, are the exception: they can only be answered with a signature the daemon can't produce.

Tip

To serve sjbis below a path, for example behind a reverse proxy at `/sjbis`, start it with `--base-path /sjbis`. Every route, including `/health`, moves under the prefix, and clients must use the full prefixed URL.

Pointing the Client at the Daemon

Install the client on your workstation with `cargo install --path .` from a checkout, then tell it where the daemon is. Either set an environment variable:

```
export SJBIS_DAEMON=http://sjbis.example.com:7878
```

or write `~/.config/sjbis/daemon.toml`:

```
url = "http://sjbis.example.com:7878"
```

Check the connection by listing open questions:

```
$ sjbis list
No open notifications.
```

Open `http://sjbis.example.com:7878` in a browser to see the dashboard. Keep it open for the rest of the chapter.

Your First Question

Every ask has three required parts: the **question** text, an **answer shape** that picks how the dashboard renders it, and an **agent name** that identifies the caller.

```
$ sjbis ask --question "Deploy build 412 to prod?" --yesno --agent-name deploybot
Posted: Deploy build 412 to prod? (id: sjbis-AbCdEfGh)
```

A card appears on the dashboard. Its source line shows `deploybot` with a glyph and color derived by hashing the name, so the same agent always looks the same. Answer it with `Y`.

That ask was fire-and-forget: the command returned at once with an ID, and nothing was waiting for your answer. You can look the answer up afterwards:

```
$ sjbis status sjbis-AbCdEfGh
id:        sjbis-AbCdEfGh
status:    answered
agent:     deploybot
question:  Deploy build 412 to prod?
answer:    yes
answered:  2026-11-07T15:02:11Z
```

> **Note**
>
> Fire-and-forget is the default on purpose. An automated workflow should never hang because a human is busy. An agent has to ask explicitly to wait.

Agent names and instances

`--agent-name` should be stable across runs of the same caller, because it drives the card's identity and the rules you will write in Chapter 3. Per-run detail goes in `--instance`, which is shown on the card but does not change identity or rule matching:

```
sjbis ask --question "Retry the flaky test?" --yesno \
  --agent-name codebot --instance "session s7b3d11"
```

Answer Shapes

The answer-shape flag picks the renderer. Each has keyboard shortcuts on the dashboard.

Shape	Flag	How you answer
Yes/no	`--yesno`	`Y` / `N`
Multiple choice	`--choices a,b,c`	`1`–`9`
Free text	`--text`	Type, then Enter
Number	`--number`	Slider, stepper, or type
File	`--file`	Drag and drop
Diff	`--diff`	Approve / reject
Acknowledge	`--ack`	Any key
Pick list	`--pick items.json`	`1`–`9`
Schedule slot	`--schedule slots.json`	`1`–`9`
Approval	`--approval`	Only a signed grant from a trusted device

The last row is the subject of Part II. For now, notice that it is just another answer shape as far as the queue is concerned: same card, same dashboard, same deadlines. What differs is who may answer it.

Context goes in `--detail` (plain text) or `--detail-markdown`. The question itself should stay short, under about 280 characters, because it is what you read first and, for an approval, what gets signed.

> **Tip**
>
> If an agent passes message bodies, links or tokens as context, have it use `--content-stdin` and write a JSON object with `question`, `detail` and `detail_markdown` to standard input. That keeps the content out of the process's argument list, where any user on the machine could read it with `ps`.

Blocking with a Deadline

When an agent actually needs your answer to continue, it asks with `--blocking` and a `--deadline`:

```
$ sjbis ask --question "Approve PR #412?" --yesno \
    --agent-name codebot --blocking --deadline 6m
Waiting for answer... (id: sjbis-QrStUvWx)
yes
```

The daemon holds the request open until you answer or the deadline passes, then returns. Agents never build their own polling loops; the daemon caps a single blocking wait at ten minutes, and the client reconnects transparently if the connection drops while the question is still open.

With `--json`, the result is a structured answer that includes the answer, your optional note, and `via`, which says how the question was resolved:

```
{
  "id": "sjbis-QrStUvWx",
  "answer": "yes",
  "note": "Security scan passed",
  "via": "dashboard"
}
```

Example: a blocking answer with `--json` (abbreviated).

Every answer surface lets you attach a short note. Agents must read it. "Yes" with the note "but only after 6 pm" is not the same answer as "yes".

Other reply channels

Standard output is the default reply channel for a blocking ask. `--reply-to` chooses another one: `webhook:URL` (with retries), `file:PATH` (written atomically), or `exit-code`. A process that was not waiting can reattach to an open question at any time with `sjbis wait ASK_ID`.

The Working Agreement

Every agent in the fleet follows the same pattern when it needs a decision: **ask, blocking, with a deadline; on timeout, use your best judgment and say what you did.**

```
# 1. Ask, blocking, with a deadline.
res=$(sjbis ask --question "Approve PR #412?" --yesno \
        --agent-name codebot --deadline 1m --blocking --json)

# 2. via == "timed_out" means nobody answered in time.
via=$(jq -r '.via // ""' <<<"$res")

# 3. Don't hang. Decide, then tell the daemon what you decided,
#    so the dashboard shows the auto-pick and your reason.
if [ "$via" = "timed_out" ]; then
  id=$(jq -r '.id' <<<"$res")
  sjbis answer "$id" --answer "no" --via caller-timeout \
    --note "No reply in time; held off because the PR touches auth."
fi
```

This is what makes sjbis safe to put in front of autonomous agents. Silence means "use your judgment", not "hang forever", and the human can always see, after the fact, which questions were decided by the agent and why.

> **Warning**
>
> That rule is deliberately reversed for approvals. If an approval ask times out there is no grant, and the agent must *not* record an auto-pick or act as if it had permission. A plain `sjbis answer` on an approval ask is refused with HTTP 409. Chapter 6 covers what the agent does instead.

The agent primer

New agents don't need to read this chapter. `sjbis prime` prints a primer written for agents: the working agreement, the question types, and the live status of the daemon it is configured to use. Put "run `sjbis prime` before asking anything" in your agents' standing instructions, and they will pick up the right pattern, including any changes to it, from the tool itself.

Managing Questions

A few more commands round out the client:

Command	Does
`sjbis list [--json]`	List open questions
`sjbis status ID`	Show one question's state and answer
`sjbis wait ID`	Block until an open question resolves
`sjbis answer ID --answer V`	Record an answer on the caller's behalf
`sjbis cancel ID`	Withdraw an unanswered question
`sjbis dismiss ID`	Mark as seen without answering; waiters wake with no answer

A question ends in exactly one terminal state: `answered`, `cancelled`, `muted` (a rule dropped it), `timed_out` or `dismissed`. Every question has a stable URL, `http://sjbis.example.com:7878/card/ID`, which you can paste into chat or mail; it opens read-only once the question is closed.

Summary

sjbis is one binary that is both a daemon, on an always-on host with PostgreSQL, and a stateless client. An agent asks with a question, an answer shape and a stable agent name; blocking asks with a deadline return your answer, your note, and how it arrived. On timeout an ordinary agent decides for itself and reports what it decided. Approvals are the one exception to that rule.

Chapter 3 turns to your side of the queue: the surfaces you answer from, and the rules that keep the queue short.

CHAPTER 3

Where the Human Answers

In this chapter you will work the queue from your side: answering from the keyboard-driven dashboard, installing the phone web app and turning on push notifications, and meeting the Pi approval console, a small dedicated screen with a Stream Deck. You will also write rules that keep the queue short, and decide whether to let a language model help route questions. By the end you will have every surface that Part II turns into a trusted signer.

The Dashboard

The dashboard is served by the daemon at its base URL. It subscribes to the daemon's event stream, so new questions appear the moment they are asked and disappear wherever they are answered. It is built for the keyboard:

Key	Action
`J` / `K`	Next / previous question
`Enter`	Open the focused question
`Y` / `N`, `1`–`9`	Answer
`S`	Snooze
`D`	Dismiss
`T`	Open the tweaks panel

Snoozing is universal, and it is capped at the question's deadline: you can't snooze past the moment an agent will act on its own judgment.

Every answer surface has a note field. Use it. A note is returned to the caller with the answer, and the working agreement tells agents to read it.

How the dashboard shows an approval

An approval ask appears in the same queue, but locked. The card shows a lock badge, the requesting agent, the verbatim question and detail, the requested grant lifetime and its age, and the line "answer on a trusted device". It has no answer buttons. You can still snooze or dismiss it, because neither of those creates authority.

The lock is enforced by the daemon, not by the page. A plain `POST /answer/ID` on an approval ask, from any client, is refused:

```
HTTP/1.1 409 Conflict
{"error":"approval asks can only be answered by a signed grant"}
```

The Phone Web App

The daemon also serves a phone-sized client at `/phone`: a list of open questions, one question at a time with big buttons, and a settings page. It is an installable web app, not an App Store app, so there is no developer account involved.

Serving it over HTTPS

Service workers, Web Push and in-browser cryptography all require a secure context, so the phone app must be served over HTTPS with a certificate the phone trusts. If your private network is Tailscale, the daemon host can do this itself:

```
tailscale serve --bg --https=7443 http://127.0.0.1:7878
```

That gives `https://sjbis.example.com:7443/phone` with a valid certificate. Any reverse proxy with a real certificate works the same way.

Installing it

On an iPhone, open the URL in Safari, tap Share, then **Add to Home Screen**, and from then on open it from the icon. Installing matters for two reasons: iOS delivers Web Push only to installed web apps, and it doesn't evict the storage of a Home Screen app, which is where the app will later keep its signing key.

Turning on push

In the app's settings, tap **enable notifications** and accept the prompt. The app subscribes with the daemon's VAPID public key and registers the subscription. From then on:

- every new open question is pushed, titled with the agent's name, with the question as the body;
- when a question is answered, cancelled or dismissed anywhere, a quiet push replaces its banner, so your lock screen doesn't fill with stale questions;
- tapping a banner opens that question, or tells you it was already answered, and by whom.

send a test push in settings round-trips one notification. Subscriptions that the push service reports as gone are dropped automatically.

> **Note**
>
> The daemon generates its VAPID key on first start in `~/.config/sjbis/vapid.pem`. Set `SJBIS_VAPID_SUBJECT` to a `mailto:` or `https:` contact; push services use it to reach the operator of a misbehaving sender.

> **Tip**
>
> Push notifications go out for questions with urgency 2 or higher, which is the default. An agent that wants to queue something quietly for later can ask with `--urgency 1`.

The Pi Console

The third surface is a piece of hardware: a Raspberry Pi with a small landscape screen and an Elgato Stream Deck, sitting on your desk and doing nothing but answering questions. Chromium runs in kiosk mode on the daemon's `/console` page, which shows one open question at a time, oldest first, in large type.

The console page needs no internet access: no fonts or scripts from a CDN. It sizes everything in viewport units, so a 1024×600 panel and a 1080p monitor look the same. The Stream Deck sends keystrokes, and the page reads them:

Key	Action
`Y` / `N`	Yes / no (on an approval: approve / deny)
`1`–`9`, `Enter`	Select a choice, then confirm it
`T`	Cycle the grant lifetime: 10m, 1h, 24h (approvals only)
`M`, `Esc`	Focus or leave the note field
`S` / `D`	Snooze 15 minutes / dismiss (press `D` twice)
`Left` / `Right`	Previous / next question
`R`	Reload the queue and recheck the signing sidecar

Why build a dedicated box when the dashboard and phone exist? Because no agent can draw on its screen. On your laptop, an agent with a terminal can open a browser window, and an agent that controls the screen can make a dialog look like anything. The console is a surface whose every pixel comes from the daemon's page, on a machine no agent can log in to. That property becomes important in Chapter 8, where the console gets a signing key of its own.

Rules: Keeping the Queue Short

Questions from agents are only part of what flows through sjbis. Plugins surface iMessage and Signal messages as cards, and any script can ask. Rules are how you keep the queue to what deserves your attention.

A rule is a line of plain English:

```
sjbis rule add "mute all iMessage except family for 1h"
sjbis rule add "auto-ack buildbot"
sjbis rule add "urgent only" --priority 10
sjbis rule list
```

When you write a rule, it is compiled once into a JSON filter, by a language model if one is configured or by a fast offline pattern matcher if not. After that, matching is deterministic: no model runs when a question arrives. Rules are evaluated in priority order, higher priorities applied later so they can override lower ones, and a rule with `--expires` removes itself when it runs out. A rule can let a question through, change its urgency, mute it, snooze it, or answer it automatically.

Named groups of contacts live in `~/.config/sjbis/entities.toml`, and rules can refer to them by name:

```
[groups]
family = ["Alice", "Bob", "Mom", "Dad"]
```

`sjbis rule allow --agent Signal --from Alice,Bob --for-duration 2h` is a shortcut for "mute everything from this agent except these people, for a while".

> **Warning**
>
> No rule can answer an approval ask, whatever it matches. The rule engine drops any automatic answer for an approval before it reaches the store. Muting or snoozing an approval is still possible, and it means the agent will time out without authority, which is the safe direction.

The Optional AI Router

sjbis can use a language model in exactly two narrow roles:

- **Routing and ranking.** One short call per question to suggest a renderer (when the caller asked for `--guess-renderer`), predict urgency, and flag likely duplicates. The caller's explicit flags always win, and the model never rewrites the question.
- **Rule compilation.** One call when you write a rule.

Both are off unless the daemon has an API key for an OpenAI-compatible provider, and both fall back to deterministic behavior when the provider is down. A question marked `--privacy private` never goes to the model. Steady state, a question costs at most one model call, and often none.

The router can never lock a card. A question becomes an approval only because the caller asked with `--approval`; no model can turn an ordinary question into one, or one into an ordinary question.

Summary

You answer from three places: the keyboard-driven dashboard, an installable phone web app with push notifications, and a Pi console that no agent can draw on. Approvals appear on all of them

but are locked everywhere a browser alone is in control; the daemon refuses any unsigned answer. Rules, compiled once and matched deterministically, keep the queue short, and no rule can answer an approval.

That completes the surfacing half. Part II starts on proof: Chapter 4 opens up a signed grant to see what exactly gets signed.

PART II

Proving the Answer

An answer is only useful to the next agent if it can be checked without trusting the agent that relays it. This part covers `ta`: what a signed grant is, how to set up the keys that sign and verify it, how agents ask for and check authority, and what makes that authority transient.

CHAPTER 4

Grants, Signatures and Short IDs

In this chapter you will take a real signed grant apart: the record, its canonical bytes, the hash, the short ID and the signature. You will verify it three ways: with `ta`, with a dozen lines of Python, and with OpenSSL. The grant comes from the test fixtures in the `ta` repository, signed with a throwaway key that is committed next to it, so everything in this chapter works offline on your machine.

Getting the Fixtures

Clone the repository and build `ta`:

```
git clone https://example.com/transient-authority.git
cd transient-authority
cargo build -p ta
alias ta="$PWD/target/debug/ta"
```

The `fixtures/` directory holds the vectors that `ta` and the sjbis daemon both test against. The two programs implement the wire format independently, in different codebases, and these files are how they prove they agree byte for byte.

File	Holds
`approve.envelope.json`	An approval with a choice, a non-ASCII note and a `renews` link
`deny.envelope.json`	A single-use denial whose question contains `"`, a newline and a tab
`revoke.envelope.json`	A revocation of the approval
`signers.toml`	A registry pinning the throwaway key `dev@fixtures`
`dev-signer.pem`	The throwaway private key itself
`grant-v1.json`	All of the above plus expected verdicts and negative cases

> **Warning**
>
> `dev@fixtures` is public: its private key is in the repository. Never pin it in a registry you use

> for real. Anyone could sign anything with it.

The Envelope

Here is the approval, exactly as the daemon stores and serves it:


```
{
  "id": "ta-ucfuud",
  "record": {
    "answer": "plan B",
    "ask_id": "sjbis-AbCdEfGh",
    "asked_by": "agent-x",
    "decision": "approve",
    "expires_at": "2026-10-23T02:12:04Z",
    "issued_at": "2026-10-23T01:12:04Z",
    "kind": "grant",
    "note": "Yes, but only the two on the left — not the café ones ✓",
    "once": false,
    "question": "Do you want to fire the missiles?",
    "renews": "ta-3kp9ab",
    "signer": "dev@fixtures",
    "v": 1
  },
  "record_sha256":
   ↪  "a08b4a0fa51465cd15d57d446e9702d6265e3ec01f220f51babeafb83b64582c",
  "alg": "es256",
  "sig": "MEQCIBaldMHJH-
   ↪  dRHslf0VQJPcAOKsItQsQuwgJKgYzmTRkoiAiACgRWjbPFhMN7kX6zDUTYTSDmHoxqpeWE/0oj0Bo2Y7Q=="
}
```


Figure 4-1. A grant envelope: the signed record and four derived fields.

The envelope has two layers. `record` is the part that is signed. The other four fields are derived from it and exist so a reader can find and check it: `record_sha256` and `id` are computed from the record, `alg` names the signature scheme, and `sig` is the signature.

The Record

Every field of the record matters, so take them in turn.

`v` and `kind` Format version, always `1`, and the record kind: `grant`, `revoke` or `enroll`. This chapter is about grants; the other two kinds appear in Chapter 5 and Chapter 7.

`ask_id` The sjbis question this grant answers. It links the grant to the full context of the ask (detail, urgency, timing) without signing all of it.

`asked_by` The agent name as the asker passed it to sjbis. It is a *claim*: sjbis doesn't authenticate agents, so nothing proves which agent asked.

`question` The question, verbatim, exactly as the agent passed it. Never rewritten, never summarized. This is what you read before you signed, and what a verifier will compare against.

`decision` `approve` or `deny`. Both are signed, and both are relayable.

`answer` Optional. The value you chose, when the ask offered choices.

`note` Optional free text from you. It carries nuance the decision can't: "yes, but only the two on the left". Agents must read it, because it can narrow an approval.

`issued_at`, **`expires_at`** When you signed, and when the grant stops being authority. Both are inside the signed bytes, so an offline verifier enforces expiry too.

`once` Whether the grant is meant to be consumed by a single action.

`renews` Optional. The ID of an earlier grant that this one re-authorizes. Because it's signed, the link between old and new is verifiable.

`signer` The ID of the key that signed, as it appears in a signer registry. The method, such as Touch ID or a software key, is a property of the key in the registry, so it is proved by the signature rather than asserted by a field.

Optional fields that are absent are *omitted*, never `null`. Unknown fields make the record malformed, and so do timestamps in any form other than exactly `YYYY-MM-DDTHH:MM:SSZ`. That strictness is what lets two independent implementations produce the same bytes.

Canonical Bytes

A signature covers bytes, not JSON values. Two JSON documents can mean the same thing and differ in key order, whitespace or escaping, so `ta` defines one byte string per record: the RFC 8785 JSON Canonicalization Scheme (JCS). In practice, for these records, that means keys sorted, no insignificant whitespace, raw UTF-8 rather than `\u` escapes, and only the mandatory escapes (`\"`, `\\`, `\n`, `\t` and other control characters).

The canonical form of the approval begins:

```
{"answer":"plan B","ask_id":"sjbis-AbCdEfGh","asked_by":"agent-x","decision":
"approve","expires_at":"2026-10-23T02:12:04Z","issued_at":"2026-10-23T01:12:0
4Z","kind":"grant","note":"Yes, but only the two on the left — not the café o
nes ✓",...
```

Figure 4-2. The start of the canonical bytes, wrapped for the page. The real bytes are one line.

The note's em dash, accented letter and check mark are raw UTF-8 bytes. A JSON library is free to write them as `\u` escapes instead, and some do; that is the same JSON value, and canonicalization turns either spelling into the same bytes. The fixture includes the note precisely to catch implementations that get this wrong.

> **Warning**
>
> Always verify against the canonical form of the record *as you received it*, never against a typed copy you parsed and serialized again. If your parser drops an unknown field or normalizes a string, the signature you check would no longer be over the bytes that were signed, and a strict verifier would be reporting on a record that doesn't exist.

The Hash and the Short ID

`record_sha256` is the lowercase hex SHA-256 of the canonical bytes. The ID is derived from the same digest:

```
id = "ta-" + first 6 characters of base32(SHA-256(canonical bytes))
```

using RFC 4648 base32, in lowercase, without padding. Six base32 characters are 30 bits, so the ID is short enough to say aloud and to drop into a sentence an agent writes, and long enough that collisions are rare in one person's lifetime of approvals. When one does happen, the daemon refuses the second grant and the signing device signs again with a fresh `issued_at`, which changes every bit of the hash.

The ID is a name, not a security boundary. Everything that matters is checked against the full record and signature, which the ID merely points to.

The Signature

`alg` is always `es256`: ECDSA on the P-256 curve with SHA-256, over the canonical bytes. `sig` is the standard base64 (with padding) of the DER-encoded signature, and public keys in the registry are the standard base64 of the 65-byte uncompressed SEC1 point, `04 ‖ x ‖ y`.

These encodings were chosen because they are what the hardware produces. The Secure Enclave on a Mac or iPhone and a YubiKey's PIV applet all emit DER signatures and SEC1 points natively, so no device has to re-encode anything, and there's no conversion step that could go wrong.

> **Note**
>
> An ECDSA signature `(r, s)` has a twin, `(r, n - s)`, which verifies just as well. Some libraries insist on the low-S form. `ta` accepts both, and the fixtures include the high-S twin of the approval to keep every implementation honest about that. Because the ID is computed from the record and not the signature, both twins have the same ID.

Verifying With ta

`ta verify --record` checks an envelope file offline. The `--question` flag gives the question you believe was approved. The listings in this chapter set the clock to a moment when the fixture grant was still live:

```
$ ta verify --record fixtures/approve.envelope.json \
    --question "Do you want to fire the missiles?"
ta-ucfuud
  kind:       grant
  question:   Do you want to fire the missiles?
  decision:   approve
  answer:     plan B
  note:       Yes, but only the two on the left — not the café ones ✓
```

```
  asked_by:   agent-x (claimed by the asking agent, not proven)
  ask_id:     sjbis-AbCdEfGh
  issued_at:  2026-10-23T01:12:04Z
  expires_at: 2026-10-23T02:12:04Z (in 47m)
  once:       false
  renews:     ta-3kp9ab
  signer:     dev@fixtures (software, ta-core fixture key (throwaway, committed,
 ↪  never trust in production))
verdict: VALID (exit 0): valid (offline: revocation and claims not checked)
```

Figure 4-3. Offline verification of the fixture approval.

`ta` printed the record so you can read it, checked the hash, the ID and the signature against the key pinned for `dev@fixtures`, checked expiry against the clock, and compared the question. Exit code 0 means all of that passed. The phrase "offline: revocation and claims not checked" is honest: a file on disk can't tell you whether the grant was revoked after it was exported. You will see online verification, which does check, in Chapter 6.

Notice also the line `asked_by: agent-x (claimed by the asking agent, not proven)`. `ta` repeats that caveat every time it prints the field.

Now try the verifications that should fail. Paraphrase the question:

```
$ ta verify --record fixtures/approve.envelope.json --question "Fire the missiles"
ta-ucfuud
  kind:       grant
  question:   Do you want to fire the missiles?
  decision:   approve
  answer:     plan B
  note:       Yes, but only the two on the left — not the café ones ✓
  asked_by:   agent-x (claimed by the asking agent, not proven)
  ask_id:     sjbis-AbCdEfGh
  issued_at:  2026-10-23T01:12:04Z
  expires_at: 2026-10-23T02:12:04Z (in 47m)
  once:       false
  renews:     ta-3kp9ab
  signer:     dev@fixtures (software, ta-core fixture key (throwaway, committed,
 ↪  never trust in production))
verdict: QUESTION_MISMATCH (exit 6): question mismatch: signed record says "Do you
 ↪  want to fire the missiles?", you passed "Fire the missiles"
[exit 6]
```

Exit 6, and the message shows both strings. Remove every pinned key:

```
$ TA_SIGNERS=/dev/null ta verify --record fixtures/approve.envelope.json
ta: warning: signer registry /dev/null is empty or missing; nothing can verify
verdict: UNKNOWN_SIGNER (exit 5): unknown signer "dev@fixtures" (not in the pinned
 ↪  registry)
[exit 5]
```

Exit 5. Without a pinned key, nothing verifies; `ta` never falls back to a key from anywhere else. And let an hour pass:

```
$ ta verify --record fixtures/approve.envelope.json
ta-ucfuud
  kind:       grant
  question:   Do you want to fire the missiles?
  decision:   approve
  answer:     plan B
  note:       Yes, but only the two on the left — not the café ones ✓
  asked_by:   agent-x (claimed by the asking agent, not proven)
  ask_id:     sjbis-AbCdEfGh
  issued_at:  2026-10-23T01:12:04Z
  expires_at: 2026-10-23T02:12:04Z (expired 47m ago)
  once:       false
  renews:     ta-3kp9ab
  signer:     dev@fixtures (software, ta-core fixture key (throwaway, committed,
 ↪  never trust in production))
verdict: EXPIRED (exit 3): expired at 2026-10-23T02:12:04Z
[exit 3]
```

Exit 3. The signature is still perfectly good, but the authority has expired.

Verifying Without ta

The format is small enough to check with standard tools, which is the best evidence that it doesn't depend on `ta`'s implementation. With Python and no third-party packages:

```
import json, hashlib, base64

env = json.load(open("fixtures/approve.envelope.json"))
can = json.dumps(env["record"], sort_keys=True,
                 separators=(",", ":"), ensure_ascii=False).encode()
digest = hashlib.sha256(can).digest()

assert digest.hex() == env["record_sha256"]
assert "ta-" + base64.b32encode(digest).decode().lower()[:6] == env["id"]

open("msg.bin", "wb").write(can)
open("sig.der", "wb").write(base64.b64decode(env["sig"]))
```

> **Note**
>
> `json.dumps` with `sort_keys` matches JCS for these records only because every key is ASCII and the records contain no floating-point numbers. For general JSON, use a real JCS implementation.

Then check the signature with OpenSSL. A SEC1 point needs a fixed 26-byte header to become a DER public key that OpenSSL accepts:

```
PK=$(awk -F'"' '/pubkey/ {print $2}' fixtures/signers.toml)
{ printf '3059301306072a8648ce3d020106082a8648ce3d030107034200' | xxd -r -p
  echo "$PK" | base64 -d; } > pub.der
openssl pkey -pubin -inform DER -in pub.der -out pub.pem
openssl dgst -sha256 -verify pub.pem -signature sig.der msg.bin
```

The last command prints `Verified OK`.

Summary

A grant is a strict JSON record, canonicalized with RFC 8785, hashed with SHA-256, named by the first six base32 characters of that hash, and signed with ES256 in the encodings that Secure Enclave and YubiKey hardware produce natively. The question, decision, note and expiry are all inside the signed bytes. `ta verify --record` checks an exported grant offline, and the format is simple enough to check with Python and OpenSSL alone.

Chapter 5 sets up `ta` for real: a configuration, a signer of your own, and a registry pinned on every machine that verifies.

CHAPTER 5

Setting Up ta

In this chapter you will install `ta`, write its configuration, create a development signer, and pin that signer's public key in the three places that need it: your own registry, the daemon's, and every machine that verifies. At the end you will hand your agents the primer that tells them how to use it. The hardware-backed signers come in Chapter 8; a software key is enough to see every part of the flow work.

Installing

`ta` is a Rust workspace with two crates: `ta-core`, the library that holds the record types, canonicalization, signing and verification, and `ta`, the command-line tool. From a checkout:

```
cargo install --path crates/ta
cargo test        # no network needed
ta --version
```

Install it on every machine where an agent runs, and on every device that will sign. `ta` has no daemon of its own. It talks to the sjbis daemon, and that is the only always-on piece.

Configuration

`ta` reads `~/.config/ta/config.toml`. Every key is optional:

```
daemon_url     = "http://sjbis.example.com:7878"   # or TA_DAEMON / --daemon
signers_file   = "~/.config/ta/signers.toml"        # or TA_SIGNERS / --signers
my_signer      = "touchid@mac"                      # signer devices only
signer_backend = "external:/usr/local/bin/ta-se"    # signer devices only
```

The first two keys are all that an agent's machine needs: where the daemon is, and where the pinned registry is. The last two make a machine a *signer device*. `my_signer` is this device's ID in the registry, and `signer_backend` says how to sign:

Backend	Form	Key lives in
Software	`software:PATH`	A PKCS#8 PEM file on disk
External	`external:PATH`	A helper program, such as `ta-se` for the Secure Enclave

An external helper is any program that speaks a two-command protocol: print the public key, or read message bytes on standard input and print a base64 DER signature. That is how `ta` reaches hardware without linking against it, and how a future helper for a YubiKey or a cheap secure element would plug in.

> **Tip**
>
> Run every command with `--daemon`, `--signers` or `--config` to override the file for one invocation. The book's own listings are generated that way, with an empty home directory, so nothing from the build machine leaks into them.

The Signer Registry

The registry is a TOML file of pinned public keys:

```toml
[[signer]]
id     = "touchid@mac"
method = "touchid"
device = "MacBook Pro"
pubkey = "BJx3..."     # base64 SEC1 uncompressed P-256 point
added  = "2026-10-23T00:00:00Z"
```

Verification trusts only this file. When `ta verify` checks a grant, it looks up the grant's `signer` in your local registry and nowhere else. It never uses a key the daemon serves, even though the daemon has a registry of its own and will happily list it. That is what makes the daemon unable to forge a grant: to get a forged grant accepted, an attacker would have to change the file on *your* machine.

`ta signers list` shows what is pinned:

```
$ ta signers list
# fixtures/signers.toml
dev@fixtures            software  ta-core fixture key (throwaway, committed, never
↪  trust in production) added 2026-10-23T00:00:00Z  pubkey BO+oCq28nf9JtALL…
```

The method is one of `faceid`, `touchid`, `yubikey` or `software`. It is a property of the key, not of the grant, so a grant signed by `touchid@mac` is proved to be a Touch ID signature by the fact that it verifies against that key.

Your First Signer

Create a development key. On the machine that will sign:

```
mkdir -p ~/.config/ta && chmod 700 ~/.config/ta
ta signers keygen-dev ~/.config/ta/dev.pem
```

`keygen-dev` writes the private key with mode `0600`, prints the public key, and prints the two next steps. Pin the key in your own registry, using the public key it printed:

```
ta signers add --id dev@laptop --method software \
  --device "laptop dev key" --pubkey PUBKEY
```

and make this machine a signer device in `~/.config/ta/config.toml`:

```
daemon_url     = "http://sjbis.example.com:7878"
my_signer      = "dev@laptop"
signer_backend = "software:~/.config/ta/dev.pem"
```

> **Warning**
>
> A software key is a file. Anyone who can read it can sign as you, and so can any agent that runs as your user on that machine. Use it to learn the flow, and in Chapter 8 replace it with a key that requires your physical presence for every signature. Never pin a software key in the registries your agents use for production work unless it lives on a box no agent can reach.

Pinning Everywhere

A grant has to verify in three places, and each has its own copy of the registry.

The signing device itself. You just did this. `ta` refuses to sign as a `my_signer` it can't find pinned, and it checks every signature it makes against the pinned key before posting it.

The daemon. The daemon rejects grants from signers it doesn't know. Append the same `[[signer]]` entry to `~/.config/sjbis/signers.toml` on the daemon host, or to the path in `SJBIS_SIGNERS`. The daemon re-reads the file on every `POST /grants`, so there's no restart. A missing file is an empty registry, which refuses every grant as `unknown_signer`; a file that doesn't parse makes grant posts fail with HTTP 500 until you fix it. The daemon's copy exists only to reject junk early. Nobody relies on it for trust.

Every verifier. Every machine where an agent will run `ta verify` needs the entry in its own `~/.config/ta/signers.toml`. Otherwise grants from that signer verify as "unknown signer", exit 5, which is the safe failure.

```
ta signers add --id dev@laptop --method software --pubkey PUBKEY
```

Distributing the registry

The design calls for a chain of custody: after you pin the first signer by hand, every later signer would be added by an `enroll` record, signed by an existing signer, which other machines pull from the daemon and verify before they update their own files. The record kind exists in the wire format, and the daemon stores and serves enroll records.

The client side does not exist yet. Today `ta signers add` is an unsigned local edit, and says so: "unsigned local add; signed enroll records are not implemented yet". Until it lands, distribute the registry the way you distribute any other piece of trusted configuration, through your dotfiles repository, configuration management, or by hand, and treat the file with the same care as an SSH `authorized_keys` file. Whoever can write it decides whose signatures your agents will believe.

Checking the Setup

On a verifier, `ta prime` prints the live configuration at the top of the agent primer. With nothing pinned, it says so loudly:

```
  Daemon:  http://sjbis.example.com:7878
  Signers: ~/.config/ta/signers.toml (NONE pinned: nothing will verify)
```

On a signer device, the quickest end-to-end check is to ask yourself a question and approve it. You will do exactly that at the start of Chapter 6.

The Agent Primer

Agents don't read books. Like `sjbis prime`, `ta prime` prints a primer written for an agent: when to use `ta ask` rather than `sjbis ask`, the exit codes, the rule that timeouts carry no authority, and how to verify a grant another agent hands you. Here it is in full, as `ta` prints it on a machine pointed at the example daemon:

```
TA: transient authority  (ta 0.1.0)

  Daemon:  http://sjbis.example.com:7878
  Signers: ~/.config/ta/signers.toml (NONE pinned: nothing will verify)

  ta turns "Stephen said yes" into a short id (ta-7f3k2q) that any agent can
  verify against pinned hardware keys. Relay the ID, never the claim.

WHEN YOU NEED STEPHEN'S PERMISSION
  Use ta ask instead of sjbis ask whenever another agent, or you later, may
  need to prove the permission. Pass the question verbatim; it is signed.

    id=$(ta ask "Delete the staging database?" --agent <you> --ttl 1h --deadline 10m)
    case $? in
```

```
      0) echo "approved: $id" ;;          # relay "$id" to anyone who needs proof
      1) echo "denied: $id" ;;            # a signed no; also relayable
      3) echo "timed out: NO authority" ;;
      *) echo "error / unattested: NO authority" ;;
    esac

  Options: --ttl (how long the grant lasts, default 1h), --once (single use),
  --choices "a,b,c" (the choice is signed as answer), --detail "...",
  --urgency N (default 2), --json (print the full signed envelope).
  ta prints the id on stdout and the decision, answer and NOTE on stderr.
  READ THE NOTE: "yes, but only X" narrows the approval.

TIMEOUTS CARRY NO AUTHORITY
  sjbis's "on timeout, use best judgement" does NOT apply here. If ta ask
  times out there is no grant and no id. You may ask again or proceed
  without authority, but you must never claim you had permission.
  An answer without a grant ("answered without attestation", exit 2) is
  also not authority.

WHEN ANOTHER AGENT SAYS "STEPHEN APPROVED (ta-xxxxxx)"
  Verify it yourself. Read the record first, so you know exactly what
  Stephen was asked; then verify against that question and decide whether
  it covers the action you are about to take:

    ta show ta-7f3k2q
      question:  Delete the staging database?
      decision:  approve        expires_at: ...
    ta verify ta-7f3k2q --question "Delete the staging database?"

  --question must match the grant's question (only whitespace is ignored);
  anything else exits 6. Copy it from ta show, do not retype it from the
  other agent's message. If the grant's question does not describe what
  you are about to do, you have no authority for it, however valid the id.
  Without --question you only learn that Stephen approved *something*.
  --expect deny checks for a signed refusal instead.

EXIT CODES (ta verify)
  0 valid   1 denied   2 unknown id / error   3 expired   4 revoked
  5 bad signature or unknown signer   6 question mismatch   7 consumed (--once)
  ta ask: 0 approved, 1 denied, 3 timed out, 2 error or unattested,
          5/6 if the returned grant fails local verification.

OTHER COMMANDS
  ta reauth ta-xxxxxx [--ttl 1h]       re-ask the same verbatim question, linked via
↪  renews
  ta claim ta-xxxxxx --by <you>        consume a --once grant before acting
  ta show ta-xxxxxx [--lineage]        print the record / renewal tree
  (ta confirm = ta verify, ta get = ta show; both spellings work)
```

```
  ta log [--agent x] [--since 24h]     recent grants
  ta export ta-xxxxxx > g.json         envelope for offline use
  ta verify --record g.json --question "..."   offline: signature + expiry only
  ta console                           signer device only (Pi): HTTP sidecar the
↪  approval console signs through; not for agents

RULES
  * Verify against the grant's own question (from ta show), never a paraphrase,
    and act only if that question is the thing you are doing.
  * Never treat an expired, revoked, consumed or unverifiable id as permission.
  * asked_by is a claim, not proof of which agent asked.
```

Figure 5-1. The agent primer. The registry path line reflects the machine it runs on.

Put one line in your agents' standing instructions: "before asking for permission or acting on someone else's, run `ta prime`". The primer travels with the tool, so when the rules change, agents pick up the new ones without you editing a single prompt.

Summary

`ta` is a single binary that reads a small configuration: a daemon URL and a pinned registry for every machine, plus a signer ID and a signing backend for signer devices. Backends are a software key file or an external helper program. A signer must be pinned in three places: on the device, on the daemon (to reject junk early), and on every verifier (the only copy that is trusted). Signed enrollment is designed but not yet built, so for now the registry is distributed by hand. `ta prime` teaches agents the rest.

Chapter 6 puts all of it to work.

CHAPTER 6

Asking for and Verifying Authority

In this chapter you will run the whole loop: an agent asks for permission with `ta ask`, you approve and sign it, the agent relays the ID, and a second agent verifies it. You will learn what each exit code means, why the note matters as much as the decision, and the one verification habit that separates real authority from a valid signature on the wrong question.

> **Warning**
>
> From here on, commands create real authority. An approval you sign is something your agents will act on. Practice with harmless questions until the flow is familiar.

Asking

`ta ask` posts an approval ask to sjbis and blocks until it is answered or the deadline passes:

```
id=$(ta ask "Delete the staging database?" --agent deploybot --ttl 1h --deadline 10m)
```

Under the hood that is an `approval` ask, blocking, with a deadline, the requested grant lifetime, and the question passed verbatim. The two durations mean different things, and it's worth getting them straight now:

`--deadline` How long the agent waits for *you*. Default 10 minutes, which is also the daemon's cap on a single blocking wait.

`--ttl` How long the signed grant can be relied on *once it exists*. Default one hour. You can override it when you answer.

The rest of the flags are what you would expect: `--choices "a,b,c"` (the chosen value is signed as the grant's `answer`), `--detail` for context that is shown to you but not signed, `--urgency`, `--once` for a single-use grant, and `--json` to print the whole envelope instead of just the ID.

While it waits, `ta ask` tells you on standard error what it is doing:

```
ta: asked sjbis-AbCdEfGh; waiting until 2026-11-07T15:20:00Z for Stephen to
answer on a trusted device
```

Example output; the ask ID and time will differ.

If the connection to the daemon drops while it waits (a daemon restart, a flaky network), `ta ask` re-polls. The question lives on the daemon, so a dropped connection is never a lost answer.

Answering

On a signer device, `ta approve` lists the open approval asks and walks you through one:

```
$ ta approve
[1] sjbis-AbCdEfGh  deploybot           2m  Delete the staging database?
answer which? [number]: 1

ask:       sjbis-AbCdEfGh
from:      deploybot (claimed, not proven)
question (verbatim, this is what you sign):
  | Delete the staging database?
approve or deny? [a/d]: a
note (optional, enter to skip): only after the 17:00 backup finishes
ttl [3600s]:

record to sign:
{"ask_id":"sjbis-AbCdEfGh","asked_by":"deploybot","decision":"approve",...}
sign this approve as dev@laptop? [y/N]: y
ta-k3v9qa
ta: signed and posted ta-k3v9qa (approve)
```

Example session (record abbreviated).

Read what `ta approve` shows you, because it was chosen carefully. The requester is labeled "claimed, not proven". The question is printed verbatim, line by line, under the words "this is what you sign". Before you confirm, you see the canonical record itself. With a Touch ID signer, the confirmation is followed by a fingerprint prompt; with the phone or the console it is a tap or a key press. Chapter 8 covers each surface.

On signing, `ta` checks its own signature against your pinned key before it posts. A device whose key doesn't match its registry entry refuses to post rather than send a grant that nobody could verify.

What the Agent Gets Back

When you sign, the daemon verifies the grant against its copy of the registry, stores it, marks the question answered with `via = "ta:dev@laptop"`, and wakes the waiting `ta ask` with the full envelope. `ta ask` then verifies the grant *itself*, against its own registry and the question it asked, before it believes it. It prints the ID on standard output and the rest on standard error:

```
ta-k3v9qa
ta: APPROVED by dev@laptop (software), expires 2026-11-07T16:12:40Z
ta: note: only after the 17:00 backup finishes   <- read this; it may narrow the
 ↪  decision
```

Example output.

The exit code carries the decision:

Exit	`ta ask` result	Authority?
0	Approved	Yes, within the note and until expiry
1	Denied	A signed no, relayable
3	Timed out	**None**
2	Error, dismissed, cancelled, or answered without a grant	**None**
5, 6	The returned grant failed local verification	**None**

A complete agent-side step looks like this:

```
id=$(ta ask "Delete the staging database?" --agent deploybot --ttl 1h)
case $? in
  0) echo "approved: $id" ;;            # relay "$id" to anyone who needs proof
  1) echo "denied: $id" ;;              # a signed no; also relayable
  3) echo "timed out: NO authority" ;;
  *) echo "error or unattested: NO authority" ;;
esac
```

Read the note

The note is part of the signed record, and it is the most common way you will narrow an approval. "Approve" with the note "only after the 17:00 backup finishes" does not authorize deleting the database at 16:30. `ta` prints the note with an arrow pointing at it for exactly that reason. An agent that acts on the exit code and ignores the note has not understood the grant.

Timeouts Carry No Authority

This is the rule that most distinguishes `ta` from ordinary sjbis use. An ordinary sjbis question that times out tells the agent "use your judgment". An approval that times out tells the agent nothing at all:

> timed out: no grant, no ID, no authority. Silence is never consent.

The agent may ask again, later or with better detail. It may proceed without authority, if what it is doing doesn't actually require it. What it must never do is claim, to you or to another agent, that it had permission.

The same goes for every other way an approval can end without a grant: dismissed, cancelled, or answered by something that couldn't sign. In the current design the daemon refuses unsigned answers to approvals outright, so "answered without attestation" should never happen. `ta ask`

checks anyway, and reports it as exit 2 with no authority, so that a future change on the daemon side can't silently turn an unsigned answer into a yes.

Relaying

The reason for all of this is the next step. The agent passes the ID along in whatever form agents use to talk: a message, a comment on a merge request, a line in a handoff file.

> Cleanup of staging is approved (ta-k3v9qa). Wait for the 17:00 backup.

That is the whole relay. The ID is enough, because the receiver will not take the sender's word for anything else.

Verifying: Show, Then Verify

The receiving agent does two things, in this order.

First, read the record. `ta show` fetches the grant and prints it, so the receiver learns exactly what you were asked, not what the sender says you were asked:

```
$ ta show ta-k3v9qa
ta-k3v9qa
  kind:       grant
  question:   Delete the staging database?
  decision:   approve
  note:       only after the 17:00 backup finishes
  asked_by:   deploybot (claimed by the asking agent, not proven)
  ...
  status:     not revoked, unclaimed
```

Example output (abbreviated).

Then verify against that question. Copy the question from the record, not from the sender's message:

```
ta verify ta-k3v9qa --question "Delete the staging database?"
```

Online verification fetches the envelope from the daemon, checks it against the pinned registry, asks the daemon whether it has been revoked or claimed, compares the question, checks expiry, and exits with a verdict:

Exit	Verdict	Meaning
0	valid	Authentic, matching question, not expired, revoked or consumed
1	denied	Authentic, but the decision is not the one you required

Exit	Verdict	Meaning
2	unknown	The daemon has no such ID, or an error occurred
3	expired	Authentic, but past `expires_at`
4	revoked	A revocation targets this grant
5	bad signature	Hash, ID or signature wrong, or the signer isn't pinned
6	question mismatch	The signed question is not the one you passed
7	consumed	A single-use grant that has already been claimed

When more than one applies, the most serious wins: a forgery is reported as a forgery even if it is also expired, and a revoked grant is reported as revoked even if it has also expired.

> **Warning**
>
> The last step is a judgment, and no exit code makes it for you. A grant for "Delete the staging database?" is valid authority to delete the staging database. It isn't authority to delete the production one, however valid the ID. Before acting, the verifying agent must decide whether the question it just verified describes the action it is about to take. If it doesn't, the agent has no authority, whatever `ta verify` printed.

Why `--question` matters

Without `--question`, `ta verify` can only tell you that you approved *something*. It says so:

```
warning: no --question given; check that the question above is the one you are acting
 ↪  on
```

With it, the comparison happens inside `ta`, so an agent can't skip it by accident or get it subtly wrong. The comparison ignores differences in whitespace (runs of spaces, tabs and newlines compare equal), and nothing else: case, punctuation and wording all count. That is strict on purpose. A paraphrase is exactly how scope gets lost in a relay, and the answer to a mismatch is to read the record again, not to loosen the comparison.

Verifying a denial

Denials are relayable too. "You said no to this at ta-qo2xz7" is useful to an agent that is about to try again. By default `ta verify` requires an approval and reports a denial as exit 1; `--expect deny` checks for a signed refusal instead and reports it as valid:

```
$ ta verify --record fixtures/deny.envelope.json
ta-qo2xz7
```

```
  kind:       grant
  question:   Deploy "v2" to prod?
It touches        billing.
  decision:   deny
  asked_by:   deploybot (claimed by the asking agent, not proven)
  ask_id:     sjbis-ZyXwVuTs
  issued_at:  2026-10-23T01:20:00Z
  expires_at: 2026-10-23T01:30:00Z (in 5m)
  once:       true
  signer:     dev@fixtures (software, ta-core fixture key (throwaway, committed,
 ↪  never trust in production))
verdict: DENIED (exit 1): decision is deny, expected approve
[exit 1]
```

Figure 6-1. A signed denial, verified with the default expectation.

Choices

When an ask offers choices, the one you pick is signed as the grant's `answer`:

```
id=$(ta ask "Which migration plan for the orders table?" \
       --agent dbbot --choices "online,offline,postpone")
```

A verifier that needs a particular choice reads it from the record: `ta show --json`, or `ta verify --json`, which includes the full record along with the verdict. Approving with the choice `postpone` is a different decision from approving with `online`, and the verifier has to check which one it got.

Summary

`ta ask` posts an approval to sjbis with a deadline for you and a lifetime for the grant, and exits 0 only for a signed approval that it has verified itself. Timeouts and every other unsigned ending carry no authority. You answer on a signer device that shows you the verbatim question and the exact record before it signs. Agents relay only the ID. The receiver reads the record with `ta show`, verifies with `ta verify --question` using the question it just read, and then judges whether that question covers what it is about to do.

Chapter 7 covers what happens over time: expiry, renewal, revocation and single-use grants.

CHAPTER 7

Expiry, Renewal, Revocation and Single Use

In this chapter you will work with the parts of `ta` that make authority *transient*: choosing a lifetime, renewing a grant without retyping its question, following a chain of renewals, revoking a grant you regret, consuming a single-use grant, and exporting a grant for a verifier that can't reach the daemon. Each one closes a gap that hearsay leaves open.

Lifetimes

Every grant expires. The agent requests a lifetime with `--ttl` (default one hour), and you can override it when you answer: `ta approve` prompts for it, the Pi console cycles through 10 minutes, 1 hour and 24 hours with the `T` key, and the phone offers the same choices. The value you pick is what gets signed.

Because `expires_at` is inside the signed record, expiry needs no daemon to enforce it. An exported grant checked on an air-gapped machine still stops working at the right moment.

Pick lifetimes by asking how long the *decision* stays true, not how long the work will take:

Decision	Reasonable lifetime
"Run this one migration now"	10 minutes, with `--once`
"Deploy today's release"	A few hours
"You may push to the release branch this week"	Probably too long for one grant; renew it

Short lifetimes are cheap, because renewal is cheap. They also bound the one real gap in the design, which is that offline verifiers can't see revocations. More on that below.

> **Note**
>
> The lifetime is separate from the sjbis deadline. The deadline is how long the agent waits for you to answer; the lifetime is how long your answer stays good. A grant asked with `--deadline 2m --ttl 24h` must be answered within two minutes and then lasts a day.

Renewal with `ta reauth`

An expired grant carries no weight, but the decision behind it can be proposed again. `ta reauth` re-asks the question of an earlier grant:

```
new=$(ta reauth ta-k3v9qa --ttl 1h)
```

Three things make a reauth different from a fresh `ta ask`:

1. **The question is copied from the signed record**, not retyped by the agent. It's byte-identical to the original by construction, so an agent can't use a "renewal" to slip in a slightly broader question.
2. **You see the history.** The ask's detail says what you decided last time, with which signer, when, and with what note, and `ta approve` prints the earlier decision under the question. You are confirming or changing a known decision, not reading a fresh one.
3. **The new grant links to the old one.** Its record carries `renews: "ta-k3v9qa"`, inside the signed bytes.

You can approve or deny, and change the note and lifetime. A denial on reauth is a signed, linked "no longer", and it is just as relayable as the original yes. Reauth works before expiry too, as a renewal; the old grant is never edited, because nothing is ever edited after it is signed.

Lineage

Grants linked by `renews` form a tree, much like Git commits: each child points to its parent. One grant can be renewed more than once, so a parent can have several children, and each child is independently valid. `ta show --lineage` walks to the root and prints everything below it:

```
$ ta show ta-p2m7rd --lineage
...
lineage (root first; edges point to the renewed grant; each grant is verified on its
↪  own):
  ta-k3v9qa  approve 2026-11-07T15:12:40Z  by dev@laptop  expired  sig ok
  └ ta-p2m7rd  approve 2026-11-07T16:20:05Z  by touchid@mac  expires in 52m  sig ok
↪  <- this
```

Example output.

> **Warning**
>
> Lineage is context, not authority. `ta verify` checks only the grant it is given. An old link doesn't make a new grant more valid, and a new grant doesn't revive an old one. If an agent hands you `ta-k3v9qa` and says "it was renewed", ask for the new ID and verify that.

Revocation

Changed your mind? On a signer device:

```
ta revoke ta-k3v9qa --reason "backup failed; hold the cleanup"
```

A revocation is not a flag the daemon flips. It is a signed record of its own, of kind `revoke`, naming the target grant and your reason:

```
$ ta verify --record fixtures/revoke.envelope.json
ta-lrtcyb
  kind:       revoke
  target:     ta-ucfuud
  reason:     changed my mind
  issued_at:  2026-10-23T01:22:30Z
  signer:     dev@fixtures (software, ta-core fixture key (throwaway, committed,
 ↪  never trust in production))
verdict: VALID (exit 0): valid (offline: revocation and claims not checked)
warning: no --question given; check that the question above is the one you are acting
 ↪  on
```

Figure 7-1. The fixture revocation, verified offline. A revoke record has no question, which is why `ta` warns that none was checked.

The daemon stores the revocation and marks the target. From then on, online verification of the target exits 4, and tells you who revoked it, when, and why:

```
verdict: REVOKED (exit 4): revoked by ta-lrtcyb (signed by touchid@mac at
2026-11-07T16:01:12Z; reason: backup failed; hold the cleanup)
```

Example output.

`ta` fails closed here. If the daemon says a grant was revoked but the revocation record can't be fetched or doesn't verify, the grant is still treated as revoked. A daemon can hide a revocation, which is a limit you will meet in Chapter 9, but it can't cheaply turn a good grant into a bad one either: the worst a fake "revoked" can do is deny authority.

A few rules worth knowing:

- **Revocation doesn't cascade.** Revoking a grant does not revoke its renewals. To kill a whole line, revoke each grant in it.
- **Any pinned signer can revoke.** Your phone can revoke a grant your Mac signed. That's deliberate: the device you have with you is the one you will use.
- **Revocation is online only.** An offline verifier checking an exported file sees signature and expiry, never revocation. Short lifetimes bound that gap.

Single-Use Grants

Some approvals should be spent exactly once: "run this migration", "send this email". Ask for one with `--once`:

```
id=$(ta ask "Run migration 0042 on prod?" --agent dbbot --once --ttl 15m)
```

Before acting, the agent that performs the action claims the grant:

```
ta verify "$id" --question "Run migration 0042 on prod?" && \
  ta claim "$id" --by dbbot && \
  ./migrate 0042
```

The first claim succeeds and prints `claimed ta-... by dbbot`. A second claim fails with exit 7, "already consumed", and from then on `ta verify` reports the grant as consumed, exit 7, naming who claimed it and when. Claiming a grant that was not single-use fails with exit 2: there is nothing to claim.

> **Warning**
>
> A claim is an unsigned mark on the daemon. It protects against accidental reuse by cooperating agents: two agents that both received the ID and both verify before acting. It doesn't protect against a hostile agent that skips the claim, or against the daemon itself. For v1 that is an accepted limit, and the reason `--once` grants should also have short lifetimes.

Exporting for Offline Verification

`ta export` prints the self-contained envelope:

```
ta export ta-k3v9qa > grant.json
```

That file is everything an offline verifier needs, alongside a pinned registry:

```
ta verify --record grant.json --question "Delete the staging database?"
```

Offline verification checks the hash, ID, signature, question and expiry, and says plainly what it couldn't check: "offline: revocation and claims not checked". Use it for archives, audits, air-gapped machines, and for attaching proof to a change record.

Looking Back

`ta log` lists recent grants, newest first:

```
ta log --since 24h
ta log --agent deploybot --since 7d
```

Each line shows the ID, issue time, decision, asking agent and question, with flags for `expired`, `revoked` and `claimed`. Every line is verified against your registry as it's printed, and a grant that doesn't verify is shown as `UNVERIFIED` rather than silently left out.

Summary

Every grant expires at a time inside its signed bytes. `ta reauth` renews a decision by re-asking the verbatim question from the stored record and linking the new grant with a signed `renews`; lineage is context, never authority. Revocations are signed records, checked online, failing closed, and not cascading. `--once` grants are consumed with an advisory claim. `ta export` and `ta verify --record` carry a grant to verifiers that can't reach the daemon, at the cost of not seeing revocations.

Part III turns to operation. Chapter 8 replaces the development key with signers that need you to be physically present.

PART III

Running It

A software key on a laptop proves the flow works, but not that you were there. This part replaces it with signers that need your finger, your phone or a key press on a dedicated box, and then states plainly what the whole system does and does not protect against.

CHAPTER 8

Trusted Surfaces: Mac, Phone and Pi Console

In this chapter you will set up the three signers that make a grant mean "you were there": a Secure Enclave key on your Mac that needs Touch ID for every signature, a key in the phone web app, and a Pi approval console that signs through a loopback sidecar. All three produce exactly the same wire format, so nothing on the verifying side changes as you add them.

This chapter needs hardware: a Mac with Apple silicon or a T2 chip, an iPhone, and, for the console, a Raspberry Pi 4 or 400 with a small screen and an Elgato Stream Deck. Each section stands alone; set up the ones you have.

What Makes a Surface Trusted

Two properties, and every surface in this chapter is judged against them:

1. **The key requires your presence.** A signature happens only after a physical act that an agent can't perform: a fingerprint, unlocking the phone, a key press on a box no agent can reach.
2. **You read the question somewhere the agent can't write.** An agent that controls the screen you read from can show you one question and get you to sign another. The surface that shows you the record should be one the asking agent has no way to draw on.

No surface is perfect on both. The table at the end of the chapter compares them honestly.

The Mac: Touch ID through `ta-se`

`ta-se` is a small Swift program that holds a P-256 key in the Mac's Secure Enclave and signs with it. `ta` calls it as an external helper: `ta-se pubkey` prints the public key, and `ta-se sign` reads the canonical bytes on standard input and prints a base64 DER signature, after a Touch ID prompt.

Building and creating the key

You need macOS 13 or later and the Swift toolchain (Xcode or the Command Line Tools):

```
cd ta-se
make install          # builds, ad-hoc code-signs, installs ~/.local/bin/ta-se
ta-se info            # Secure Enclave and biometry status
ta-se keygen          # creates the key; prints its public key
```

By default the key's access control is "a currently enrolled fingerprint": no password fallback. Two consequences follow:

- On a Mac without built-in Touch ID, such as a Mac Studio or Mac mini, you need a Magic Keyboard with Touch ID, paired, with a fingerprint enrolled. Without one, `keygen` fails and tells you to use `--allow-password`, which accepts Touch ID when available and the login password otherwise. That still requires someone at the keyboard, but anyone who knows your password qualifies.
- Enrolling or removing **any** fingerprint permanently invalidates a biometry-only key. When that happens, run `ta-se keygen --force` and pin the new public key. Grants signed with the old key stay valid; it is the key, not its past signatures, that is gone.

> **Note**
>
> A command-line tool without a provisioned app bundle can't store a Secure Enclave key in the keychain, so `ta-se` stores the key *wrapped*: the Secure Enclave generates it and hands back an opaque blob that only that chip can use, which `ta-se` keeps in `~/.config/ta-se/` with mode `0600`. The private key never exists in the clear, the blob is useless on any other Mac, and every signature still enforces Touch ID. Losing the file loses the key.

Pinning it and using it

Configure the Mac as a signer device:

```
# ~/.config/ta/config.toml
daemon_url     = "http://sjbis.example.com:7878"
my_signer      = "touchid@mac"
signer_backend = "external:~/.local/bin/ta-se"
```

then pin it here, on the daemon and on every verifier. With an external backend configured, `ta signers add` reads the public key from the helper:

```
ta signers add --id touchid@mac --method touchid --device "MacBook Pro"
```

From now on `ta approve` ends with a Touch ID prompt. The terminal shows you the canonical record before you touch the sensor.

> **Warning**
>
> Any process running as you can ask `ta-se` for a signature, and the system prompt shows a reason chosen by the caller, not the bytes being signed. Touch ID proves that *you* were

present; it does not prove *what* you saw. The protection is the habit of reading the record that `ta approve` prints, and touching the sensor only when you just asked to sign. An unexpected Touch ID prompt is a reason to stop, not to touch.

The Phone: Signing in the Web App

The phone web app from Chapter 3 can sign too. On first run it generates a P-256 key with the browser's WebCrypto API, marked non-extractable, and keeps it in IndexedDB. Settings shows the key's signer ID (`phone@iphone`), its public key, and the exact `ta signers add ...` command to pin it.

Until the key is pinned in the daemon's registry, the app says so and keeps the approve and deny buttons disabled; the daemon would refuse its grants anyway. The app's registration with the daemon reports `enrolled` once the signer ID is pinned with exactly that public key.

Once pinned, answering an approval on the phone does what `ta approve` does. It builds the grant record from the verbatim question, the lifetime you chose, your note, and the ask's `once` and `renews`, then signs it, posts the envelope to the daemon, and shows you the new `ta-` ID. Grants from the phone verify with `ta verify` anywhere the key is pinned. The repository's browser regression test signs a real envelope with WebCrypto and checks it with the Rust verifier, so the two implementations stay in step.

> **Note**
>
> A WebCrypto key is non-extractable to the page's scripts, but it is not held in the iPhone's Secure Enclave, and signing doesn't trigger Face ID. The presence check is that the phone is unlocked and in your hand. The design's stronger phone signer is a native app with a Secure Enclave key that needs Face ID for each signature. That lives in the sjbis iOS app, outside the `ta` repository, and plugs into the same registry with method `faceid` when it ships.

The phone is the surface you will use most, because it is the one in your pocket when a push arrives. Pick its lifetime defaults with that in mind: grants signed on a lock-screen tap should be short.

The Pi Approval Console

The console is a Raspberry Pi with a small screen and a Stream Deck, running Chromium in kiosk mode on the daemon's `/console` page. The page can't sign, and shouldn't: it is a web page, and anything the page can do, a script on the page could do. Instead, approvals go to `ta console`, a sidecar on the Pi that listens on `127.0.0.1:7879` and signs with the Pi's own key.

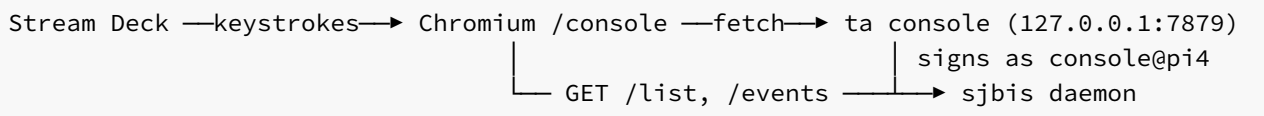

Figure 8-1. The console: the page shows and collects, the sidecar signs.

Setting it up

Build `ta` for the Pi, either natively or by cross-compiling on a Mac:

```
rustup target add aarch64-unknown-linux-musl
cargo zigbuild --release --locked -p ta --target aarch64-unknown-linux-musl
scp target/aarch64-unknown-linux-musl/release/ta pi4:.local/bin/ta
```

On the Pi, create the console's key and configuration:

```
mkdir -p ~/.config/ta && chmod 700 ~/.config/ta
ta signers keygen-dev ~/.config/ta/console.pem
```

```
# ~/.config/ta/config.toml on the Pi
daemon_url     = "http://sjbis.example.com:7878"
my_signer      = "console@pi4"
signer_backend = "software:~/.config/ta/console.pem"
```

Pin it on the Pi, on the daemon and on every verifier, as in Chapter 5. The sidecar refuses to start if its own signer isn't pinned locally.

Then run the sidecar and the kiosk as systemd user services. The repository's `pi/` directory has unit files for both, plus a helper that drives the Stream Deck:

```
cp pi/ta-console.service pi/kiosk.service ~/.config/systemd/user/
systemctl --user daemon-reload
systemctl --user enable --now ta-console.service kiosk.service
loginctl enable-linger "$USER"
curl -s http://127.0.0.1:7879/health
# {"ok":true,"signer":"console@pi4","daemon":"http://sjbis.example.com:7878"}
```

The console page polls the sidecar's health every ten seconds. While it fails, the page says "signing sidecar offline" and disables approve and deny; snooze and dismiss still work.

The sidecar's contract

The page sends `POST /approve` with the ask ID, the decision, the chosen answer and note, and the lifetime. The sidecar then does what `ta approve` does: fetches the ask from the daemon, signs *the daemon's copy* of the verbatim question, checks its own signature, and posts the grant. The page never sends a question to sign, so a tampered page can't get the sidecar to sign words you didn't see on the daemon. On success the page shows the new `ta-` ID in large type for a few seconds.

Two guards keep the sidecar from becoming a signing service for anything on the network:

- **It binds to loopback.** `--listen` with any other address logs a warning, because then anyone who can reach the port can ask the Pi to sign.

- **It checks the browser's `Origin`.** Only the daemon's own origin is allowed by default; add others with `--allow-origin` if you serve the page from somewhere else, such as behind a reverse proxy.

```
$ ta console --help
Signer device: loopback HTTP sidecar that signs for the Pi console page

Usage: ta console [OPTIONS]

Options:
      --config <PATH>          Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --listen <ADDR:PORT>     Address to listen on. Anything but loopback must be
↪  asked for explicitly [default: 127.0.0.1:7879]
      --allow-origin <ORIGIN>  Origin allowed to call the sidecar from a browser
↪  (repeatable; default: the origin of daemon_url)
      --daemon <URL>           sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --signers <PATH>         Pinned signer registry (overrides config signers_file)
↪  [env: TA_SIGNERS=]
  -h, --help                   Print help
```

The Stream Deck

The Stream Deck is a USB keypad with a small screen on every key. On the console, a helper draws a label on each key and types each press as Ctrl+Alt+Shift plus the key through a virtual keyboard, which works under both Wayland and X11. The page treats those combinations as hard keys that act even while the note field has focus. The layout puts choices across the top, navigation and the lifetime in the middle, and YES (green), NO (red), NOTE, SNOOZE and DISMISS along the bottom.

The signer ladder

A software key on the Pi is the first of three rungs, from weakest to strongest, all producing the same wire format:

1. **A software key** (`method = "software"`), as above. The presence check is a key press on a box no agent can reach. The key is a file, so anyone with the SD card or root on the Pi has it. Mitigate with key-only SSH and no SSH access from any host your agents run on.
2. **A secure element**, such as an ATECC608 breakout on the Pi's I2C bus, for a few dollars: a non-extractable P-256 key that signs ES256 natively. It needs a small helper speaking the same `pubkey` / `sign` protocol as `ta-se`. A TPM module or a FIDO2 key works the same way.
3. **A YubiKey** in PIV mode with touch policy "always": the key blinks, you touch it, it signs.

Climbing a rung is a new registry entry and a signed revocation of the old one. The console page, the daemon and every agent stay exactly as they are.

Comparing the Surfaces

Surface	Key	Presence	Who can draw the screen
Mac, `ta approve`	Secure Enclave, wrapped blob	Touch ID per signature	Any process running as you
Phone web app	WebCrypto, non-extractable	Unlocked phone in hand	Only the daemon's page
Pi console, rung 1	Software key file	Key press on an isolated box	Only the daemon's page
Pi console, rung 3	YubiKey PIV	Touch on the YubiKey	Only the daemon's page

The Mac has the strongest key and the weakest screen. The console, once it reaches the YubiKey rung, is strong on both, which is why the design calls it the cleanest answer to "the prompt surface must be out of band".

Summary

Every trusted surface signs the same record in the same format; they differ in how they check presence and in who controls the screen you read. On the Mac, `ta-se` keeps a Secure Enclave key that needs Touch ID for each signature. The phone web app signs with a non-extractable WebCrypto key and won't sign until the daemon has pinned it. The Pi console signs through a loopback sidecar that only ever signs the daemon's copy of the question, and it can climb from a software key to a secure element or YubiKey without anything else changing.

Chapter 9 puts it all together: what this system defends against, and what it doesn't.

CHAPTER 9

The Threat Model, Honestly

In this chapter you will see who can do what in a fleet that runs sjbis and `ta`: what each party can forge, hide or misreport, which defenses are cryptographic and which are habits, and which known gaps were accepted on purpose. You will finish with the decisions that are still open, so you know where the design might move.

The Bar

The design's own words set the bar: "better than agentic hearsay", not "NSA-grade". The adversary in mind is not a remote attacker with a zero-day. It is an agent that is confused, overconfident, or acting on a garbled handoff; occasionally an agent that has been prompt-injected into claiming something it shouldn't; and the plain decay of information as it passes through several hands. Against that adversary, the goal is that the honest path is the easy path, and every dishonest or mistaken claim fails loudly and specifically.

Who Can Do What

Signer devices are the only parties that can create a valid grant. That is the root of trust: private keys that require your presence, ideally in hardware that never releases them.

The sjbis daemon and its database can store, serve, lose or hide grants. They can't forge one or alter one, because every grant is signed and every verifier checks against its own pinned registry, never against keys the daemon serves. This is why the lack of authentication on the daemon matters less for approvals than for ordinary questions: anyone on your network can answer a yes/no question, but nobody without a signer can answer an approval.

Agents can misreport what a grant authorizes: say it covers more, or something else, or that it was renewed. `ta show` followed by `ta verify --question` catches that, because the question is in the signed record and the verifier compares against it.

The asking agent chooses the question's wording. It can phrase a dangerous request blandly, or put the important part in the detail, which is not signed. The surfaces show you the requester and the verbatim question on a screen the agent can't write to (the Mac excepted), but a misleading question is still yours to catch by reading.

What the Signature Proves, and What It Doesn't

Property	Proved?	How
A pinned signer approved these exact bytes	Yes	ES256 signature over JCS bytes
The question that was approved	Yes	Verbatim question inside the record
Your note and choice	Yes	Inside the record
When it expires	Yes	Inside the record; offline verifiers enforce it
A renewal links to its parent	Yes	Signed `renews`
Which agent asked	**No**	`asked_by` is a claim
What you saw on the screen	**No**	Depends on the surface
That it hasn't been revoked	Online only	The daemon's status, failing closed
That a `--once` grant was used once	**No**, advisory	An unsigned claim on the daemon

Accepted Gaps

Each of these is known, written down in the design, and accepted for now.

Agent identity. `asked_by` is whatever name the agent gave sjbis. The design explicitly leaves "which agent asked" out of scope for v1; sjbis's planned agent tokens would close it. Until then, never let `asked_by` decide anything, and note that `ta` labels it "claimed, not proven" every time it prints it.

Hidden revocations. A daemon that wants to keep a revoked grant alive can hide the revocation from online verifiers, and offline verifiers never see revocations at all. Short lifetimes are the mitigation: a hidden revocation can extend a grant's life only up to its own `expires_at`.

Advisory claims. A `--once` grant is protected by an unsigned mark on the daemon. It stops two cooperating agents from both acting; it does not stop an agent that skips `ta claim`.

The local-signature problem. On the Mac, any process running as you can request a Touch ID signature with a misleading reason. Touch ID proves you were present, not what you read. The defense is a habit: read the record `ta approve` prints, and treat an unexpected prompt as an alarm. The Pi console exists largely to remove this problem.

Software keys. A software key on the Pi console is a file. Anyone with root on the Pi or the SD card in hand can sign as the console. The signer ladder from Chapter 8 is the path away from that.

Registry distribution. Signed enrollment isn't built yet, so the pinned registries are distributed by hand. Whoever can write a verifier's `signers.toml` decides whose signatures it believes. Guard those files like `authorized_keys`.

Lost devices. If a signer device is lost, revoke its registry entry with another device and remove it from every registry. Grants it signed before that point stay valid until they expire; another reason to keep lifetimes short.

Things That Are Not Gaps

Some worries come up often and are already handled by construction.

Replaying a grant for a different question. Impossible: the question is inside the signed bytes, and verify compares it.

Stretching a grant's lifetime. Impossible: `expires_at` is signed. Renewal makes a *new* grant, which you sign.

Auto-approval by timeout. Impossible: an approval that times out has no grant, and no rule, model or dashboard click can answer an approval without a signature.

The daemon editing a stored grant. Detected: any change to the record changes its hash, its ID and the validity of its signature.

One implementation drifting from the other. Guarded: `ta` and the daemon implement the wire format independently and both test against the same fixtures, including deliberately awkward strings and negative cases.

Why a small tool held up

The two tools have been, in their author's words, "basically rock solid", and the reasons are mostly about what they leave out. `ta` has no daemon, no database and no network protocol of its own; everything stateful is sjbis's problem, and everything `ta` does to a grant is a pure function of bytes and a pinned key file. The wire format was written down as normative before either side implemented it, and the fixtures, generated deterministically from a committed throwaway key, made disagreement between the Rust daemon, the Rust CLI and the phone's JavaScript a test failure rather than a production mystery. Exit codes are specific, so an agent that hits a problem learns *which* problem. And every place a human might be tempted to relax a rule (a timeout, a paraphrase, an unsigned answer) fails closed by default.

Open Decisions

A few questions in the design are still open, and you should know them before you build on the system.

1. **Unattested answers.** Should the web dashboard ever be allowed to record an unsigned answer to an approval, reported by `ta ask` as "unattested" with no ID? The proposal is no, and that is how the system behaves today: the daemon refuses with HTTP 409. `ta ask` still handles the case, as exit 2 with no authority, so that changing the decision later can't silently create authority.
2. **Agent identity.** Tokens issued by sjbis would let a grant prove which agent asked, and would let the daemon refuse asks from unknown agents.
3. **Hardware for the console.** Moving the Pi from a software key to a secure element or a YubiKey.

Beyond those, the roadmap lists a menubar approver for the Mac, a hash-chained ledger export of every grant, and signed enrollment.

Summary

The system aims to beat hearsay, not nation-states. Only signer devices can create grants; the daemon can hide but not forge them; agents can misreport but not survive `ta verify --question`. The signature proves the question, decision, note, expiry and renewal links. It doesn't prove which agent asked or what you saw, and revocation and single use are only as strong as the daemon you check with. Keep lifetimes short, read the record before you sign, guard the registry files, and climb the signer ladder when you can.

That is the whole system. The appendices collect the reference material.

Command and Exit-Code Reference

The help text in this appendix is generated from the `ta` source when the book is built. Global options (`--config`, `--daemon`, `--signers`) work on every command and are listed once, under `ta --help`.

Exit Codes

`ta verify`:

Exit	Verdict
0	Valid
1	Denied (or approved, with `--expect deny`)
2	Unknown ID, or an error
3	Expired
4	Revoked
5	Bad signature, malformed record, or unknown signer
6	Question mismatch
7	Consumed (`--once` grant already claimed)

Precedence when several apply: authenticity (5), question (6), revoked (4), expired (3), consumed (7), denied (1).

`ta ask` and `ta reauth`:

Exit	Result
0	Approved, and verified locally
1	Denied, and verified locally
2	Error, dismissed, cancelled, or answered without a grant
3	Timed out: no authority
5, 6	The daemon returned a grant that fails local verification

`ta claim`: 0 claimed, 2 unknown ID or not a single-use grant, 7 already consumed.

ta

```
$ ta --help
Transient authority: signed, verifiable approvals from Stephen

Usage: ta [OPTIONS] <COMMAND>

Commands:
  ask      Ask Stephen for a signed approval (blocks until answered or deadline)
  verify   Verify a grant id (online) or an exported envelope (--record, offline)
 ↪  [alias: confirm]
  reauth   Re-ask the verbatim question of an earlier grant, linked via `renews`
  show     Print a grant record; --lineage walks the renews chain [alias: get]
  log      Recent grants
  claim    Consume a --once grant (advisory, unsigned)
  export   Print the self-contained envelope JSON for offline verify
  approve  Signer device: answer and sign pending approval asks
  console  Signer device: loopback HTTP sidecar that signs for the Pi console page
  revoke   Signer device: sign a revocation of a grant
  signers  Manage the pinned signer registry
  prime    Agent primer: how to ask, verify, and what the exit codes mean
  help     Print this message or the help of the given subcommand(s)

Options:
      --config <PATH>   Config file (default ~/.config/ta/config.toml) [env:
 ↪  TA_CONFIG=]
      --daemon <URL>    sjbis daemon URL (overrides config daemon_url) [env:
 ↪  TA_DAEMON=]
      --signers <PATH>  Pinned signer registry (overrides config signers_file) [env:
 ↪  TA_SIGNERS=]
  -h, --help            Print help
  -V, --version         Print version
```

ta ask

```
$ ta ask --help
Ask Stephen for a signed approval (blocks until answered or deadline)

Usage: ta ask [OPTIONS] --agent <AGENT> <QUESTION>

Arguments:
  <QUESTION>  The question, verbatim. It is signed exactly as given

Options:
```

```
      --agent <AGENT>        Your agent name (as sjbis knows it)
      --config <PATH>        Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --daemon <URL>         sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --ttl <TTL>           How long the grant may be relied on once signed [default:
↪  1h]
      --once                 Request a single-use grant (consume with `ta claim`)
      --signers <PATH>       Pinned signer registry (overrides config signers_file)
↪  [env: TA_SIGNERS=]
      --deadline <DEADLINE>  How long to wait for an answer (daemon caps blocking
↪  waits at 10m) [default: 10m]
      --choices <CHOICES>    Comma-separated choices; the chosen value is signed as
↪  `answer`
      --detail <DETAIL>      Extra context shown to Stephen (not signed)
      --urgency <URGENCY>    sjbis urgency 0..5 (push notifications need >= 2)
↪  [default: 2]
      --json                 Print the whole signed envelope as JSON
  -h, --help                 Print help
```

ta verify

```
$ ta verify --help
Verify a grant id (online) or an exported envelope (--record, offline)

Usage: ta verify [OPTIONS] [ID]

Arguments:
  [ID]  Grant id (ta-xxxxxx), fetched from the daemon

Options:
      --config <PATH>        Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --record <FILE>        Offline: verify an exported envelope file (signature +
↪  expiry only)
      --daemon <URL>         sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --question <QUESTION>  The verbatim question you believe was approved. Always
↪  pass this
      --expect <EXPECT>      Required decision (default approve) [default: approve]
↪  [possible values: approve, deny]
      --signers <PATH>       Pinned signer registry (overrides config signers_file)
↪  [env: TA_SIGNERS=]
      --json
  -h, --help                 Print help
```

ta show

```
$ ta show --help
Print a grant record; --lineage walks the renews chain

Usage: ta show [OPTIONS] <ID>

Arguments:
  <ID>

Options:
      --config <PATH>   Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --lineage         Walk the renews chain up to the root and list every renewal
↪  below it
      --daemon <URL>    sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --json
      --signers <PATH>  Pinned signer registry (overrides config signers_file) [env:
↪  TA_SIGNERS=]
  -h, --help            Print help
```

ta reauth

```
$ ta reauth --help
Re-ask the verbatim question of an earlier grant, linked via `renews`

Usage: ta reauth [OPTIONS] <ID>

Arguments:
  <ID>  Earlier grant to re-authorise

Options:
      --config <PATH>        Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --ttl <TTL>            TTL for the new grant [default: 1h]
      --agent <AGENT>        Agent name (default: asked_by of the earlier grant)
      --daemon <URL>         sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --deadline <DEADLINE>  [default: 10m]
      --signers <PATH>       Pinned signer registry (overrides config signers_file)
↪  [env: TA_SIGNERS=]
      --urgency <URGENCY>    [default: 2]
      --json
  -h, --help                 Print help
```

ta claim

```
$ ta claim --help
Consume a --once grant (advisory, unsigned)

Usage: ta claim [OPTIONS] --by <BY> <ID>

Arguments:
  <ID>

Options:
      --by <BY>          Agent consuming the grant
      --config <PATH>    Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --daemon <URL>     sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --signers <PATH>   Pinned signer registry (overrides config signers_file) [env:
↪  TA_SIGNERS=]
  -h, --help             Print help
```

ta approve

```
$ ta approve --help
Signer device: answer and sign pending approval asks

Usage: ta approve [OPTIONS] [ASK_ID]

Arguments:
  [ASK_ID]  Answer this sjbis ask id directly instead of picking from the list

Options:
      --config <PATH>         Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --decision <DECISION>   approve or deny (prompted if omitted) [possible values:
↪  approve, deny]
      --answer <ANSWER>       Chosen value when the ask has choices
      --daemon <URL>          sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --note <NOTE>           Note to sign with the decision
      --signers <PATH>        Pinned signer registry (overrides config signers_file)
↪  [env: TA_SIGNERS=]
      --ttl <TTL>             Grant lifetime (default: the TTL the agent requested,
↪  else 1h)
```

```
  -y, --yes                  Do not prompt for anything not given on the command line
  -h, --help                 Print help
```

ta revoke

```
$ ta revoke --help
Signer device: sign a revocation of a grant

Usage: ta revoke [OPTIONS] <ID>

Arguments:
  <ID>  Grant to revoke

Options:
      --config <PATH>    Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --reason <REASON>  [default: ""]
      --daemon <URL>     sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
  -y, --yes              Skip the confirmation prompt
      --signers <PATH>   Pinned signer registry (overrides config signers_file) [env:
↪  TA_SIGNERS=]
  -h, --help             Print help
```

ta console

```
$ ta console --help
Signer device: loopback HTTP sidecar that signs for the Pi console page

Usage: ta console [OPTIONS]

Options:
      --config <PATH>          Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --listen <ADDR:PORT>     Address to listen on. Anything but loopback must be
↪  asked for explicitly [default: 127.0.0.1:7879]
      --allow-origin <ORIGIN>  Origin allowed to call the sidecar from a browser
↪  (repeatable; default: the origin of daemon_url)
      --daemon <URL>           sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --signers <PATH>         Pinned signer registry (overrides config signers_file)
↪  [env: TA_SIGNERS=]
  -h, --help                   Print help
```

ta signers

```
$ ta signers --help
Manage the pinned signer registry

Usage: ta signers [OPTIONS] <COMMAND>

Commands:
  list        List pinned signers
  add         Pin a signer public key in the local registry
  rm          Remove a signer from the local registry
  keygen-dev  Generate a development software key (PKCS#8 PEM) and print its pubkey
  help        Print this message or the help of the given subcommand(s)

Options:
      --config <PATH>   Config file (default ~/.config/ta/config.toml) [env:
↪  TA_CONFIG=]
      --daemon <URL>    sjbis daemon URL (overrides config daemon_url) [env:
↪  TA_DAEMON=]
      --signers <PATH>  Pinned signer registry (overrides config signers_file) [env:
↪  TA_SIGNERS=]
  -h, --help            Print help
```

sjbis daemon endpoints for grants

Endpoint	Does
`POST /grants`	Verify an envelope, store it, and apply it (answer the ask, mark a revocation)
`GET /grants/ID`	`{envelope, status}`
`GET /grants?ask=ASK_ID`	Grants answering one ask
`GET /grants?since=TS[&agent=NAME][&limit=N]`	Recent grants, newest first (what `ta log` reads)
`GET /grants?renews=ID`	Renewals of a grant (what `ta show --lineage` reads)
`GET /grants/ID/status`	`{expired, revoked_by, claimed_by, claimed_at}`
`POST /grants/ID/claim`	Consume a `once` grant; body `{"by": "AGENT"}`
`GET /signers`	The daemon's registry copy; never trusted by verifiers

Wire-Format Reference

This appendix restates the normative wire format that `ta`, the sjbis daemon and the phone web app each implement independently. The repository's `fixtures/` directory holds test vectors for every rule here.

Encodings

Item	Format
Canonical bytes	RFC 8785 JCS of `record`: sorted keys, no whitespace, raw UTF-8, mandatory escapes only, `/` unescaped
`record_sha256`	Lowercase hex of SHA-256(canonical bytes)
`id`	`"ta-"` + first 6 characters of RFC 4648 base32 (alphabet `a-z2-7`, lowercase, no padding) of the SHA-256 digest
`alg`	`"es256"`: ECDSA P-256 over SHA-256(canonical bytes)
`sig`	Standard base64 with padding of the DER signature; low-S and high-S both accepted
Public key	Standard base64 of the 65-byte SEC1 uncompressed point, `04 ‖ x ‖ y`
Timestamps	Exactly `YYYY-MM-DDTHH:MM:SSZ`

Envelope

```
{
  "id": "ta-xxxxxx",
  "record": { },
  "record_sha256": "<64 hex characters>",
  "alg": "es256",
  "sig": "<base64 DER>"
}
```

Unknown envelope fields are rejected.

Record Kinds

All records have `"v": 1`, a `kind`, an `issued_at` and a `signer`. Unknown fields, `null` values, any other `v`, and non-canonical timestamps make a record malformed. Optional fields are omitted when absent.

grant

Field	Required	Notes
`ask_id`	Yes	The sjbis ask answered
`asked_by`	Yes	Claimed agent name
`question`	Yes	Verbatim
`decision`	Yes	`approve` or `deny`
`answer`	No	The chosen value, when the ask had choices
`note`	No	Free text from the approver
`expires_at`	Yes	Must be after `issued_at`
`once`	No	Single use
`renews`	No	ID of the grant this one re-authorizes

revoke

Field	Required	Notes
`target`	Yes	ID of the revoked grant
`reason`	Yes	Free text, may be empty

enroll (stored by the daemon; client support is planned)

Field	Required	Notes
`signer_id`	Yes	The new signer's ID
`method`	Yes	`faceid`, `touchid`, `yubikey` or `software`
`device`	Yes	Description
`pubkey`	Yes	SEC1 point, base64

Revoke and enroll records never expire.

Daemon Errors on `POST /grants`

Bodies are `{"error": "<message>", "code": "<code>"}`. Nothing is written when a post is refused.

HTTP	Code	When
400	malformed	Envelope or record shape, unknown field, null, bad timestamp, expires_at <= issued_at
404	notification_not_found, target_not_found	Unknown ask or revoke target
409	not_open, not_approval	The ask is closed, or not an approval
409	id_collision	The ID exists; re-sign with a fresh issued_at
422	unsupported_alg, hash_mismatch, id_mismatch	Derived fields don't match the record
422	unknown_signer, bad_signature	Not pinned on the daemon, or the signature fails
422	question_mismatch	The record's question differs from the ask's by any byte
500	registry_error	The daemon's signers.toml is unreadable or invalid

The daemon's question check is byte-exact; ta verify --question ignores whitespace differences only. The two serve different purposes: the daemon guarantees that the signed question *is* the asked question, and the verifier guarantees that it is the question the agent *thinks* it is.