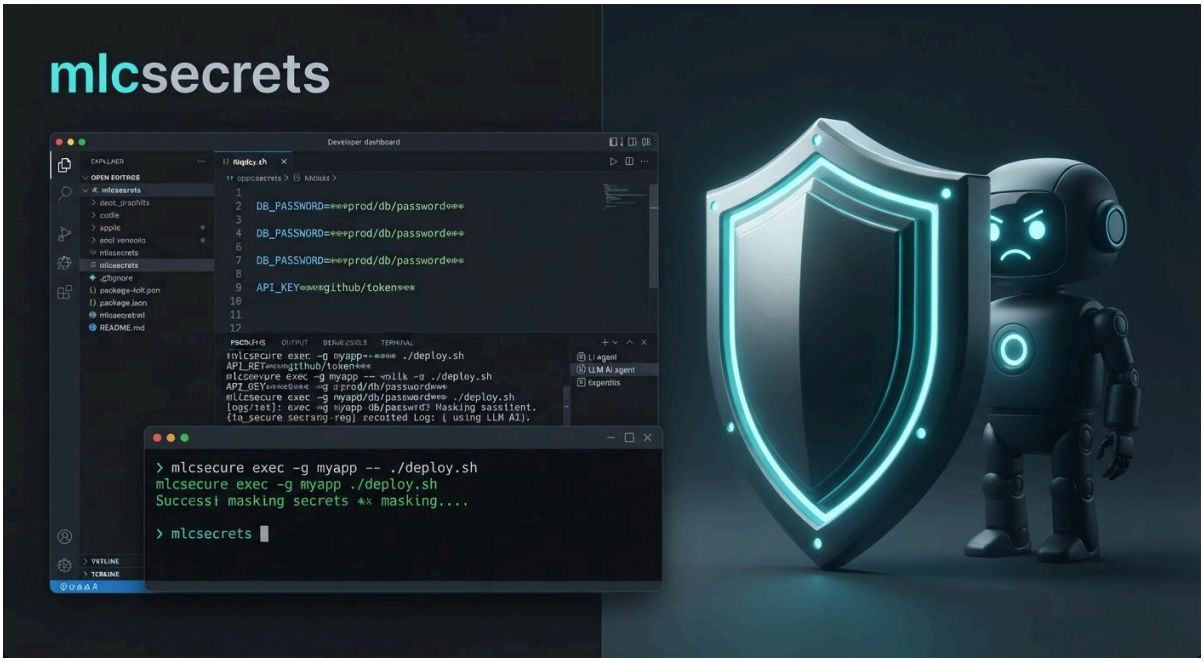
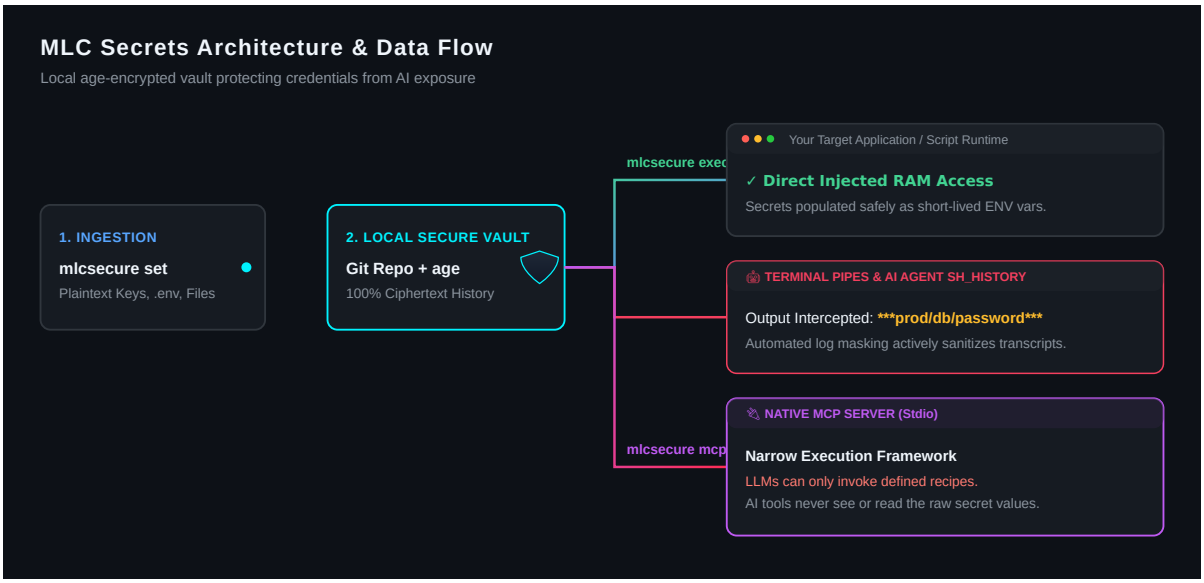
MLC Secrets



Der lokale, KI-sichere Secrets-Tresor für Entwickler.

Dieses Handbuch beschreibt den Befehl `mlcsecure`: die tägliche Arbeit, das Einrichten weiterer Rechner und KI-Agenten und das Sicherheitsmodell — was geschützt ist, was nicht, und was du beitragen musst.

Autor: **Michael Lechner** · https://mlcgo.eu/products/mlcsecrets/



mlcsecure — Bedienung

`mlcsecure` bewahrt deine Secrets — Passwörter, API-Keys, Tokens und ganze Dateien — verschlüsselt in einem Git-Repository auf deinem Rechner auf. Jede Änderung ist ein Commit, also geht nie wirklich etwas verloren, und das Repository darf auf einen Remote gepusht werden, weil es nur Chiffretext enthält.

`mlcsecure` ist der Befehl von **MLC Secrets**. Seine Datenverzeichnisse heißen nach dem Produkt: `~/.local/share/mlcsecrets` (Store) und `~/.config/mlcsecrets` (Identität, Konfiguration).

> Diese Anleitung beschreibt **Version 1.9**.

Inhalt

- Installation
- KI-Agenten anlernen: setup
- Erster Start
- Secrets benennen
- Ein Secret ablegen
- Eine Datei ablegen
- Eine .env-Datei importieren
- Beschreibungen
- Auflisten
- Umbenennen und bearbeiten
- Löschen
- Verlauf und Rollback
- Ein Secret lesen
- Einen Befehl mit Secrets ausführen
- Konfigurationsdateien und bestehende Skripte
- Einmal entsperren: der Agent
- Exec-only-Gruppen
- On-demand-Secrets, befristete Freigaben und Sperren
- Konfigurationsdateien von Servern
- Rezepte und der MCP-Server
- Sicherung und Synchronisierung mit einem Remote-Repository
- Ein zweiter Rechner
- Wiederherstellungsschlüssel
- Exit-Codes
- Sicherheitshinweise

Installation

```
task install          # baut mlcsecure und kopiert es nach ~/.local/bin
mlcsecure --version
```

Installation auf einem anderen Rechner, von einem aus, der den Quellcode hat, über ssh:

```
task deploy HOST=macstudio   # erkennt das System, baut, kopiert, zeigt den Setup-Plan
task deploy HOST=win10       # Windows: %LOCALAPPDATA%\Programs\mlcsecrets, wird in den PATH aufgenommen
task deploy HOST=ich@win11   # anderer Benutzername auf dem Rechner
```

Voraussetzungen: `git` im PATH; Linux, macOS oder Windows 10 (ab 1803) / 11.

Diese Anleitung und das Sicherheitsmodell sind als PDF-Handbuch in mlcsecure eingebaut: `mlcsecure manual --lang de` öffnet es, `mlcsecure manual --lang de --out DATEI` speichert es.

Für einen Mac mit Apple Silicon baust du unter Linux mit `task build:darwin` (→ `bin/mlcsecure-darwin-arm64`) oder führst `task install` direkt auf dem Mac aus.

Für Windows baust du mit `task build:windows` (→ `bin/mlcsecure-windows-amd64.exe`), kopierst die Datei als `mlcsecure.exe` in einen Ordner im PATH und nutzt sie aus PowerShell oder cmd. Unterschiede zu Linux und macOS:

- Pfade: Store in `%USERPROFILE%\.local\share\mlcsecrets\store`, Identität in `%USERPROFILE%\.config\mlcsecrets\identity.age`, Agent-Socket in `%LOCALAPPDATA%\mlcsecrets\agent.sock`.
- `edit` nutzt `%EDITOR%`, sonst Notepad (`$env:EDITOR = "code --wait"` für VS Code).
- `ssh-add` braucht den laufenden Windows-Dienst *OpenSSH Authentication Agent*.
- `backup-file`, `restore` und `check` funktionieren nur mit `--host` (ein Linux-Server); lokale Systemdateien werden unter Windows nicht unterstützt.

WSL ist für mlcsecure ein eigener Linux-Rechner, auch auf derselben Maschine. Die Windows-`mlcsecure.exe` nutzt Schlüssel und Agent von Windows, WSL erreicht den Agent-Socket von Windows nicht, und Agenten in WSL lesen ihren Skill aus dem Home-Verzeichnis von WSL. Gib WSL deshalb das Linux-Binary (`task install` in WSL, oder `bin/mlcsecure` nach `~/.local/bin` kopieren) und arbeite dort die Checkliste: ein neuer Rechner ab, mit eigener Passphrase, eigenem ssh-Schlüssel für das Remote und eigenem Empfängernamen wie `win11-wsl`. `task deploy` erreicht nur die Windows-Seite. Über ssh rufst du es über eine Login-Shell auf, sonst ist `~/.local/bin` nicht im PATH: `ssh ich@win11 'wsl -e bash -lc "mlcsecure key"'`.

KI-Agenten anlernen: setup

Coding-Agenten lernen aus einem *Skill*, wie sie mlcsecure benutzen — und was sie damit nie tun dürfen. Ein Skill ist eine kurze Anleitungsdatei, die sie von selbst laden. `mlcsecure setup` legt ihn dort ab, wo jeder Agent auf diesem Rechner nachsieht:

```
mlcsecure setup              # zeigen, was getan würde
mlcsecure setup --yes        # ausführen
mlcsecure setup --check      # Exit 5, wenn etwas fehlt oder veraltet ist
mlcsecure setup --remove --yes
```

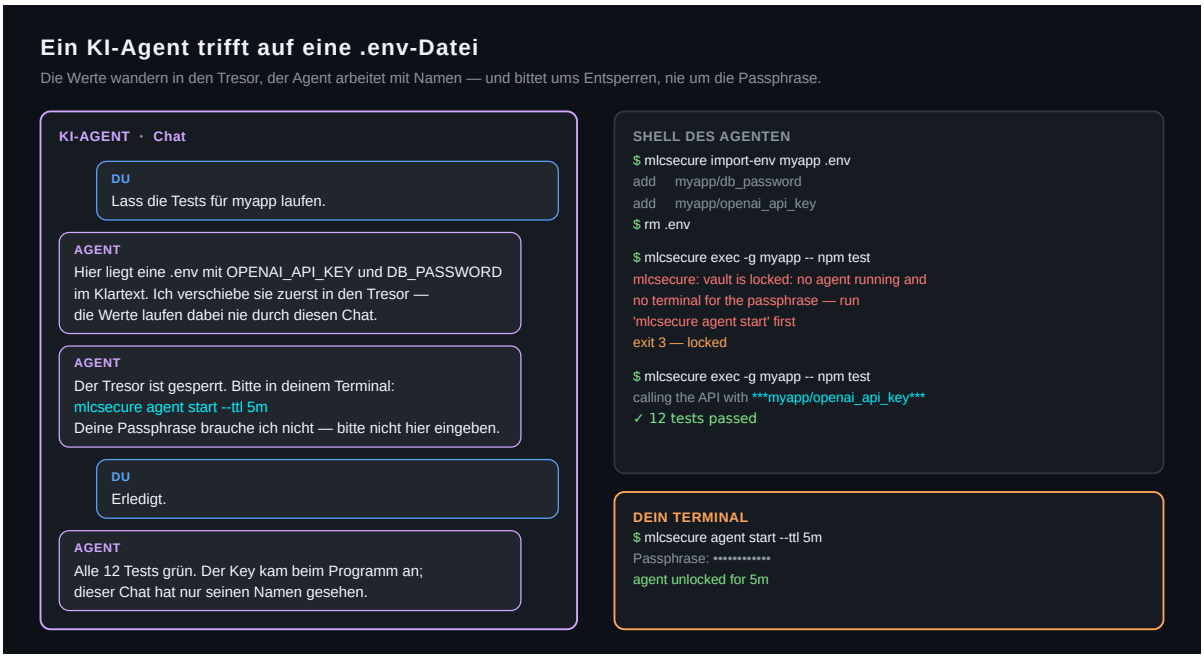
Agent	Skill	MCP-Server (`--mcp`)
Claude Code	`~/.claude/skills/mlcsecure/`	`claude mcp add --scope user`
OpenAI Codex	`~/.agents/skills/mlcsecure/`	`~/.codex/config.toml`
Gemini CLI, Antigravity	`~/.gemini/skills/mlcsecure/`	`~/.gemini/settings.json`, `~/.gemini/antigravity/mcp_config.json`
OpenCode	liest `~/.claude/skills`	der Eintrag wird angezeigt; trag ihn selbst in `opencode.json` ein

Agenten werden an ihrem Konfigurationsordner erkannt; mit `--agent claude,gemini` wählst du sie selbst aus. Der Skill ist ins Binary eingebaut, also führe nach einem Update `mlcsecure setup --yes` erneut aus. `mlcsecure setup --print-skill` gibt ihn aus.

Agenten mit Shell brauchen nur den Skill: Sie rufen `mlcsecure exec` auf. `--mcp` registriert zusätzlich den MCP-Server für Werkzeuge ohne Shell. Geändert wird nur der `mlcsecure`-Eintrag einer Konfigurationsdatei; die vorherige Datei bleibt als `FILE.bak` erhalten, und eine Konfigurationsdatei, die ein Symlink ist (z. B. in ein Dotfiles-Repository), wird nie geschrieben — setup zeigt stattdessen den Eintrag an.

Ob ein Agent dem Skill folgt, hängt vom Modell ab. In Testsitzungen haben Claude Code und OpenCode mit einem gehosteten Modell ihn von selbst geladen und nie einen Wert angefasst. Ein kleines lokales Modell (Qwen 27B über Ollama) ignorierte den Skill, selbst als der Auftrag „use mlcsecure“ sagte, und wollte direkt die `.env` lesen; OpenCode hat das von sich aus blockiert. Für Arbeit mit Secrets ein Modell nehmen, das Skills befolgt.

So sieht eine Sitzung mit dem Skill aus — der Agent findet eine `.env`-Datei, verschiebt sie in den Tresor, bittet dich ums Entsperren und lässt die Tests laufen; er sieht dabei nur Namen:



Erster Start

```
mlcsecure init
```

Du wirst zweimal nach einer **Passphrase** gefragt. Sie schützt deinen privaten Schlüssel — das Einzige, womit sich deine Secrets entschlüsseln lassen. Wähl eine lange und bewahre sie sicher auf: **Ohne sie sind deine Secrets nicht wiederherstellbar.**

`init` legt zwei Dinge an:

Was	Standardort	Hinweis
Deine Identität (privater Schlüssel, mit der Passphrase verschlüsselt)	`~/.config/mlcsecrets/identity.age`	diese Datei nie committen oder weitergeben
Der Store (Git-Repository mit den verschlüsselten Secrets)	`~/.local/share/mlcsecrets/store`	darf auf einen Remote gepusht werden

Für andere Orte setzt du `MLCSECURE_IDENTITY` und/oder `MLCSECURE_STORE`.

Gibt es schon eine Identität (zum Beispiel nach dem Einrichten eines zweiten Stores), fragt `init` nach ihrer Passphrase und verwendet sie weiter.

Secrets benennen

Secrets werden über **Pfade** angesprochen; die Teile vor dem letzten `/` sind Gruppen:

```
prod/db/password
prod/db/user
openai/api_key
```

Jeder Teil besteht aus Kleinbuchstaben, Ziffern, `.`, `_` und `-` und darf nicht mit einem Punkt beginnen. Die Pfade sind im Store lesbar (damit `git log` Sinn ergibt); die Werte und alle Metadaten sind verschlüsselt.

Ein Secret ablegen

```
mlcsecure set prod/db/password
```

Du tippst den Wert zweimal; auf dem Bildschirm erscheint nichts. Du kannst ihn auch per Pipe übergeben — ein abschließender Zeilenumbruch wird entfernt:

```
pass show db | mlcsecure set prod/db/password
```

Werte werden nie als Argumente übergeben (`mlcsecure set name VALUE` gibt es nicht), weil Argumente in deiner Shell-History und in `ps` landen.

Zum Ablegen brauchst du deine Passphrase **nicht** — nur zum Lesen. Setzt du einen vorhandenen Pfad, wird er überschrieben; der alte Wert bleibt im Verlauf.

Eine Datei ablegen

Zertifikate, kubeconfigs, SSH-Schlüssel — jede Datei bis 10 MiB:

```
mlcsecure add-file certs/server ./server.pem
```

Der ursprüngliche Dateiname (`server.pem`) wird ebenfalls gespeichert, verschlüsselt. Die Klartextdatei kannst du danach löschen.

Eine .env-Datei importieren

```
mlcsecure import-env myapp .env --dry-run   # zeigen, was passieren würde
mlcsecure import-env myapp .env
```

Aus jedem `KEY=VALUE` wird `myapp/<key in Kleinbuchstaben>` — aus `OPENAI_API_KEY` wird `myapp/openai_api_key` — alles in einem Commit. `export`, Kommentare und Werte in Anführungszeichen werden verstanden. Vorhandene Secrets mit demselben Pfad werden überschrieben (die alten Werte bleiben im Verlauf). Die `.env`-Datei selbst wird nicht angefasst: **Lösch sie selbst**, sobald der Import stimmt.

Beschreibungen

Bei vielen Projekten werden Namen allein kryptisch. Secrets und ganze Gruppen können eine kurze Beschreibung haben:

```
mlcsecure describe mlcauth "Project mlcauth (auth server)"     # eine Gruppe
mlcsecure describe mlcauth/db_password Postgres, user app       # ein Secret
echo "$TOKEN" | mlcsecure set ci/token --desc "GitHub token for CI"
mlcsecure import-env myapp .env --desc "Project myapp"          # beschreibt die Gruppe
mlcsecure describe mlcauth                                       # anzeigen
mlcsecure describe mlcauth --clear                               # entfernen
mlcsecure ls -d                                                  # Liste mit Beschreibungen
```

```
mlcauth/              Project mlcauth (auth server)
mlcauth/db_password   Postgres, user app
mlcauth/jwt_key
```

Beschreibungen werden **lesbar** gespeichert, wie die Namen (`.mlcs/descriptions` im Store): `ls -d` braucht keine Passphrase, und KI-Agenten können damit das richtige Secret auswählen. Jeder, der den Store hat — auch der Remote —, kann sie lesen, also **schreib nie ein Secret in eine Beschreibung**. `mv` nimmt die Beschreibung eines Secrets mit, `rm` entfernt sie. Höchstens 200 Zeichen, eine Zeile.

Auflisten

```
mlcsecure ls            # alles
mlcsecure ls prod       # eine Gruppe
mlcsecure ls -d         # mit Beschreibungen (ohne Passphrase)
mlcsecure ls -l         # mit Art, Policy, Dateiname, Beschreibung (fragt nach der Passphrase)
mlcsecure show myapp/config   # ein Secret im Detail: Größe, letzte Änderung, Schlüssel einer Datei
```

Werte werden nie angezeigt. Art und Dateiname sind ebenfalls verschlüsselt, deshalb brauchen `-l` und `show` die Passphrase (oder den entsperrten Agent).

Bei einer gespeicherten `.env`-, INI-, TOML- oder JSON-Datei listet `show` die Namen ihrer Schlüssel auf — praktisch, wenn die Datei lang ist oder du nicht mehr weißt, was drinsteht:

```
path         myapp/config
kind         file  config.toml
size         1.2 KiB, 48 lines, toml
changed      2026-11-10 14:02  4bfcc49  set myapp/config
description  myapp: Server und Datenbank
keys         12
  title
  server.port
  db.url
  db.password
  …
```

Es erscheinen nur Namen, die wie Schlüssel aussehen; eine Zeile, die `show` nicht versteht, wird gezählt, nicht gezeigt — sie könnte ein Wert sein. Bei einem einzelnen Wert lässt `show` die Größe weg: Die Länge eines Passworts ist ein Hinweis darauf.

Umbenennen und bearbeiten

```
mlcsecure mv prod/db/pass prod/db/password   # Ziel darf nicht existieren
mlcsecure edit prod/db/password
```

`edit` entschlüsselt das Secret in eine private Datei im Arbeitsspeicher (tmpfs, nie auf der Festplatte), öffnet sie in `$VISUAL`, `$EDITOR` oder `vi` und speichert nur, wenn du etwas geändert hast. Bei Werten wird der Zeilenumbruch, den dein Editor am Ende anfügt, entfernt. Die temporäre Datei wird danach gelöscht.

Wo Klartext kurz liegt (`edit`, `exec --file`):

System	Ort	Hinweis
Linux	`$XDG_RUNTIME_DIR` oder `/dev/shm`	nur im Arbeitsspeicher (tmpfs); nie auf der Festplatte
macOS	dein privater Temp-Ordner (`$TMPDIR`, Modus 0700)	macOS hat kein tmpfs: Das liegt auf der Festplatte — verschlüsselt, wenn FileVault an ist — und wird direkt nach der Nutzung gelöscht
Windows	`%TEMP%\mlcsecrets`, nur dein Benutzer und SYSTEM dürfen ihn öffnen	auf der Festplatte — verschlüsselt, wenn BitLocker an ist — und direkt nach der Nutzung gelöscht
andere	—	`edit` und `exec --file` verweigern

Editoren und Klartextkopien

`edit` hält seine Kopie im Arbeitsspeicher, aber manche Editoren legen eigene Kopien an, in deinem Profil auf der Festplatte, wo mlcsecure sie nicht löschen kann:

Editor	Was er aufbewahrt	Was tun
vim, neovim	Swap-, Undo- und Backup-Dateien; viminfo/shada speichert kopierten Text	nichts: `edit` startet sie mit `-n -i NONE` und ohne Undo- und Backup-Dateien
nano	nichts, außer du hast `set backup` eingeschaltet	—
VS Code, VSCodium, Cursor	die Local History speichert jede gespeicherte Fassung; ungespeicherte Änderungen werden gesichert	`"workbench.localHistory.exclude": {"**/mlcsecrets/s-*/**": true}` in die Benutzereinstellungen eintragen und vor dem Schließen des Tabs speichern
JetBrains-IDEs	die Local History speichert jede gespeicherte Fassung	für Secrets einen Terminal-Editor nehmen

Editor	Was er aufbewahrt	Was tun
Sublime Text, Notepad++	die Sitzung speichert ungespeicherte Änderungen	vor dem Schließen speichern oder Hot Exit / Sitzungs-Snapshots ausschalten
Editor/Notepad (Windows 11)	merkt sich ungespeicherte Tabs	vor dem Schließen speichern

`edit` gibt einen Hinweis aus, wenn es einen der Editoren startet, die Kopien anlegen. Für wichtige Dateien ist ein Terminal-Editor die sichere Wahl: `EDITOR=vim mlcsecure edit myapp/config`.

Löschen

```
mlcsecure rm openai/api_key
```

Das Secret verschwindet aus dem Store, bleibt aber in der Git-Historie.

Verlauf und Rollback

Jede Änderung ist ein Git-Commit, also ist jeder frühere Wert noch da:

```
mlcsecure history prod/db/password
# 4f2a91c  2026-10-24 16:40  Michael  update prod/db/password
# 9be0d13  2026-10-20 09:12  Michael  add prod/db/password

mlcsecure rollback prod/db/password 9be0d13
```

`history` zeigt Daten und Nachrichten, nie Werte, und braucht keine Passphrase. `rollback` holt den alten Wert als neuen Commit zurück — nichts wird umgeschrieben — und verschlüsselt ihn für die heutigen Rechner, sodass auch ein inzwischen hinzugekommener Rechner ihn lesen kann. Es stellt auch gelöschte Secrets wieder her: Ihre Löschung erscheint in `history`, nimm den Commit davor. Ein mit `mv` umbenanntes Secret beginnt einen neuen Verlauf; frühere Versionen liegen unter seinem alten Namen.

Ein Secret lesen

```
mlcsecure get prod/db/password
```

Fragt nach der Passphrase und gibt den Wert aus. Gespeicherte Dateien werden nicht ausgegeben; dafür gibt es `--out`. Damit Secrets nicht in Logs und nicht in Chatverläufe von KI-Assistenten geraten, **verweigert `get` die Ausgabe in eine Pipe oder eine Dateiumleitung**. Brauchst du den Wert in einer Datei:

```
mlcsecure get certs/server --out ./server.pem     # wird mit Modus 0600 angelegt
```

`--out` überschreibt nie eine vorhandene Datei.

Einen Befehl mit Secrets ausführen

So nutzt du Secrets hauptsächlich in Skripten: Der Befehl bekommt sie, dein Terminal und deine Logs nicht.

```
mlcsecure exec --group openai -- python run.py
mlcsecure exec --env-file myapp/env -- npm start
mlcsecure exec -e PGPASSWORD=prod/db/password -- psql -h db -U app
mlcsecure exec -f KUBECONFIG=certs/kube -- kubectl get pods
mlcsecure exec -f CONFIG=myapp/config -- myapp --config '$CONFIG'
```

Option	Wirkung
`-g, --group GROUP`	jedes Secret unterhalb von `GROUP` wird zu einer Variablen, benannt nach dem letzten Pfadteil in Großbuchstaben: `prod/db/password` → `PASSWORD`, `myapp/openai_api_key` → `OPENAI_API_KEY`

Option	Wirkung
`--prefix P`	stellt `P` vor die Namen aus Gruppen (`--prefix DB_` → `DB_PASSWORD`)
`--env-file PATH`	`PATH` ist eine gespeicherte `.env`-Datei; jede Zeile `KEY=VALUE` wird zu einer Variablen
`-e, --env NAME=PATH`	`NAME` bekommt den Wert von `PATH`
`-f, --file NAME=PATH`	der Inhalt von `PATH` wird in eine private Datei im Arbeitsspeicher (tmpfs) geschrieben; `NAME` bekommt ihren Pfad

Alle Optionen lassen sich wiederholen. Gespeicherte Dateien, die über `--group` gefunden werden, werden immer als Dateien übergeben. Würden zwei Gruppen denselben Namen setzen, bricht `exec` ab und bittet dich, `--env` oder `--prefix` zu verwenden. Eine `.env`-Datei ersetzt einen Namen, der aus einer Gruppe kam, ein ausdrückliches `--env`/`--file` ersetzt beide. Vorhandene Umgebungsvariablen mit demselben Namen werden ersetzt.

In den Argumenten des Befehls wird `$NAME` oder `${NAME}` einer `--file`-Variablen durch den Pfad ihrer temporären Datei ersetzt — schreib es in einfache Anführungszeichen (`'$CONFIG'`), damit deine Shell es für mlcsecure stehen lässt. In PowerShell geht das genauso; in cmd schreibst du `$CONFIG` ohne Anführungszeichen. In Argumente kommen nur Dateipfade, nie Werte: Argumente sieht jedes Programm auf dem Rechner (`ps`).

Alles nach `--` ist der Befehl, einschließlich seiner eigenen Optionen. Die temporären Dateien werden gelöscht, wenn der Befehl endet — auch wenn du Ctrl-C drückst oder der Prozess beendet wird; diese Signale werden an den Befehl weitergereicht. `exec` endet mit dem Exit-Status des Befehls (128+n, wenn er durch Signal n beendet wurde).

`exec` fragt nach deiner Passphrase — außer der Agent ist entsperrt (nächster Abschnitt).

Was du siehst: Maskierung

- **Ausgabe in dein Terminal wird gezeigt, wie sie ist** — du siehst genau, was der Befehl ausgibt, Farben und Fortschrittsbalken eingeschlossen.
- **Ausgabe in eine Pipe oder eine Datei** — ein Log, `| tee`, das Werkzeug eines KI-Assistenten — hat jeden Secret-Wert durch `***path***` ersetzt, zum Beispiel `token=***prod/api/token***`. Ebenfalls erkannt: der Wert in base64 (alle gängigen Varianten), URL-kodiert und hex; bei gespeicherten Dateien der ganze Inhalt und jede Zeile ab 16 Zeichen.
- stdout und stderr werden getrennt beurteilt: `exec … 2>err.log` zeigt stdout auf dem Bildschirm und maskiert das Log.
- Werte unter 4 Zeichen werden nicht maskiert (sie würden überall normale Wörter ersetzen); `exec` nennt sie auf stderr.
- `--mask` oder `MLCSECURE_MASK=always` maskiert auch dein Terminal — nützlich beim Bildschirmteilen oder für KI-Werkzeuge, die Befehle in einem Pseudo-Terminal ausführen.

Einen Schalter, der die Maskierung für eine Pipe abschaltet, gibt es bewusst nicht. Brauchst du die unmaskierte Ausgabe wirklich in einer Datei, leite *innerhalb* des Befehls um: `mlcsecure exec -g prod/db -- sh -c './dump.sh > out.txt'`.

> Die Maskierung erkennt Secrets, die unverändert oder in einer gängigen Kodierung ausgegeben werden. Ein Programm, das sie auf andere Weise umformt (umdreht, aufteilt, verschlüsselt), kommt daran vorbei. Sie schützt vor Versehen, nicht vor einem Programm, das es aufs Leaken anlegt.

Konfigurationsdateien und bestehende Skripte

Die Einstellungen eines Programms stehen oft in einer großen Datei: `.env`, `config.toml`, `app.ini`, `settings.json`. Leg diese Datei als Ganzes im Store ab: Du bearbeitest sie wie gewohnt in deinem Editor, sie wird verschlüsselt gespeichert, und das Programm bekommt sie nur, solange es läuft.

Eine Konfigurationsdatei sichern

```
mlcsecure add-file myapp/config ./config.toml --desc "myapp: Server und Datenbank"
rm ./config.toml                    # der Klartext wird nicht mehr gebraucht
mlcsecure show myapp/config         # was drinsteht, ohne Werte
mlcsecure edit myapp/config         # im Editor ändern
mlcsecure history myapp/config      # jede Änderung; bei Bedarf rollback
```

Das geht genauso mit `.env`, `.ini`, `.json`, `.yaml` oder jeder anderen Datei bis 10 MiB. `edit` öffnet die Datei unter ihrem ursprünglichen Namen, deshalb hebt dein Editor sie als TOML, INI usw. hervor. Lies einmal Editoren und Klartextkopien.

Sie einem Programm übergeben

Das Programm liest …	Befehl
einen Pfad als Option (`--config DATEI`)	`mlcsecure exec -f CONFIG=myapp/config -- myapp --config '$CONFIG'`
einen Pfad aus einer Variablen (`KUBECONFIG`, `GOOGLE_APPLICATION_CREDENTIALS`)	`mlcsecure exec -f KUBECONFIG=certs/kube -- kubectl get pods`
Umgebungsvariablen (wofür eine `.env`-Datei da ist)	`mlcsecure exec --env-file myapp/env -- npm start`
einen festen Pfad wie `./config.toml`	siehe Ein Programm mit festem Konfigurationspfad

`-f` schreibt die Datei unter ihrem ursprünglichen Namen in den Arbeitsspeicher (tmpfs) und löscht sie, wenn das Programm endet. `--env-file` und `-f` lassen sich kombinieren und wiederholen.

.env-Dateien: als Ganzes oder aufgeteilt

Es gibt zwei Wege, eine `.env`-Datei abzulegen. Beide geben dem Programm dieselben Variablen.

	Ganze Datei	Ein Secret pro Zeile
ablegen	`mlcsecure add-file myapp/env .env`	`mlcsecure import-env myapp .env`
nutzen	`exec --env-file myapp/env -- …`	`exec --group myapp -- …`
bearbeiten	`edit myapp/env` — die ganze Datei	`edit myapp/openai_api_key` — ein Wert
Policies, Freigaben, Verlauf	für die ganze Datei	für jeden Wert
gut für	viele Einstellungen, die zusammengehören	ein paar Schlüssel, die du auch anderswo nutzt

Ein bestehendes Skript umstellen

Meist bleibt das Skript, wie es ist: Du startest es über `mlcsecure exec`, und das Skript und alles, was es startet, sieht die Variablen. Nur die Zeile, die die Klartextdatei liest, fällt weg.

Ein Shell-Skript, das `.env` lädt:

```
# vorher: deploy.sh beginnt mit  set -a; . ./.env; set +a
./deploy.sh
# nachher: diese Zeile löschen (oder optional machen:  [ -f .env ] && . ./.env)
mlcsecure exec --env-file myapp/env -- ./deploy.sh
```

Python mit `python-dotenv`, Node mit `dotenv`: `load_dotenv()` und `require('dotenv').config()` tun nichts, wenn es keine `.env`-Datei gibt, und ersetzen nie Variablen, die schon gesetzt sind. Der Code bleibt; nur der Start ändert sich:

```
mlcsecure exec --env-file myapp/env -- python app.py
mlcsecure exec --env-file myapp/env -- node server.js
```

Ein Programm, das seine Konfigurationsdatei als Option bekommt:

```
# vorher
myapp serve --config ./config.toml
# nachher
mlcsecure exec -f CONFIG=myapp/config -- myapp serve --config '$CONFIG'
```

In einem Taskfile setzt du `mlcsecure exec` vor den Befehl:

```
tasks:
  run:
    cmds:
      - mlcsecure exec --env-file myapp/env -f CONFIG=myapp/config -- myapp serve --config '$CONFIG'
```

In einem Makefile muss `$` verdoppelt werden: `-- myapp serve --config '$$CONFIG'`.

docker compose: Statt `env_file: .env` führst du die Variablennamen unter `environment:` auf; compose nimmt ihre Werte aus seiner eigenen Umgebung:

```
services:
  app:
    environment:
      - DB_URL
      - API_TOKEN
```

```
mlcsecure exec --env-file myapp/env -- docker compose up -d
```

Beachte, dass Docker die Umgebung eines Containers in seiner Konfiguration speichert (`docker inspect` zeigt sie) — das liegt außerhalb von mlcsecure.

Ein Programm mit festem Konfigurationspfad

Viele Programme nehmen den Pfad auch über eine Variable oder eine Option — prüf das zuerst. Liest ein Programm immer nur `./config.toml`, bind die Datei in einen Container ein oder verlinke sie für die Laufzeit des Programms (Linux und macOS):

```
mlcsecure exec -f CONFIG=myapp/config -- docker run --rm -v '$CONFIG:/app/config.toml:ro' myapp
mlcsecure exec -f CONFIG=myapp/config -- sh -c 'ln -sf "$CONFIG" config.toml; myapp; rc=$?; rm -f
  config.toml; exit $rc'
```

Der Link zeigt in den Arbeitsspeicher und wird danach entfernt; trag `config.toml` trotzdem in `.gitignore` ein.

Einmal entsperren: der Agent

Die Passphrase bei jedem `exec` zu tippen, nervt auf Dauer, und KI-Assistenten oder Skripte ohne Terminal können sie gar nicht tippen. Entsperr stattdessen einmal:

```
mlcsecure agent start            # fragt nach der Passphrase, läuft im Hintergrund
mlcsecure agent start --ttl 2h   # sperrt automatisch nach 2 Stunden (Standard 8h)
mlcsecure agent status           # "unlocked (pid …), locks in 7h59m at 18:30"
mlcsecure agent lock             # jetzt sperren (auch: agent stop)
```

Solange der Agent läuft, fragen `exec`, `get`, `edit` und `ls -l` nicht mehr nach der Passphrase. Der Agent hält deinen Schlüssel nur im Arbeitsspeicher; der Schlüssel verlässt ihn nie — andere `mlcsecure`-Prozesse schicken ihm verschlüsselte Daten und bekommen den Klartext über einen Socket zurück, den nur dein Benutzer öffnen kann (`$XDG_RUNTIME_DIR/mlcsecrets/agent.sock`; unter Windows `%LOCALAPPDATA%\mlcsecrets\agent.sock`). Ist die Zeit abgelaufen oder sperrst du ihn, vergisst der Agent den Schlüssel und beendet sich. `agent status` endet mit 3, wenn gesperrt ist, damit Skripte das prüfen können.

> **Solange der Agent entsperrt ist, kann jedes Programm, das unter deinem Benutzer läuft, deine Secrets über `mlcsecure` entschlüsseln** — genau das macht ihn für Skripte und KI-Assistenten nützlich. Halte die TTL kurz und sperr ihn, wenn du fertig bist.

Meldet `agent start`, dass der Socket-Pfad zu lang ist (Unix-Sockets erlauben etwa 100 Zeichen), setz `MLCSECURE_AGENT_SOCK` auf einen kürzeren Pfad.

Zum Betrieb unter systemd oder in einem eigenen Terminal: `mlcsecure agent start --foreground`. Ein anderer Socket-Pfad lässt sich mit `MLCSECURE_AGENT_SOCK` setzen.

Exec-only-Gruppen

Manche Secrets sollen nutzbar sein, aber nie angezeigt werden — nicht dir aus Versehen beim Bildschirmteilen und nicht einem KI-Assistenten:

```
mlcsecure policy set prod exec-only    # bittet dich, zur Bestätigung "prod" zu tippen
mlcsecure policy ls
mlcsecure policy set prod none         # wieder aufheben
```

Für jedes Secret unterhalb von `prod` verweigern `get`, `get --out` und `edit` (Exit-Code 2), während `exec` und `ls -l` weiter funktionieren. Einen Wert kannst du mit `set` trotzdem ersetzen. Die Bestätigung muss am Terminal getippt werden, daher können Skripte und KI-Assistenten keine Policy ändern. Policies liegen im Store (`.mlcs/policy`) und werden mit ihm versioniert.

> Ein Programm mit vollem Zugriff auf dein Konto könnte diese Datei trotzdem bearbeiten oder den Agent direkt benutzen. `exec-only` hält ehrliche Werkzeuge und Versehen davon ab, Werte preiszugeben; eine Sandbox ist es nicht.

On-demand-Secrets, befristete Freigaben und Sperren

Manche Secrets sollen normalerweise unerreichbar bleiben — selbst für `exec` — und nur für einen Moment erlaubt sein, zum Beispiel ein privater SSH-Schlüssel, den ein KI-Assistent für ein einziges Deployment braucht.

```
mlcsecure policy set ssh on-demand          # einmalig; zur Bestätigung "ssh" tippen
mlcsecure grant ssh/deploy_key --for 15m    # bei Bedarf; zur Bestätigung "yes" tippen
mlcsecure grants                            # was gerade freigegeben oder gesperrt ist
mlcsecure revoke ssh/deploy_key             # vorzeitig beenden
```

Ein **on-demand**-Secret wird von `exec`, `ssh-add`, `get` und `edit` verweigert, solange keine Freigabe dafür (oder für seine Gruppe) läuft. Hast du mit deiner Passphrase am Terminal entsperrt, bist du anwesend, und on-demand gilt für dich nicht; es gilt für alles, was über den Agent läuft — Skripte und KI-Assistenten.

Das Gegenteil geht für jedes Secret und jede Gruppe:

```
mlcsecure block prod --for 2h      # prod/* für alle verweigern, auch für dich
mlcsecure unblock prod             # zur Bestätigung "prod" tippen
```

Freigaben und Sperren leben **nur im Arbeitsspeicher des Agents** auf diesem Rechner: Sie werden nicht synchronisiert und enden, wenn ihre Zeit abgelaufen ist oder der Agent sperrt (`agent lock` oder seine `--ttl`). Sie brauchen deshalb einen laufenden Agent. Freigeben und das Aufheben einer Sperre müssen am Terminal bestätigt werden; Widerrufen und Sperren — die die Lage nur strenger machen — brauchen keine Bestätigung.

SSH-Schlüssel ohne Schlüsseldateien

```
mlcsecure add-file ssh/deploy_key ~/.ssh/deploy_key --desc "Deploy key for server"
rm ~/.ssh/deploy_key
mlcsecure ssh-add ssh/deploy_key --for 10m
ssh deploy@server                  # funktioniert 10 Minuten, dann vergisst ssh-agent den Schlüssel
```

`ssh-add` leitet den Schlüssel direkt in deinen laufenden `ssh-agent` (`ssh-add -t`), er wird also nie zu einer Datei. Es gelten dieselben Regeln wie für `exec`: on-demand braucht eine Freigabe, Sperren verweigern. Der gespeicherte Schlüssel darf keine eigene Passphrase haben — MLC Secrets verschlüsselt ihn ohnehin.

Konfigurationsdateien von Servern

Die geheimen Teile der Konfiguration eines Servers — WireGuard- und VPN-Configs, private TLS-Schlüssel, `.htpasswd`, `.env`-Dateien mit Passwörtern — gehören meist root. mlcsecure speichert sie zusammen mit Ort, Modus und Besitzer und legt sie genau so wieder zurück:

```
# von diesem Rechner (sudo nur, wenn du die Datei nicht lesen kannst)
mlcsecure backup-file local/nginx-key /etc/nginx/ssl/key.pem

# von einem Server, über ssh und "sudo -n"
mlcsecure backup-file vps/wg0 /etc/wireguard/wg0.conf --host vps --desc "WireGuard on vps"

mlcsecure check vps/wg0      # identisch? – vergleicht Prüfsumme, Modus und Besitzer, zeigt nie Inhalt
mlcsecure restore vps/wg0    # zeigt, was es tun wird, fragt nach "yes", schreibt atomar
```

- Der Inhalt läuft nur durch Pipes; unterwegs wird er nie auf eine Festplatte geschrieben. `restore` schreibt eine temporäre Datei neben das Ziel, setzt Besitzer und Modus und benennt sie über das Original um — ein fehlgeschlagenes Restore lässt die alte Datei unangetastet.

- `restore` tut nichts, wenn die Datei schon identisch ist. `--to PATH` und `--host H` stellen woanders wieder her; `--yes` überspringt die Frage (Skripte, Rezepte).
- `check` endet mit **5**, wenn die Datei abweicht oder fehlt — praktisch in einem Cron-Job oder einem Rezept, um zu bemerken, dass jemand einen Server geändert hat.
- Auf einem Server muss sudo dafür ohne Passwort funktionieren (`sudo -n`), weil ssh ohne Terminal läuft. Andernfalls meld dich als Benutzer an, der die Datei lesen darf, und ergänze `--no-sudo` (z. B. `--host root@vps --no-sudo`).
- Eine Datei, die du als Datenstrom statt als Pfad hast: `ssh vps sudo cat /etc/x.key | mlcsecure add-file vps/x --name x.key` (ohne Ort gespeichert; wiederherstellen mit `exec -f`).
- Gedacht für die geheimen Teile. Für den Rest von `/etc` ist ein Konfigurationsmanagement-Werkzeug oder etckeeper der bessere Ort.

Führ mlcsecure selbst nicht mit sudo aus. Als root würde es die Identität und den Store von root verwenden, nicht deine; es warnt davor (zum Stummschalten `MLCSECURE_ALLOW_ROOT=1`, wenn du das wirklich willst). Für Dateien, die root gehören, ruft es `sudo` selbst auf.

Rezepte und der MCP-Server

KI-Werkzeuge, die keine Shell-Befehle ausführen können (Claude Desktop, Chat-Clients, eigene Agenten), können deine Secrets über den **MCP-Server** nutzen. Er ist bewusst eng: Ein Agent kann Namen und Beschreibungen auflisten, sehen, ob der Tresor entsperrt ist, und **Rezepte** ausführen — Befehle, die *du* festgelegt hast. Er kann nie einen Wert lesen, nichts anderes ausführen und weder Secrets noch Regeln, Freigaben oder Rezepte ändern.

Rezepte festlegen

```
mlcsecure recipe add deploy-barcode \
  --desc "Signed Android release of mlc-barcode" \
  --dir /mnt/data2tb/mlc_barcode \
  -g android/mlc-barcode -f ANDROID_KEYSTORE_FILE=android/mlc-barcode/keystore \
  --param 'channel=beta|stable' --param-desc 'channel=release channel' \
  --timeout 30m --confirm \
  -- task deploy {channel}
```

Dir wird das vollständige Rezept gezeigt, und du musst zur Bestätigung seinen Namen tippen. Regeln:

- Das Programm und seine festen Argumente bestimmst du; ein Agent kann nur `{param}`-Platzhalter füllen, und jeder muss ein **ganzes Argument** sein (es ist keine Shell beteiligt, also haben `;`, `$(…)` und Anführungszeichen keine besondere Bedeutung).
- Jeder Wert muss vollständig zu seinem Muster passen (`beta|stable` erlaubt genau diese zwei Wörter), darf nicht länger als 256 Zeichen sein, keine Zeilenumbrüche enthalten und nicht mit `-` beginnen (außer du erlaubst das mit `--allow-dash NAME`). Mit `--optional NAME` darf ein Parameter weggelassen werden.
- Secrets kommen wie bei `exec` hinein: `-g` Gruppen, `-e` Werte, `-f` Dateien. On-demand-Regeln und Sperren gelten wie gewohnt.
- `--confirm` lässt den Server vor jedem Lauf **dich** fragen — der MCP-Client zeigt die Frage mit der genauen Befehlszeile; die KI kann sie nicht beantworten.
- Standard-Zeitlimit 10 Minuten (`--timeout`, höchstens 24h); bei Zeitüberschreitung wird die ganze Prozessgruppe beendet.

```
mlcsecure recipe ls                # Name, Parameter, Beschreibung
mlcsecure recipe show deploy-barcode
mlcsecure recipe rm deploy-barcode
```

Rezepte gehören zu einem Rechner (`~/.config/mlcsecrets/recipes.json`): Pfade unterscheiden sich zwischen Linux und Mac, und was ein Agent ausführen darf, wird pro Maschine entschieden. Sie werden nicht synchronisiert.

Einen Client anbinden

`mlcsecure setup --mcp --yes` registriert den Server bei den Agenten auf diesem Rechner (siehe setup). Von Hand:

```
{ "mcpServers": { "mlcsecure": { "command": "mlcsecure", "args": ["mcp"] } } }
```

Der Client startet den Server selbst als `mlcsecure mcp`; er läuft nur über stdio — kein Netzwerk-Port. Er entschlüsselt über den Agent und fragt nie nach einer Passphrase: Starte den Agent (`mlcsecure agent start`), bevor du eine KI arbeiten lässt.

Werkzeug	Was es tut
`status`	ist der Agent entsperrt, bis wann; aktive Freigaben und Sperren
`list_secrets`	Namen, Beschreibungen und Regeln (optional unterhalb einer Gruppe) — nie Werte
`show_secret`	ein Secret wie `mlcsecure show`: Art, Regel, letzte Änderung und die Schlüsselnamen einer gespeicherten `.env`-, INI-, TOML- oder JSON-Datei — nie Werte; braucht den Agent
`list_recipes`	deine Rezepte mit ihren Parametern und den Secrets, die sie bekommen
`run_recipe`	führt ein Rezept aus; liefert Exit-Code und Ausgabe, Secrets **immer** maskiert, höchstens 64 KiB pro Stream

Ein Prompt `use_secrets` führt ein Modell durch eine Aufgabe. Die vollständige Referenz mit allen Parametern wird aus dem laufenden Server erzeugt: docs/mcp-tools.md (englisch).

Jeder Lauf — und jede Ablehnung — wird in `~/.config/mlcsecrets/mcp-audit.jsonl` geschrieben (Zeit, Rezept, Argumente, Ergebnis, Exit-Code, Dauer; nie Werte).

> Der MCP-Server schützt den **MCP-Weg**. Eine KI, die zusätzlich eine Shell auf deinem Konto hat, kann die Rezeptdatei bearbeiten oder `mlcsecure` direkt aufrufen — für solche Werkzeuge gelten die Regeln in SKILLS.md (englisch) und die übrigen Schutzmechanismen (Maskierung, exec-only, on-demand).

Sicherung und Synchronisierung mit einem Remote-Repository

Der Store enthält nur Chiffretext, deshalb kann er als normales Git-Repository auf einem Server liegen, den du kontrollierst:

```
mlcsecure remote set mlc@nas.local:/volume1/homes/mlc/repositories/secrets-store.git
mlcsecure remote          # anzeigen
mlcsecure sync            # pull, dann push: "pulled 0, pushed 12 change(s)"
```

Arbeitest du nur auf einem Rechner? Dann kann der Remote genauso gut ein Bare-Repository auf einer lokalen Platte sein, etwa einem zweiten Laufwerk:

```
git init --bare -b main /mnt/backup/secrets-store.git
mlcsecure remote set /mnt/backup/secrets-store.git
mlcsecure sync
```

Es ist dann neben dem Store deine einzige Kopie, also **nimm es in deine reguläre Datensicherung auf** — ein Remote auf derselben Platte wie der Store schützt vor Fehlbedienung, nicht vor einer defekten Platte.

Änderungen werden sofort lokal committet; `sync` tauscht sie mit dem Remote aus. Führ es aus, nachdem du Secrets geändert hast und bevor du sie auf einem anderen Rechner nutzt. Änderungen an verschiedenen Secrets auf zwei Rechnern werden automatisch zusammengeführt. Wurde **dasselbe** Secret auf beiden geändert, ändert `sync` nichts, nennt das Secret und hält an:

```
mlcsecure: sync conflict in prod/db/password — local state unchanged; rerun with --prefer local or --prefer
  remote
```

Entscheide mit `mlcsecure sync --prefer local` oder `--prefer remote`, welcher Wert gewinnt. Der andere Wert geht nicht verloren: Er bleibt in `history`.

Automatische Synchronisierung

Statt `sync` selbst auszuführen, lass das jeden Rechner tun:

```
mlcsecure config set sync.auto write    # nach jeder Änderung synchronisieren
mlcsecure config set sync.auto always   # …und zusätzlich vor dem Lesen pullen
mlcsecure config                        # Einstellungen dieses Rechners anzeigen
```

`sync.auto`	Wirkung
`off` (Standard)	nur `mlcsecure sync` spricht mit dem Remote
`write`	nach jeder erfolgreichen Änderung (`set`, `add-file`, `import-env`, `mv`, `edit`, `rm`, `rollback`, `describe`, `policy set`, `recipients add/rm`)
`always`	wie `write`, plus ein Pull vor `get`, `ls`, `exec`, `history` und vor jeder Änderung — die neuesten Werte des anderen Rechners sind immer da, um den Preis eines Round-Trips zum Remote pro Befehl

Automatische Synchronisierung lässt nie einen Befehl fehlschlagen: Ist der Remote nicht erreichbar oder gibt es einen Konflikt, bleibt die Änderung lokal committet, und eine Warnung auf stderr sagt dir, dass du später `mlcsecure sync` ausführen sollst. Die Ausgabe auf stdout ist nicht betroffen, Skripte, die `ls` lesen, funktionieren also weiter. Ohne Remote passiert nichts.

Einstellungen gehören zu einem Rechner und liegen neben seiner Identität (`~/.config/mlcsecrets/config`, einfache `key = value`-Zeilen).

Was der Remote sehen kann: die Namen deiner Secrets, wann sie sich geändert haben, und die Autorennamen aus deiner Git-Konfiguration. Nicht die Werte.

Ein zweiter Rechner

Checkliste: ein neuer Rechner

Die ganze Runde, Einzelheiten weiter unten. *Alt* ist ein Rechner, der schon entschlüsseln kann, *neu* der, der hinzukommt. Wie die Teile zusammenspielen — mit Schaubildern — steht in ARCHITECTURE.de.md.

- ☐ Installieren: auf *alt* (mit dem Quellcode) `task deploy HOST=new` — oder `task install` auf *neu*
- ☐ Auf *neu*: `mlcsecure setup --yes` — der Skill für seine KI-Agenten
- ☐ Auf *neu*, in einem Terminal: `mlcsecure init --clone URL` — die Passphrase dieses Rechners wählen
- ☐ Auf *alt*: `mlcsecure recipients add "$(ssh new mlcsecure key)" --name new`, `yes` tippen, dann `mlcsecure sync`
- ☐ Auf *neu*: `mlcsecure sync`, dann `mlcsecure ls -l` — Werte sind lesbar
- ☐ Auf *neu*: `mlcsecure config set sync.auto write`, wenn du automatische Synchronisierung willst
- ☐ Auf *neu*: `mlcsecure agent start`, wenn eine KI Secrets nutzen soll

Schritt für Schritt

Jeder Rechner hat seine eigene Identität (Schlüssel + Passphrase); der private Schlüssel verlässt ihn nie. Ein zweiter Rechner kommt hinzu, indem er den Store klont und auf einem Rechner, der schon entschlüsseln kann, als *Empfänger* eingetragen wird:

```
# 1. auf dem ersten Rechner (schon eingerichtet, mit Remote):
mlcsecure sync

# 2. auf dem neuen Rechner:
mlcsecure init --clone mlc@nas.local:/volume1/homes/mlc/repositories/secrets-store.git
#    → fragt nach einer Passphrase für diesen Rechner und gibt seinen öffentlichen Schlüssel aus

# 3. zurück auf dem ersten Rechner — holt den Schlüssel über ssh, ohne Kopieren:
mlcsecure recipients add "$(ssh macstudio mlcsecure key)" --name macstudio   # "yes" tippen
mlcsecure sync

# 4. auf dem neuen Rechner:
mlcsecure sync
mlcsecure ls -l                                  # funktioniert jetzt
```

`recipients add` verschlüsselt **jedes** Secret neu, sodass der neue Schlüssel es lesen kann, in einem einzigen Commit. Weil das Zugriff auf alles vergibt, muss es am Terminal bestätigt werden. Ältere Versionen im Verlauf bleiben für die Rechner ihrer Zeit verschlüsselt; `rollback` verschlüsselt für die heutigen neu.

`mlcsecure key` gibt jederzeit den öffentlichen Schlüssel eines Rechners aus; er ist nicht geheim und braucht keine Passphrase. (Vor 0.11 eingerichtet? Dann fragt es einmal nach der Passphrase und merkt sich den Schlüssel in `identity.age.pub`.) Findet `ssh mlcsecure` auf dem anderen Rechner nicht, nimm den vollen Pfad, z. B. `ssh macstudio .local/bin/mlcsecure key`.

```
mlcsecure recipients                 # Liste: Name und Schlüssel
mlcsecure recipients rm macstudio    # nicht mehr für ihn verschlüsseln (mit dem Namen bestätigen)
```

Nach `recipients rm` sind neue Versionen für diesen Rechner unlesbar, aber er kann weiterhin **alte Versionen** in der Git-Historie lesen, die er schon hat. Ändere die Secrets, die er nicht mehr kennen darf. Der letzte Empfänger kann nicht entfernt werden.

Wiederherstellungsschlüssel

Der Schlüssel jedes Rechners ist durch seine Passphrase geschützt, und es gibt kein Zurücksetzen. Gehen alle Rechner verloren — oder sind alle Passphrasen vergessen —, lässt sich der Store nicht mehr entschlüsseln. Ein offline aufbewahrter **Wiederherstellungsschlüssel** verhindert das:

```
mlcsecure key generate --out recovery.txt         # gibt den öffentlichen Schlüssel age1… aus
mlcsecure recipients add age1… --name recovery    # "yes" tippen; verschlüsselt alles neu
mlcsecure sync
```

Leg `recovery.txt` jetzt in deinen Passwortmanager (oder druck sie aus) und lösch sie von der Festplatte. Es ist ein normaler age-Schlüssel: Auch ohne MLC Secrets entschlüsselt `age -d -i recovery.txt <file>.age` eine gespeicherte Datei.

Wiederherstellen auf einem neuen Rechner:

```
mlcsecure key import recovery.txt                # wird die Identität dieses Rechners (neue Passphrase)
mlcsecure init --clone mlc@nas.local:/volume1/homes/mlc/repositories/secrets-store.git
```

Der Wiederherstellungsschlüssel kann alles lesen, wie jeder Rechner. Behandle ihn wie den Generalschlüssel, der er ist — und erwäge `recipients rm recovery` plus einen neuen, falls er offengelegt worden sein könnte.

Exit-Codes

Code	Bedeutung
0	Erfolg
1	Fehler
2	falsche Benutzung (unbekannter Befehl oder unbekanntes Flag, falsche Zahl von Argumenten, ungültiger Pfad oder ungültige Revision, kein Terminal, verweigerte Ausgabe, Ziel existiert, zu groß, exec-only, on-demand ohne Freigabe, gesperrt)
3	gesperrt — falsche Passphrase, keine Identität, Agent läuft nicht (`agent status`) oder kein Agent und kein Terminal für die Passphrase (Skripte, KI-Agenten)
4	Secret nicht gefunden
5	`check`: die aktuelle Datei weicht vom Backup ab oder fehlt

Skripte können sich darauf verlassen. `exec` liefert stattdessen den Exit-Status des ausgeführten Befehls, sobald dieser gestartet ist.

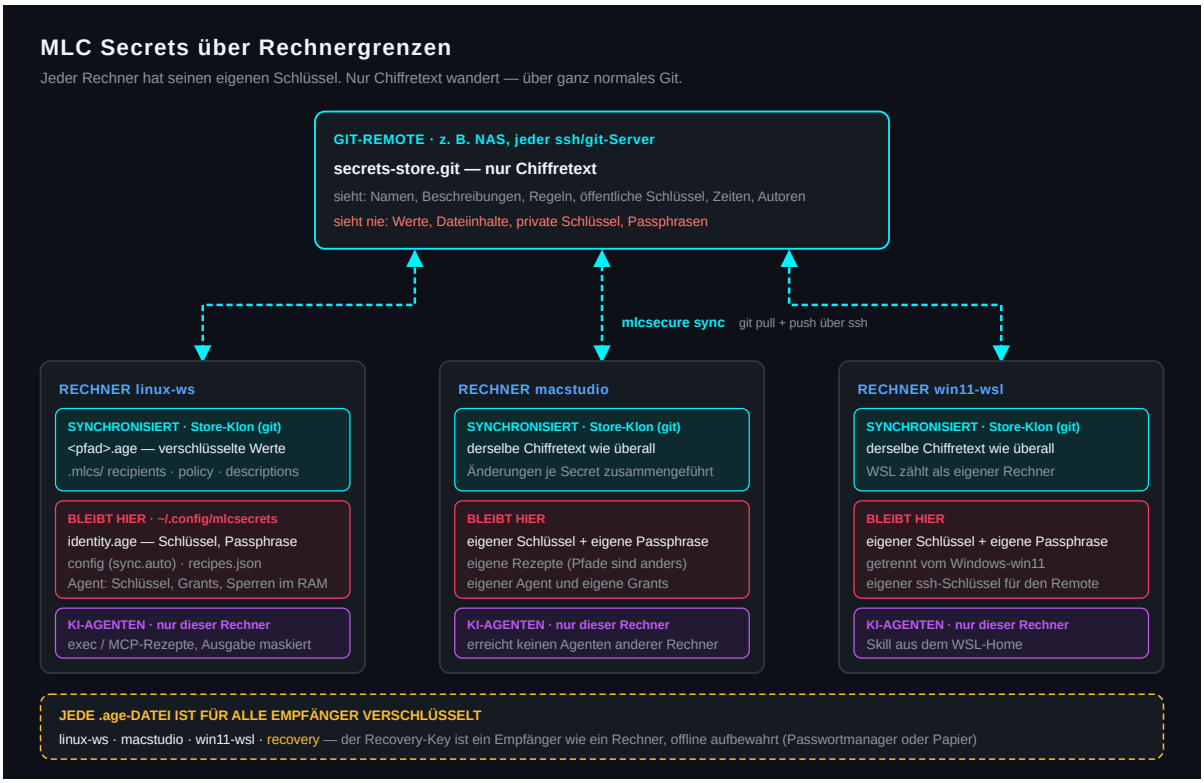
Sicherheitshinweise

- Verschlüsselt wird mit age (X25519); die Identitätsdatei jedes Rechners ist mit scrypt geschützt. Klartext wird nie in den Store geschrieben.
- Der Remote sieht die **Namen** der Secrets, Beschreibungen, Regeln, die öffentlichen Schlüssel der Rechner und Änderungszeiten — nie Werte oder Dateiinhalte.
- Solange der Agent entsperrt ist, kann jedes Programm, das unter deinem Benutzer läuft, über ihn entschlüsseln. Maskierung, `exec-only` und `on-demand` stoppen Versehen und ehrliche Werkzeuge, nicht ein entschlossenes Programm, das als du läuft.
- Ein entfernter Rechner behält die alten Versionen, die er schon hat.

Das vollständige Bild — was geschützt ist, wer was sieht, wogegen MLC Secrets **nicht** schützt und was dein Teil ist — steht in SECURITY.de.md.

Architektur: mehrere Rechner

Wie MLC Secrets funktioniert, wenn du es auf mehr als einem Rechner nutzt: was auf jedem Rechner bleibt, was zwischen ihnen wandert und wer unterwegs was lesen kann. Die Befehle stehen in USAGE.de.md, das Bedrohungsmodell in SECURITY.de.md.



Die zwei Hälften jedes Rechners

Jeder Rechner hat zwei Verzeichnisse. Das eine bleibt auf diesem Rechner. Das andere ist ein Git-Repository, das alle Rechner über den Remote teilen.

Verzeichnis	Enthält	Verlässt den Rechner?
`~/.config/mlcsecrets`	`identity.age` (privater Schlüssel, mit der Passphrase dieses Rechners verschlüsselt), `identity.age.pub`, `config` (z. B. `sync.auto`), `recipes.json`	nie
`~/.local/share/mlcsecrets/store`	eine `<pfad>.age`-Datei je Secret, dazu `.mlcs/recipients`, `.mlcs/policy`, `.mlcs/descriptions`	ja, als Git-Commits über den Remote
Speicher des Agenten	der entsperrte Schlüssel, Freigaben und Sperren	nie; weg, sobald der Agent sperrt

Es gibt keinen gemeinsamen Hauptschlüssel. Jeder Rechner hat sein eigenes Schlüsselpaar und seine eigene Passphrase, und eine neue Passphrase auf einem Rechner ändert nichts auf den anderen.

Der Store ist ein Git-Repository

Jede Änderung (`set`, `mv`, `rm`, `rollback`, `recipients add` …) ist sofort ein Commit im lokalen Store, mit oder ohne Netz. `mlcsecure sync` macht einen Pull und dann einen Push gegen den Remote. Der Remote kann jeder Git-Server sein, den du über ssh erreichst, etwa ein NAS. Er braucht selbst kein mlcsecrets. Mit nur einem Rechner geht auch ein Bare-Repository auf einer lokalen Platte; das gehört dann in deine Datensicherung.

- Änderungen an **verschiedenen** Secrets auf zwei Rechnern werden automatisch zusammengeführt.

- Wurde **dasselbe** Secret auf beiden geändert, hält `sync` an, nennt es und überlässt dir die Wahl (`--prefer local` oder `--prefer remote`). Der unterlegene Wert bleibt in `history`.
- `sync.auto` legt je Rechner fest, ob das von selbst passiert: `off`, `write` (nach jeder Änderung) oder `always` (zusätzlich ein Pull vor dem Lesen).

Weil die Werte vor dem Commit verschlüsselt werden, speichert der Remote nur Chiffretext. Die Struktur sieht er trotzdem: Namen der Secrets, Beschreibungen, Gruppenregeln, die öffentlichen Schlüssel der Empfänger, Änderungszeiten und Git-Autorennamen.

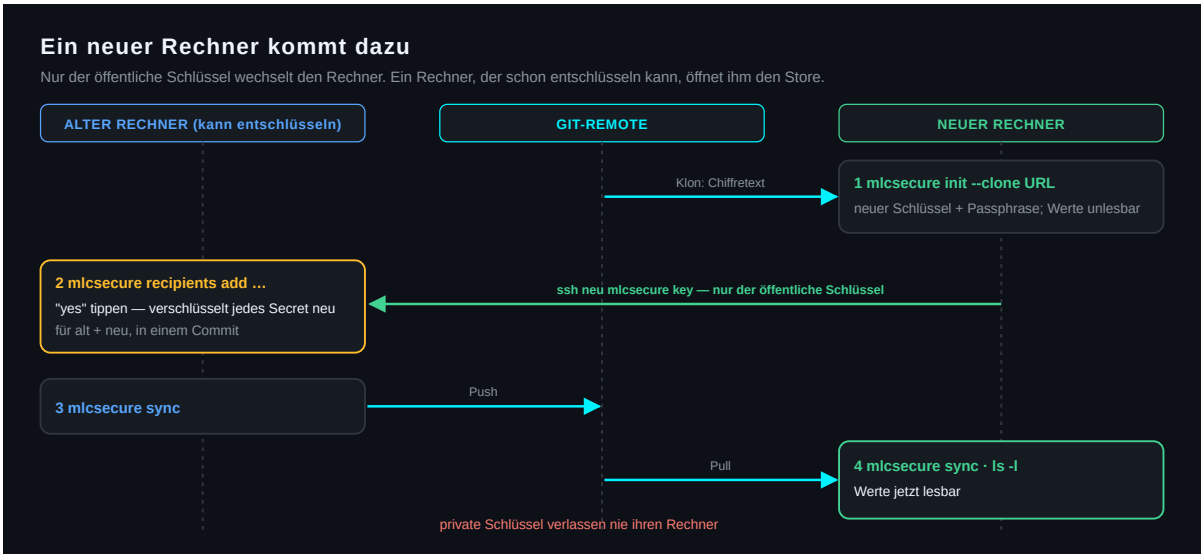
Empfänger: wer entschlüsseln kann

`.mlcs/recipients` listet den öffentlichen Schlüssel jedes Rechners, der den Store lesen darf. Jede `.age`-Datei ist mit age für **alle** gleichzeitig verschlüsselt. Jeder davon kann sie also mit seinem eigenen privaten Schlüssel entschlüsseln und braucht dafür keinen Schlüssel eines anderen Rechners.

Aktion	Wirkung auf den Store
`recipients add`	verschlüsselt jedes Secret für die neue Liste neu, in einem Commit; muss im Terminal bestätigt werden
`recipients rm`	neue Versionen lassen den entfernten Rechner aus; alte Versionen in seiner vorhandenen Historie kann er weiter lesen
`rollback`	stellt einen alten Wert wieder her und verschlüsselt ihn für die **heutigen** Empfänger

Der **Wiederherstellungsschlüssel** ist ein weiterer Empfänger, wie ein ausgeschalteter Rechner: ein age-Schlüssel, den du offline aufbewahrst (Passwortmanager oder Papier). Er ist der Weg zurück, wenn alle Rechner oder alle Passphrasen verloren sind. Siehe Wiederherstellungsschlüssel.

Ein neuer Rechner kommt dazu



1. Der neue Rechner klont den Store und erzeugt mit `mlcsecure init --clone URL` seinen eigenen Schlüssel. Er hat den Chiffretext, kann aber nichts davon lesen.
2. Ein Rechner, der schon entschlüsseln kann, holt den neuen **öffentlichen** Schlüssel (`ssh neu mlcsecure key`) und führt `recipients add` aus. Das verschlüsselt alles neu.
3. Dieser Rechner pusht den Commit mit `sync`.
4. Der neue Rechner holt ihn mit `sync`, und ab dann sind die Werte lesbar.

Zu keinem Zeitpunkt wandert ein privater Schlüssel oder eine Passphrase zwischen Rechnern. Die vollständige Checkliste mit Installation und Agenten-Skill steht unter Ein zweiter Rechner.

WSL ist ein eigener Rechner: mit eigenem Schlüssel, eigener Passphrase, eigenem Agenten und eigenem Empfängernamen (z. B. `win11-wsl`), getrennt von der Windows-Seite derselben Maschine.

Was je Rechner bleibt

Manches wird absichtlich nicht synchronisiert, weil es zu einer Maschine gehört:

Nicht synchronisiert	Warum
Rezepte (`recipes.json`)	Pfade unterscheiden sich zwischen Rechnern, und was eine KI ausführen darf, wird je Maschine entschieden
`config` (`sync.auto`)	jeder Rechner entscheidet selbst, wie oft er mit dem Remote spricht
Agent, Freigaben und Sperren	sie liegen im Speicher des Agenten und enden mit ihm
Zugriff von KI-Agenten	ein KI-Agent kann nur den Agenten und den MCP-Server des Rechners nutzen, auf dem er läuft

Ein KI-Agent auf einem Rechner kommt also nie an die Secrets eines anderen. Er erreicht sie nur über den entsperrten Agenten seines eigenen Rechners, im Rahmen der Freigaben und Rezepte dieses Rechners.

Wo Klartext existiert

Nur auf einem Rechner, der einen Empfängerschlüssel hat, und nur kurz:

- im Agenten (der Schlüssel) und im `mlcsecure`-Prozess, der gerade arbeitet;
- in dem Programm, das `exec` oder ein Rezept startet, als Umgebungsvariablen, Pipes oder temporäre Dateien, die danach gelöscht werden (tmpfs unter Linux);
- in deinem Terminal, wenn du ihn mit `get` anforderst.

Nie im Store, nie auf dem Remote, nie in der Git-Historie und nie in der Ausgabe, die ein KI-Agent sieht (Werte sind dort als `***pfad***` maskiert). Details und Grenzen: SECURITY.de.md.

Sicherheitsmodell

Was MLC Secrets schützt, gegen wen — und wo es aufhört. Lies das, bevor du ihm etwas anvertraust, das du nicht ersetzen kannst.

Was geschützt ist, und wie

Was	Schutz
Secret-Werte und Dateien	verschlüsselt mit age (X25519) für jeden Rechner, der in `.mlcs/recipients` steht; der Store enthält nur Chiffretext
Metadaten eines Secrets (Art, Dateiname, Herkunft einer gesicherten Server-Datei)	im selben verschlüsselten Umschlag
Der private Schlüssel eines Rechners	`~/.config/mlcsecrets/identity.age`, verschlüsselt mit deiner Passphrase (scrypt); nie im Store, wird nirgendwohin gesendet
Klartext während der Nutzung	wird an Programme über Umgebungsvariablen, Pipes oder Dateien im Arbeitsspeicher übergeben (tmpfs unter Linux; unter macOS und Windows ein privater Temp-Ordner pro Benutzer, der auf der Festplatte liegt), die danach gelöscht werden

Wer was sieht

Wer	Sieht	Sieht nicht
Der Git-Remote (z. B. dein NAS)	die **Namen** der Secrets, Beschreibungen, Gruppenregeln (`policy`), die Liste der öffentlichen Schlüssel der Rechner, Änderungszeiten, Git-Autorennamen	Werte, Dateiinhalte, Herkunft der Dateien
Jemand mit einer Kopie deiner Identitätsdatei, aber ohne die Passphrase	nicht mehr als der Remote	Werte
Ein entfernter Rechner (`recipients rm`)	alte Versionen, die er schon in seiner Git-Historie hat	alles, was nach seiner Entfernung geändert wurde — also ändere die Secrets, die er nicht mehr kennen darf
KI-Agenten, die `mlcsecure exec` / den MCP-Server nutzen	Namen, Beschreibungen und Programmausgaben, in denen Secret-Werte als `***path***` maskiert sind	Werte (außer ein Programm formt sie um und gibt sie aus, siehe unten)

Wogegen es schützt

- **Versehentliche Leaks**: Secrets in der Shell-History, in `.env`-Dateien neben dem Code, in Chatverläufen von KI-Assistenten, in Logs. Werte werden nie als Argumente angenommen, `get` verweigert die Ausgabe in Pipes, und die Ausgabe von `exec` und von MCP-Rezepten wird maskiert (unverändert, base64, URL- und Hex-Form).
- **Ehrliche Werkzeuge, die zu weit gehen**: `exec-only`-Gruppen werden nie angezeigt; `on-demand`-Gruppen funktionieren nur, solange du sie freigibst; befristete Sperren verweigern alles. Alles, was den Zugriff erweitert — Freigeben, eine Sperre aufheben, einen Rechner hinzufügen, eine Policy ändern, ein Rezept festlegen —, muss von einem Menschen am Terminal bestätigt werden (oder, bei MCP-Rezepten mit `--confirm`, im MCP-Client), sodass ein Skript oder KI-Agent es nicht allein tun kann.

- **KI-Clients ohne Shell** (MCP): Sie können nur Namen auflisten und die Rezepte ausführen, die du festgelegt hast; Parameter werden gegen Muster geprüft, es ist keine Shell beteiligt, die Ausgabe ist immer maskiert, jeder Lauf wird protokolliert.
- **Verlust eines Rechners**: Jeder Rechner hat seinen eigenen Schlüssel; der Store wird über Git synchronisiert. Mit einem offline aufbewahrten **Wiederherstellungsschlüssel** kannst du selbst dann wiederherstellen, wenn alle Rechner verloren sind.

Wogegen es nicht schützt

- **Ein Prozess mit vollem Zugriff auf dein Benutzerkonto, während der Agent entsperrt ist.** Ein solcher Prozess kann mit dem Agent sprechen und alles entschlüsseln, Freigaben setzen, `.mlcs/policy` oder `recipes.json` bearbeiten. Maskierung, `exec-only` und `on-demand` stoppen Versehen und gutartige Werkzeuge — nicht einen entschlossenen Angreifer, der als du läuft. Halte die Laufzeit des Agents kurz (`agent start --ttl`) und sperr ihn, wenn du fertig bist (`agent lock`).
- **Programme, die weitergeben, was sie bekommen.** Ein Rezept oder ein `exec`-Befehl ist nur so sicher wie das Programm, das er ausführt. Die Maskierung erkennt Werte, die unverändert oder in gängigen Kodierungen ausgegeben werden, nicht Werte, die umgedreht, aufgeteilt oder neu verschlüsselt wurden.
- **root auf deinem Rechner**, ein kompromittierter Kernel oder Schadsoftware, die Arbeitsspeicher oder Tastatureingaben (deine Passphrase) mitliest.
- **Kopien, die dein Editor anlegt.** `edit` hält den Klartext im Arbeitsspeicher und startet vim und neovim ohne Swap-, Undo-, Backup- und viminfo-Dateien, aber VS Code, JetBrains-IDEs, Sublime Text und andere legen eigene Kopien (Local History, Sicherungen ungespeicherter Änderungen) in deinem Profil ab, außerhalb der Reichweite von mlcsecure. `edit` nennt solche Editoren; USAGE.de.md, „Editoren und Klartextkopien“, sagt, was du ausschalten musst.
- **Werte unter 4 Zeichen** werden nicht maskiert (sie würden normale Wörter ersetzen); `exec` und der MCP-Server nennen sie.
- **Unter Windows** sind Bestätigungen am Terminal schwächer: Ein Programm, das als du läuft, kann Tastendrücke in ein Konsolenfenster einschleusen, daher ist „ein Mensch an der Tastatur“ nicht so stark nachgewiesen wie durch `/dev/tty` unter Linux und macOS. Socket und Temp-Ordner des Agents sind durch eine Zugriffsliste geschützt, die nur deinen Benutzer und SYSTEM zulässt; Identitätsdatei und Store verlassen sich auf die Berechtigungen deines Profilordners (die Identität ist ohnehin mit der Passphrase verschlüsselt).

Deine Verantwortung

1. Wähl auf jedem Rechner eine lange Passphrase; es gibt kein Zurücksetzen.
2. Leg einen Wiederherstellungsschlüssel an und bewahre ihn offline auf: `mlcsecure key generate --out recovery.txt`, füg ihn mit `mlcsecure recipients add age1… --name recovery` hinzu, `mlcsecure sync`, dann leg `recovery.txt` in einen Passwortmanager oder auf Papier ab und lösch sie von der Festplatte. Ohne ihn verlierst du alles, wenn du die Passphrasen aller Rechner verlierst.
3. Halte die Laufzeit des Agents kurz und sperr ihn, wenn eine KI ihn nicht mehr braucht.
4. Schreib nie ein Secret in eine Beschreibung — Beschreibungen sind lesbar.
5. Bearbeite Secrets in einem Editor, der keine Kopien anlegt, oder schalte seine Kopien aus (USAGE.de.md, „Editoren und Klartextkopien“).

Ein Problem melden

MLC Secrets wird von Michael Lechner gepflegt. Bitte melde Sicherheitsprobleme vertraulich statt in einem öffentlichen Issue.