Manual 05 · keys, trust, secrets, TLS

Security & Certificates

The security model of Stilton in practice — what each protection guarantees and what it does not, how to manage signing keys over their whole life, how to provide secrets, how to set up TLS and certificates for every kind of connection, and how to expose the monitor. Ends with a hardening checklist.

CONTENTS

01 **The model in one page**

02 **Signing keys over their whole life**
- 2.1 Rules for developers
- 2.2 Rules for administrators of run machines
- 2.3 Algorithms

03 **Secrets**

04 **TLS and certificates**
- 4.1 Trusting a company certificate authority
- 4.2 Connection by connection
- 4.3 Example — PostgreSQL with full verification
- 4.4 SSH host keys
- 4.5 Webhooks

05 **Exposing the monitor**
- 5.1 Behind a reverse proxy with TLS

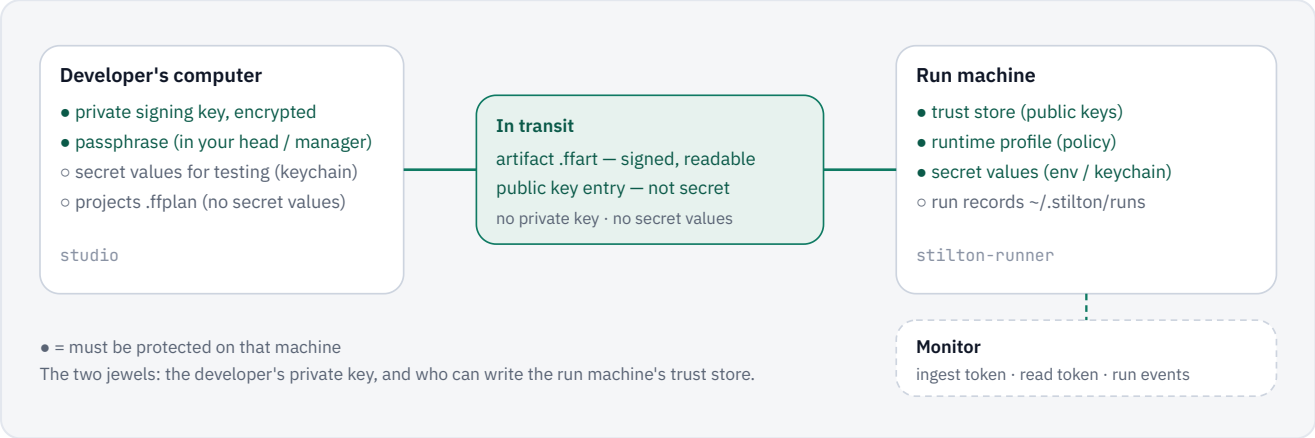
06 **The studio as an attack surface**

07 **Hardening checklist**

Copyright © 2025–2026 Marco Dongu. All rights reserved.

01 The model in one page

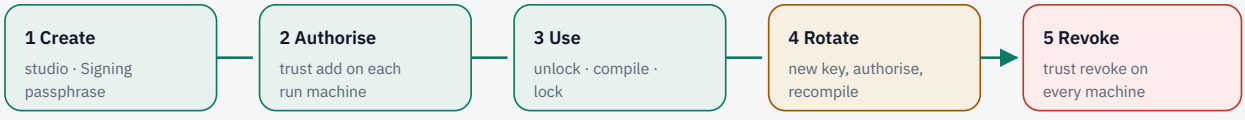
Stilton's security answers one question end to end: *can the machine that runs a job prove exactly what it is about to execute, and who authorised it?* Each piece holds only what it needs.



Protection	Guarantees	Does not guarantee
Artifact signature (Ed25519)	the artifact was authorised by a trusted key, and nothing in it changed since	that the flow is harmless — a trusted developer can sign a harmful flow
Runtime profile	the machine refuses node types it should never run (shell, SSH…)	anything about node types it allows
Encrypted signing key	a stolen key file is useless without the passphrase	protection on a computer already infected while the key is unlocked
Secrets by reference	secret values never enter projects, artifacts or logs	protection of the run machine's environment itself
TLS on connections	data and passwords are encrypted, and the server is the right one when the certificate is verified	anything, when verification is switched off
Monitor tokens	only runners with the ingest token write; only people with the read token read	confidentiality on the network: the monitor speaks plain HTTP, put TLS in front

The artifact is **signed, not encrypted**: whoever holds it can read the flow — SQL, host names, structure. Treat artifacts as internal documents.

02 Signing keys over their whole life



A lost laptop or a passphrase that may have been seen: go straight to 5 for the old key, then 1–2 for a new one.

Rules for developers

- **One key per computer.** If a laptop is lost, only its key needs revoking.
- **A strong passphrase** — at least four random words, or 16+ characters from a password manager. The key file is protected by Argon2id, which slows down guessing, but a weak passphrase can still be guessed offline from a stolen file.
- **Lock the key** (*Signing → Lock now*) when you leave the computer; it is wiped from memory.
- **Never share** `~/.stilton/signing-key.json` or the passphrase. Share only the trust store entry.

Rules for administrators of run machines

- **Who can write the trust store decides what runs.** Keep it `600`, owned by the service account; keep the service account's home `700`; do not let developers log in as the service account.
- **Verify every new keyId on a second channel** before `trust add`.
- **Record an owner** that identifies a person and a device: `--owner "Mark Rossi — laptop 2026"`.
- **Revoke on every machine** when someone leaves or a device is lost. Keep a list of which machines trust which keys; `trust list` on each machine is the source of truth.

Algorithms

Purpose	Algorithm	Notes
Artifact signature	**Ed25519**	32-byte keys; the keyId is `ed25519:` + fingerprint of the public key
Plan integrity	**SHA-256** of the plan in canonical form	recomputed by the runner at every admission
Key at rest	**Argon2id → XChaCha20-Poly1305**	random salt and nonce per file
Randomness	operating system's random source	for keys, salts and nonces
Implementation	audited Rust crates (`ed25519-dalek`, `sha2`, `argon2`, `chacha20poly1305`)	no home-made cryptography

03 Secrets

A secret is declared by **name** in the studio and referenced as `${NAME}` — in the fields of a resource, and in the credential fields of a node (password, bearer token, API key value, client secret, HMAC secret, SMTP password, key passphrase…). The value is resolved by the native side when the flow runs or when a panel tests a connection; it never reaches the interface, the project file or the artifact.

Warning. A credential typed in clear in one of those fields is saved in the `.ffplan` and in every artifact compiled from it. Validation shows a warning (`CREDENTIAL_IN_CLEAR`) on each node and resource that has one: replace the value with `${NAME}` before publishing.

Where	How the value is provided
Developer's computer (tests)	*Environments* → *Secrets*: stored in the system keychain, service `stilton`
Linux server	an environment file readable only by the service account, loaded by systemd (`EnvironmentFile=`)
Windows server	Credential Manager of the service account, or environment variables of the task
Container	the orchestrator's secret mechanism, exposed as environment variables

Good practice:

- **Least privilege at the source.** A flow that only reads gets a database user that can only read: `GRANT SELECT ON orders TO etl_reader`.
- **One secret per purpose**, not one shared administrator password: revoking one job does not break the others.
- **`onlyDeclared` in production.** With `"secrets": { "onlyDeclared": true }` in the runtime profile, an artifact can resolve only the secrets it declares — it cannot read other environment variables of the machine by writing `${AWS_SECRET_ACCESS_KEY}` in a field.
- **Rotate** by changing the value on the run machine; artifacts do not change.
- **Never in Diagnostic logs.** Secret values are not logged, but the *Diagnostic* level sends row content to the monitor: keep it for test data.

---

04 TLS and certificates

Every network connection of the engine can be encrypted. The engine uses the operating system's TLS stack — OpenSSL on Linux, SChannel on Windows — and therefore **the operating system's trusted certificate authorities**.

Trusting a company certificate authority

If your servers use certificates issued by an internal CA, add the CA to the system store of the run machine (and of the developers' computers):

Debian / Ubuntu — bash

```bash
sudo cp company-root-ca.crt /usr/local/share/ca-certificates/
sudo update-ca-certificates
```

Fedora / RHEL — bash

```bash
sudo cp company-root-ca.crt /etc/pki/ca-trust/source/anchors/
sudo update-ca-trust
```

Windows (as administrator) — powershell

```powershell
Import-Certificate -FilePath .\company-root-ca.cer -CertStoreLocation Cert:\LocalMachine\Root
```

Then verify from the machine, outside Stilton:

Linux bash

```
openssl s_client -connect db.internal:5432 -starttls postgres -brief </dev/null
openssl s_client -connect ldap.internal:636 -brief </dev/null       # look for "Verification: OK"
```

Warning. Do not fix a certificate error by switching verification off. An encrypted connection to an unverified server protects nothing: the password goes, encrypted, to whoever answered.

Connection by connection

Connection	Setting	Recommended	Notes
PostgreSQL	resource *SSL*	`verify-full`	`require` encrypts without checking the server; `verify-ca` checks the CA; `verify-full` also checks the host name
MySQL / MariaDB	resource *SSL*	`verify-full`	same values, mapped to MySQL's `ssl-mode`
Oracle, Informix, ODBC	driver configuration	TLS in the driver	set in the vendor's client (Oracle wallet, Informix `sqlhosts` options) or in the connection string's additional parameters
SQLite	—	—	a local file: protect it with file permissions
HTTP sources and outputs	`https://` URL	always	the certificate is always verified; there is no switch to disable it
LDAP	resource *TLS mode*, *Verify certificate*	`ldaps` + verify `true`	`starttls` is also encrypted; `plain` sends the bind password in clear
FTP	protocol	`sftp` or `ftps`	`ftps` = explicit TLS before login, data channel protected, certificate verified; `ftp` is plain text
SFTP, SSH Exec	host key	verified against `known_hosts`	see below
Mail (SMTP)	*Security*	`STARTTLS` (587) or `SSL/TLS` (465)	the server certificate is verified; `None` sends in clear
MQTT	resource *TLS*	on (port 8883)	
ActiveMQ (STOMP)	resource *TLS*	on	
Monitor	URL	`https://` through a reverse proxy	the monitor itself speaks HTTP

Example — PostgreSQL with full verification

Field	Value
Host	`db.internal` — must match the name in the server certificate
Port	`5432`
SSL	`verify-full`
User / Password	`etl_reader` / `${DB_PASSWORD}`

If **Test connection** fails with a certificate error, the server's certificate is not issued by a CA the machine trusts, or its name does not match `db.internal`. Fix the CA store or use the host name written in the certificate — not an IP address.

SSH host keys

SFTP and SSH Exec check that the server's host key is the one recorded in `~/.ssh/known_hosts` of the account that runs the flow. A server that is not listed, or whose key changed, is refused before any password is sent.

Record a server once, from the run machine and as the service account, after checking the fingerprint with the server's administrator:

Linux — bash

```
sudo -u stilton ssh-keyscan -t ed25519 sftp.partner.com >> /var/lib/stilton/.ssh/known_hosts
ssh-keygen -lf /var/lib/stilton/.ssh/known_hosts      # compare the fingerprint
```

Windows — powershell

```
ssh-keyscan -t ed25519 sftp.partner.com | Out-File -Append -Encoding ascii
$env:USERPROFILE\.ssh\known_hosts
```

Prefer key authentication to passwords for SSH and SFTP: a private key file readable only by the service account (`chmod 600`), with its passphrase as a secret.

Webhooks

The **Webhook Receiver** listens on plain HTTP on the port you choose. In production:

- put it behind a reverse proxy that terminates TLS (see the monitor example below — the same configuration applies);
- set an **HMAC secret** shared with the sender (GitHub, Shopify, your systems). A request with a missing or wrong signature is refused with **401** and never enters the flow; the refusal is logged with the sender's address;
- restrict the senders with the **IP allowlist** when their addresses are known.

05 Exposing the monitor

The monitor refuses to start without two different tokens, and listens only on `127.0.0.1` by default.

Variable	Default	Purpose
`MONITOR_TOKEN`	required	**ingest** token: runners send it with every batch. At least 16 characters
`MONITOR_READ_TOKEN`	required, different	**read** token: the password for the web page and the API
`MONITOR_BIND`	`127.0.0.1`	listening address
`MONITOR_MAX_BODY_BYTES`	8 MiB	maximum size of one push
`MONITOR_RATE_BYTES_PER_MIN`	64 MiB	volume accepted per IP address per minute
`MONITOR_AUTH_FAIL_MAX`	20	wrong read passwords per IP per minute before refusing
`MONITOR_INSECURE_LOCAL`	off	`1` makes tokens optional, on loopback only — for local experiments

Generate tokens with a real random source:

Linux — bash

```
openssl rand -hex 32
```

Windows — powershell

```
-join ((1..32) | ForEach-Object { '{0:x2}' -f (Get-Random -Maximum 256) })
```

Behind a reverse proxy with TLS

Keep the monitor on `127.0.0.1:8787` and publish it through a proxy that owns the certificate.

/etc/nginx/sites-available/stilton-monitor nginx

```
server {
    listen 443 ssl;
    server_name monitor.internal;
    ssl_certificate     /etc/ssl/monitor.internal.crt;
    ssl_certificate_key /etc/ssl/monitor.internal.key;

    client_max_body_size 8m;              # same as MONITOR_MAX_BODY_BYTES

    location / {
        proxy_pass http://127.0.0.1:8787;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

Runners then use `MONITOR_URL=https://monitor.internal/ingest`. The monitor sees the proxy as the sender address; the run's provenance still shows the host each runner declares.

Note. What the read token protects is the content of run events: node names, messages, error texts and — at *Diagnostic* level only — row content. Give the read token to the people who operate the jobs, not to everyone.

---

06 The studio as an attack surface

The studio holds the developer's key, so it is hardened like a sensitive application:

- the user interface can only load code shipped with it (strict content security policy; no scripts or fonts from the internet);
- it has the minimum desktop permissions: no general file system or shell access from the interface; files are read and written only through the studio's own commands;
- the private key is handled only by the native side and never reaches the interface; when locked, it is wiped from memory;
- connection tests are made by the native side too — for example *Test* in the HTTP source panel: the `${SECRET}` references are resolved there and the response comes back with secret values masked (`***`);
- the interface runs **only inside the desktop app**: the production build opened in a web browser shows a page saying so, instead of an interface with no back-end.

Browser mode is a development tool. During development (`npm run dev`) the same interface can be opened in a browser at `http://localhost:5173`: it cannot open files, reach the keychain or run flows. The development server serves the whole source code, so it listens **only on this computer**; starting it with `--host` fails on purpose (it can be overridden with `STILTON_DEV_EXPOSE=1`, which should never be needed). Do not run it on shared machines.

What remains your responsibility: keep the operating system updated, do not run untrusted software on the computer that holds a signing key, and lock the key when you are away.

07 Hardening checklist

Developer computers

- [ ] one signing key per computer, strong passphrase, kept in a password manager
- [ ] key locked when not compiling
- [ ] secrets for tests in the keychain, not in project variables
- [ ] no `CREDENTIAL_IN_CLEAR` warning in the Validation tab before publishing
- [ ] development server never exposed on the network
- [ ] projects in version control; artifacts compiled from saved, named versions

Run machines

- [ ] a dedicated service account, no interactive login, home folder `700`
- [ ] `trust-store.json` owned by the service account, `600`; every keyId verified on a second channel; owners recorded
- [ ] a runtime profile denying the node types the machine does not need (`shell_exec`, `ssh_exec` at least) and `onlyDeclared: true`
- [ ] secrets in an environment file `600` or the account's credential store — never on command lines
- [ ] company CA in the system store; databases with `verify-full`; LDAP with `ldaps` and verification
- [ ] SSH/SFTP servers recorded in the service account's `known_hosts`
- [ ] artifacts checked with `--admit-only` before being scheduled

Monitor

- [ ] two different random tokens of 32+ bytes
- [ ] bound to `127.0.0.1`, published only through a TLS reverse proxy
- [ ] read token given only to operators
- [ ] *Diagnostic* log level never used on personal or confidential data

When something happens

Event	Do this
Laptop lost or stolen	revoke its keyId on every run machine; create a new key on the replacement
Passphrase may have been seen	rotate the key (new key, authorise, recompile, revoke the old one)
A secret leaked	change it at the source (database, API), then update the run machines
A run machine compromised	revoke nothing on the developers' side — but rotate every secret that machine held, and rebuild it
A developer leaves	revoke their keyIds; recompile with another key the artifacts still in use