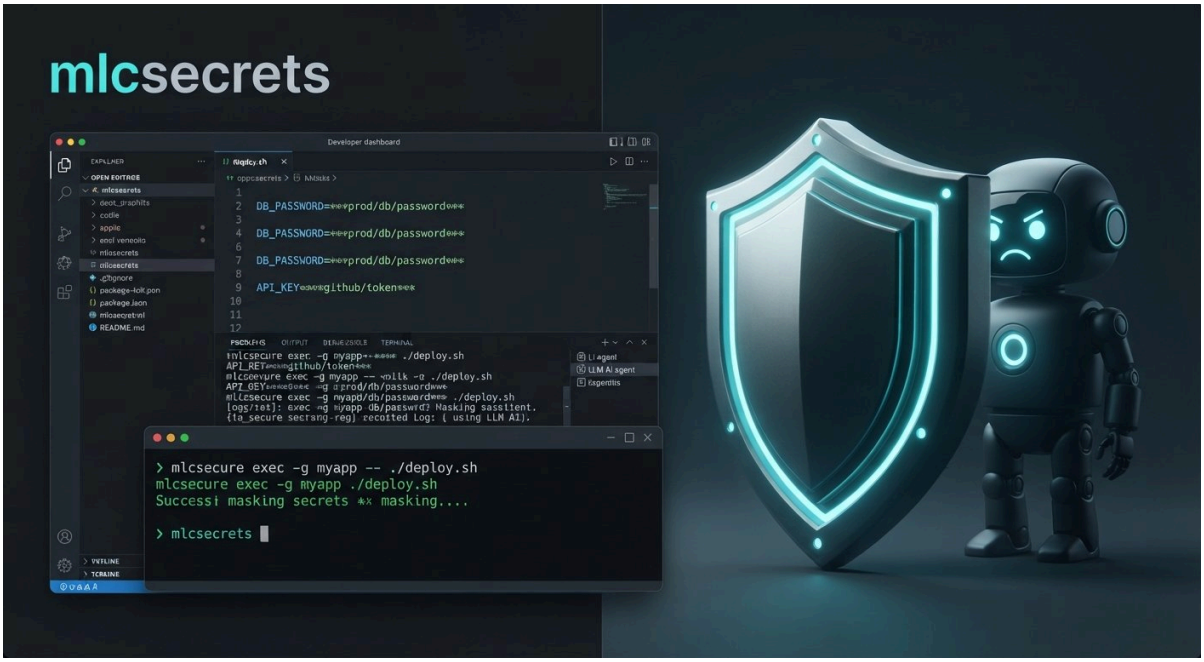
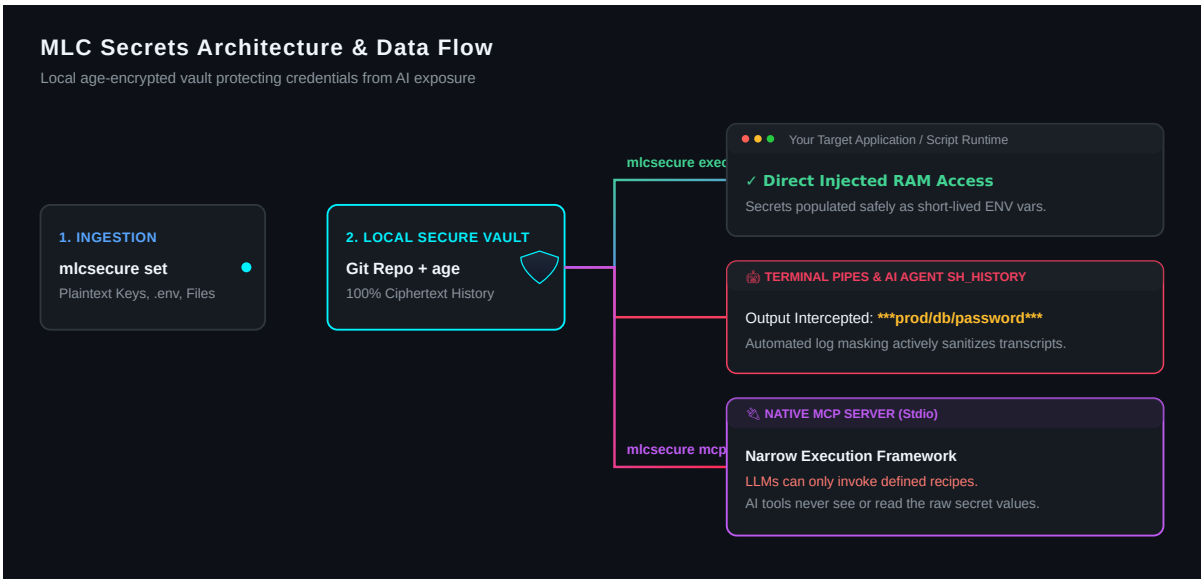
MLC Secrets



The local, AI-safe secrets vault for developers.

This handbook covers the command `mlcsecure`: everyday use, setting up further computers and AI agents, and the security model — what is protected, what is not, and your part.

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



mlcsecure — Usage

`mlcsecure` keeps your secrets — passwords, API keys, tokens and whole files — encrypted in a git repository on your machine. Every change is a commit, so nothing is ever really lost, and the repository can be pushed to a remote because it only ever contains ciphertext.

`mlcsecure` is the command of **MLC Secrets**. Its data directories are named after the product: `~/.local/share/mlcsecrets` (store) and `~/.config/mlcsecrets` (identity, configuration).

> This guide describes **version 1.9**.

Contents

- Install
- Teaching your AI agents: setup
- First start
- Naming secrets
- Storing a secret
- Storing a file
- Importing a .env file
- Descriptions
- Listing
- Renaming and editing
- Deleting
- History and rollback
- Reading a secret
- Running a command with secrets
- Config files and existing scripts
- Unlock once: the agent
- Exec-only groups
- On-demand secrets, temporary grants and blocks
- Server configuration files
- Recipes and the MCP server
- Backup and sync with a remote repository
- A second computer
- Recovery key
- Exit codes
- Security notes

Install

```
task install          # builds and copies mlcsecure to ~/.local/bin
mlcsecure --version
```

To install on another computer from one that has the source, over ssh:

```
task deploy HOST=macstudio   # detects the system, builds, copies, shows the setup plan
task deploy HOST=win10       # Windows: %LOCALAPPDATA%\Programs\mlcsecrets, added to PATH
task deploy HOST=me@win11    # another user name on that computer
```

Requirements: `git` on the PATH; Linux, macOS, or Windows 10 (1803+) / 11.

This guide and the security model are built into mlcsecure as a PDF handbook: `mlcsecure manual` opens it, `mlcsecure manual --lang de --out FILE` saves the German one.

For a Mac with Apple Silicon, build on Linux with `task build:darwin` (→ `bin/mlcsecure-darwin-arm64`) or run `task install` on the Mac itself.

For Windows, build with `task build:windows` (→ `bin/mlcsecure-windows-amd64.exe`), copy it as `mlcsecure.exe` into a folder on your PATH and use it from PowerShell or cmd. Differences from Linux and macOS:

- Paths: store in `%USERPROFILE%\.local\share\mlcsecrets\store`, identity in `%USERPROFILE%\.config\mlcsecrets\identity.age`, agent socket in `%LOCALAPPDATA%\mlcsecrets\agent.sock`.
- `edit` uses `%EDITOR%`, else Notepad (`$env:EDITOR = "code --wait"` for VS Code).
- `ssh-add` needs the Windows *OpenSSH Authentication Agent* service running.
- `backup-file`, `restore` and `check` work only with `--host` (a Linux server); local system files are not supported on Windows.

WSL is a separate Linux computer to mlcsecure, even on the same machine. The Windows `mlcsecure.exe` uses the Windows key and agent, WSL cannot reach the Windows agent socket, and agents running in WSL read their skill from WSL's home. So give WSL the Linux binary (`task install` inside WSL, or copy `bin/mlcsecure` to `~/.local/bin`), and follow Checklist: a new computer there, with its own passphrase, its own ssh key for the remote and its own recipient name such as `win11-wsl`. `task deploy` reaches only the Windows side. Over ssh, call it through a login shell, otherwise `~/.local/bin` is not on the PATH: `ssh me@win11 'wsl -e bash -lc "mlcsecure key"'`.

Teaching your AI agents: setup

Coding agents learn how to use mlcsecure — and what never to do with it — from a *skill*, a short instruction file they load on their own. `mlcsecure setup` puts it where each agent on this computer looks:

```
mlcsecure setup              # show what would be done
mlcsecure setup --yes        # do it
mlcsecure setup --check      # exit 5 if something is missing or outdated
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	reads `~/.claude/skills`	the entry is shown; add it to `opencode.json` yourself

Agents are found by their configuration folder; `--agent claude,gemini` picks them yourself. The skill is built into the binary, so after an update run `mlcsecure setup --yes` again. `mlcsecure setup --print-skill` prints it.

Agents with a shell need only the skill: they call `mlcsecure exec`. `--mcp` additionally registers the MCP server for tools without a shell. Only the `mlcsecure` entry of a config file is changed; the previous file is kept as `FILE.bak`, and a config file that is a symlink (e.g. into a dotfiles repository) is never written — setup shows the entry instead.

Whether an agent follows the skill depends on its model. In test sessions Claude Code and OpenCode with a hosted model loaded it by themselves and never touched a value. A small local model (Qwen 27B via Ollama) ignored the skill, even when the request said "use mlcsecure", and went for `.env` directly; OpenCode blocked that read on its own. For work that touches secrets, use a model that follows skills.

What a session with the skill looks like — the agent finds a `.env` file, moves it into the vault, asks you to unlock, and runs the tests; it only ever sees names:



First start

```
mlcsecure init
```

You are asked for a **passphrase** twice. It protects your private key — the one thing that can decrypt your secrets. Choose a long one and keep it safe: **without it, your secrets cannot be recovered.**

`init` creates two things:

What	Default location	Note
Your identity (private key, encrypted with the passphrase)	`~/.config/mlcsecrets/identity.age`	never commit or share this file
The store (git repository with the encrypted secrets)	`~/.local/share/mlcsecrets/store`	safe to push to a remote

To use other locations, set `MLCSECURE_IDENTITY` and/or `MLCSECURE_STORE`.

If an identity already exists (for example after setting up a second store), `init` asks for its passphrase and reuses it.

Naming secrets

Secrets are addressed by **paths**; the parts before the last `/` are groups:

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

Each part uses lowercase letters, digits, `.`, `_` and `-`, and must not start with a dot. Paths are readable in the store (so `git log` makes sense); the values and any metadata are encrypted.

Storing a secret

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

You type the value twice; nothing is shown on screen. You can also pipe it in — one trailing newline is removed:

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

Values are never passed as arguments (`mlcsecure set name VALUE` does not exist), because arguments end up in your shell history and in `ps`.

Storing does **not** need your passphrase — only reading does. Setting an existing path overwrites it; the old value stays in the history.

Storing a file

Certificates, kubeconfigs, SSH keys — any file up to 10 MiB:

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

The original file name (`server.pem`) is stored too, encrypted. You can delete the plaintext file afterwards.

Importing a .env file

```
mlcsecure import-env myapp .env --dry-run   # show what would happen
mlcsecure import-env myapp .env
```

Every `KEY=VALUE` becomes `myapp/<key in lower case>` — `OPENAI_API_KEY` turns into `myapp/openai_api_key` — all in one commit. `export`, comments and quoted values are understood. Existing secrets with the same path are overwritten (the old values stay in the history). The `.env` file itself is not touched: **delete it yourself** once the import looks right.

Descriptions

With many projects, names alone get cryptic. Secrets and whole groups can have a short description:

```
mlcsecure describe mlcauth "Project mlcauth (auth server)"     # a group
mlcsecure describe mlcauth/db_password Postgres, user app       # a secret
echo "$TOKEN" | mlcsecure set ci/token --desc "GitHub token for CI"
mlcsecure import-env myapp .env --desc "Project myapp"          # describes the group
mlcsecure describe mlcauth                                       # show
mlcsecure describe mlcauth --clear                               # remove
mlcsecure ls -d                                                  # list with descriptions
```

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

Descriptions are stored **readable**, like the names (`.mlcs/descriptions` in the store): `ls -d` needs no passphrase, and AI agents can use them to pick the right secret. Anyone with the store — including the remote — can read them, so **never put a secret into a description**. `mv` takes a secret's description along, `rm` removes it. At most 200 characters, one line.

Listing

```
mlcsecure ls            # everything
mlcsecure ls prod       # one group
mlcsecure ls -d         # with descriptions (no passphrase)
mlcsecure ls -l         # with kind, policy, file name, description (asks for the passphrase)
mlcsecure show myapp/config   # one secret in detail: size, last change, keys in a file
```

Values are never shown. Kind and file name are encrypted too, which is why `-l` and `show` need the passphrase (or the unlocked agent).

For a stored `.env`, INI, TOML or JSON file, `show` lists the names of its keys — handy when the file is long or you forgot what is in it:

```
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 and database
keys         12
  title
  server.port
  db.url
  db.password
  …
```

Only names that look like keys are listed; a line `show` does not understand is counted, not shown — it could be a value. For a single value, `show` leaves out the size: the length of a password is a hint to it.

Renaming and editing

```
mlcsecure mv prod/db/pass prod/db/password   # target must not exist
mlcsecure edit prod/db/password
```

`edit` decrypts the secret into a private file in memory (tmpfs, never on disk), opens it in `$VISUAL`, `$EDITOR` or `vi`, and saves it only if you changed something. For values, the newline your editor adds at the end is dropped. The temporary file is deleted afterwards.

Where plaintext goes for a moment (`edit`, `exec --file`):

System	Location	Note
Linux	`$XDG_RUNTIME_DIR` or `/dev/shm`	memory only (tmpfs); never on disk
macOS	your private temp folder (`$TMPDIR`, mode 0700)	macOS has no tmpfs: this is on disk — encrypted if FileVault is on — and deleted right after use
Windows	`%TEMP%\mlcsecrets`, only your user and SYSTEM may open it	on disk — encrypted if BitLocker is on — and deleted right after use
others	—	`edit` and `exec --file` refuse

Editors and plaintext copies

`edit` keeps its copy in memory, but some editors keep copies of their own, in your profile on disk, where mlcsecure cannot delete them:

Editor	What it keeps	What to do
vim, neovim	swap, undo and backup files; viminfo/shada keeps text you yanked	nothing: `edit` starts them with `-n -i NONE` and without undo and backup files
nano	nothing, unless you turned on `set backup`	—
VS Code, VSCodium, Cursor	Local History keeps every saved version; unsaved changes are backed up	add `"workbench.localHistory.exclude": {"**/mlcsecrets/s-*/**": true}` to your user settings, and save before you close the tab
JetBrains IDEs	Local History keeps every saved version	use a terminal editor for secrets
Sublime Text, Notepad++	the session keeps unsaved changes	save before closing, or turn off hot exit / session snapshots
Notepad (Windows 11)	remembers unsaved tabs	save before closing

`edit` prints a note when it starts one of the editors that keep copies. For files that matter, a terminal editor is the safe choice: `EDITOR=vim mlcsecure edit myapp/config`.

Deleting

```
mlcsecure rm openai/api_key
```

The secret disappears from the store but remains in the git history.

History and rollback

Every change is a git commit, so every earlier value is still there:

```
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` shows dates and messages, never values, and needs no passphrase. `rollback` brings the old value back as a new commit — nothing is rewritten — and encrypts it for today's set of machines, so a computer added in the meantime can read it. It also restores deleted secrets: their deletion shows up in `history`, pick the commit before it. A secret renamed with `mv` starts a new history; earlier versions are under its old name.

Reading a secret

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

Asks for the passphrase and prints the value. Stored files are not printed; use `--out`. To keep secrets out of logs and out of AI assistants' transcripts, `get` **refuses to print into a pipe or a file redirect**. If you need the value in a file:

```
mlcsecure get certs/server --out ./server.pem     # created with mode 0600
```

`--out` never overwrites an existing file.

Running a command with secrets

This is the main way to use secrets in scripts: the command gets them, your terminal and logs don't.

```
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	Effect
`-g, --group GROUP`	every secret below `GROUP` becomes a variable named after the last path part in upper case: `prod/db/password` → `PASSWORD`, `myapp/openai_api_key` → `OPENAI_API_KEY`
`--prefix P`	put `P` in front of names from groups (`--prefix DB_` → `DB_PASSWORD`)
`--env-file PATH`	`PATH` is a stored `.env` file; every `KEY=VALUE` line becomes a variable
`-e, --env NAME=PATH`	`NAME` gets the value of `PATH`
`-f, --file NAME=PATH`	the content of `PATH` is written to a private file in memory (tmpfs); `NAME` gets its path

All options can be repeated. Stored files found through `--group` are always handed over as files. If two groups would set the same name, `exec` stops and asks you to use `--env` or `--prefix`. A `.env` file replaces a name that came from a group, and an explicit `--env`/`--file` replaces both. Existing environment variables of the same name are replaced.

In the command's arguments, `$NAME` or `${NAME}` of a `--file` variable is replaced by the path of its temporary file — write it in single quotes (`'$CONFIG'`) so that your shell leaves it for mlcsecure. This works the same in PowerShell; in cmd,

write `$CONFIG` without quotes. Only file paths are put into arguments, never values: arguments are visible to every program on the computer (`ps`).

Everything after `--` is the command, including its own options. The temporary files are deleted when the command ends — also when you press Ctrl-C or the process is terminated; those signals are passed on to the command. `exec` exits with the command's exit status (128+n if it was killed by signal n).

`exec` asks for your passphrase — unless the agent is unlocked (next section).

What you see: masking

- **Output to your terminal is shown as it is** — you see exactly what the command prints, colours and progress bars included.
- **Output to a pipe or a file** — a log, `| tee`, an AI assistant's tool — has every secret value replaced by `***path***`, for example `token=***prod/api/token***`. Also caught: the value in base64 (all common variants), URL-encoded and hex; for stored files, the whole content and each line of 16+ characters.
- stdout and stderr are judged separately: `exec … 2>err.log` shows stdout on screen and masks the log.
- Values shorter than 4 characters are not masked (they would replace ordinary words everywhere); `exec` names them on stderr.
- `--mask` or `MLCSECURE_MASK=always` masks your terminal too — useful for screen sharing, or for AI tools that run commands in a pseudo-terminal.

There is deliberately no switch to turn masking off for a pipe. If you really need the plain output in a file, redirect *inside* the command: `mlcsecure exec -g prod/db -- sh -c './dump.sh > out.txt'`.

> Masking catches secrets that are printed as they are or in a common encoding. A program that transforms them in other ways (reverses, splits, encrypts them) gets past it. It protects against accidents, not against a program that sets out to leak.

Config files and existing scripts

A program's settings often live in one big file: `.env`, `config.toml`, `app.ini`, `settings.json`. Keep that file whole in the store: you edit it in your editor as before, it is stored encrypted, and the program gets it only while it runs.

Securing a config file

```
mlcsecure add-file myapp/config ./config.toml --desc "myapp: server and database"
rm ./config.toml                    # the plaintext is no longer needed
mlcsecure show myapp/config         # what is inside, without values
mlcsecure edit myapp/config         # change it in your editor
mlcsecure history myapp/config      # every change; rollback if needed
```

The same works for `.env`, `.ini`, `.json`, `.yaml` or any other file up to 10 MiB. `edit` opens the file under its original name, so your editor highlights it as TOML, INI and so on. Read Editors and plaintext copies once.

Handing it to a program

The program reads …	Command
a path given as an option (`--config FILE`)	`mlcsecure exec -f CONFIG=myapp/config -- myapp --config '$CONFIG'`
a path from a variable (`KUBECONFIG`, `GOOGLE_APPLICATION_CREDENTIALS`)	`mlcsecure exec -f KUBECONFIG=certs/kube -- kubectl get pods`
environment variables (what a `.env` file is for)	`mlcsecure exec --env-file myapp/env -- npm start`
a fixed path such as `./config.toml`	see A program with a fixed config path

`-f` writes the file to memory (tmpfs) under its original name and deletes it when the program ends. `--env-file` and `-f` can be combined, and repeated.

.env files: whole or split

There are two ways to keep a `.env` file. Both give the program the same variables.

	Whole file	One secret per line
store it	`mlcsecure add-file myapp/env .env`	`mlcsecure import-env myapp .env`
use it	`exec --env-file myapp/env -- …`	`exec --group myapp -- …`
edit	`edit myapp/env` — the whole file	`edit myapp/openai_api_key` — one value
policies, grants, history	for the whole file	for each value
good for	many settings that belong together	a few keys you also use elsewhere

Adapting an existing script

Usually the script stays as it is: you start it through `mlcsecure exec`, and the script and everything it starts see the variables. Only the line that reads the plaintext file goes.

A shell script that loads `.env`:

```
# before: deploy.sh starts with  set -a; . ./.env; set +a
./deploy.sh
# after: delete that line (or make it optional:  [ -f .env ] && . ./.env)
mlcsecure exec --env-file myapp/env -- ./deploy.sh
```

Python with `python-dotenv`, Node with `dotenv`: `load_dotenv()` and `require('dotenv').config()` do nothing when there is no `.env` file, and they never replace variables that are already set. The code stays; only the start changes:

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

A program that takes its config file as an option:

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

In a Taskfile, put `mlcsecure exec` in front of the command:

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

In a Makefile, `$` must be doubled: `-- myapp serve --config '$$CONFIG'`.

docker compose: instead of `env_file: .env`, list the variable names under `environment:`; compose takes their values from its own environment:

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

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

Note that Docker keeps a container's environment in its configuration (`docker inspect` shows it) — that is outside mlcsecure.

A program with a fixed config path

Many programs also accept the path in a variable or an option — check that first. If a program only ever reads `./config.toml`, mount the file into a container, or link it for the time the program runs (Linux and 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'
```

The link points into memory and is removed afterwards; add `config.toml` to `.gitignore` all the same.

Unlock once: the agent

Typing the passphrase for every `exec` gets old, and AI assistants or scripts without a terminal cannot type it at all. Unlock once instead:

```
mlcsecure agent start            # asks for the passphrase, runs in the background
mlcsecure agent start --ttl 2h   # lock automatically after 2 hours (default 8h)
mlcsecure agent status           # "unlocked (pid …), locks in 7h59m at 18:30"
mlcsecure agent lock             # lock now (also: agent stop)
```

While the agent runs, `exec`, `get`, `edit` and `ls -l` no longer ask for the passphrase. The agent keeps your key in memory only; the key never leaves it — other `mlcsecure` processes send it encrypted data and get the plaintext back over a socket only your user can open (`$XDG_RUNTIME_DIR/mlcsecrets/agent.sock`; on Windows `%LOCALAPPDATA%\mlcsecrets\agent.sock`). When the time is up or you lock it, the agent forgets the key and exits. `agent status` exits with 3 when locked, so scripts can check it.

> **While the agent is unlocked, any program running as your user can decrypt your secrets through `mlcsecure`** — that is what makes it useful for scripts and AI assistants. Keep the TTL short and lock it when you're done.

If `agent start` reports that the socket path is too long (Unix sockets allow about 100 characters), set `MLCSECURE_AGENT_SOCK` to a shorter path.

To run it under systemd or in a terminal of its own: `mlcsecure agent start --foreground`. Another socket path can be set with `MLCSECURE_AGENT_SOCK`.

Exec-only groups

Some secrets should be usable but never shown — not to you by accident during a screen share, and not to an AI assistant:

```
mlcsecure policy set prod exec-only    # asks you to type "prod" to confirm
mlcsecure policy ls
mlcsecure policy set prod none         # lift it again
```

For every secret below `prod`, `get`, `get --out` and `edit` refuse (exit code 2), while `exec` and `ls -l` keep working. You can still replace a value with `set`. The confirmation must be typed on the terminal, so scripts and AI assistants cannot change a policy. Policies are stored in the store (`.mlcs/policy`) and versioned with it.

> A program with full access to your account could still edit that file or use the agent directly. `exec-only` keeps honest tools and accidents from revealing values; it is not a sandbox.

On-demand secrets, temporary grants and blocks

Some secrets should normally stay out of reach — even for `exec` — and be allowed only for a moment, for example a private SSH key an AI assistant needs for one deployment.

```
mlcsecure policy set ssh on-demand          # once; type "ssh" to confirm
mlcsecure grant ssh/deploy_key --for 15m    # when needed; type "yes" to confirm
mlcsecure grants                            # what is granted or blocked right now
mlcsecure revoke ssh/deploy_key             # end it early
```

An **on-demand** secret is refused by `exec`, `ssh-add`, `get` and `edit` unless a grant for it (or its group) is running. If you unlocked with your passphrase at the terminal, you are present and on-demand does not apply to you; it applies to everything that runs through the agent — scripts and AI assistants.

The opposite works for any secret or group:

```
mlcsecure block prod --for 2h      # refuse prod/* to everyone, also to you
mlcsecure unblock prod             # type "prod" to confirm
```

Grants and blocks live **only in the memory of the agent** on this computer: they are not synced, and they end when their time is up or the agent locks (`agent lock`, or its `--ttl`). They therefore need a running agent. Granting and lifting a block must be confirmed on the terminal; revoking and blocking — which only make things stricter — need no confirmation.

SSH keys without key files

```
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                  # works for 10 minutes, then ssh-agent forgets the key
```

`ssh-add` pipes the key straight into your running `ssh-agent` (`ssh-add -t`), so it never becomes a file. The same rules as for `exec` apply: on-demand needs a grant, blocks refuse. The stored key must not have a passphrase of its own — MLC Secrets already encrypts it.

Server configuration files

The secret parts of a server's configuration — WireGuard and VPN configs, TLS private keys, `.htpasswd`, `.env` files with passwords — usually belong to root. mlcsecure stores them together with their location, mode and owner, and puts them back exactly like that:

```
# from this computer (sudo is used only if you cannot read the file)
mlcsecure backup-file local/nginx-key /etc/nginx/ssl/key.pem

# from a server, through ssh and "sudo -n"
mlcsecure backup-file vps/wg0 /etc/wireguard/wg0.conf --host vps --desc "WireGuard on vps"

mlcsecure check vps/wg0      # identical? — compares checksum, mode and owner, never shows content
mlcsecure restore vps/wg0    # shows what it will do, asks for "yes", writes atomically
```

- The content travels through pipes only; it is never written to a disk on the way. `restore` writes a temporary file next to the target, sets owner and mode, and renames it over the original — a failed restore leaves the old file untouched.
- `restore` does nothing if the file is already identical. `--to PATH` and `--host H` restore somewhere else; `--yes` skips the question (scripts, recipes).
- `check` exits with **5** if the file differs or is missing — handy in a cron job or a recipe to notice that someone changed a server.
- On a server, sudo must work without a password for this (`sudo -n`), since ssh is run without a terminal. Otherwise log in as a user that may read the file and add `--no-sudo` (e.g. `--host root@vps --no-sudo`).
- A file you have as a stream rather than a path: `ssh vps sudo cat /etc/x.key | mlcsecure add-file vps/x - --name x.key` (stored without location; restore it with `exec -f`).
- Meant for the secret parts. For the rest of `/etc`, a configuration management tool or etckeeper is the better place.

Do not run mlcsecure itself with sudo. As root it would use root's identity and store, not yours; it warns about that (silence with `MLCSECURE_ALLOW_ROOT=1` if you really mean it). For root-owned files it calls `sudo` by itself.

Recipes and the MCP server

AI tools that cannot run shell commands (Claude Desktop, chat clients, your own agents) can use your secrets through the **MCP server**. It is deliberately narrow: an agent can list names and descriptions, see whether the vault is unlocked, and run **recipes** — commands *you* defined. It can never read a value, run anything else, or change secrets, rules, grants or recipes.

Defining recipes

```
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}
```

You are shown the complete recipe and must type its name to confirm. Rules:

- The program and its fixed arguments are yours; an agent can only fill in `{param}` placeholders, and each must be a **whole argument** (no shell is involved, so `;`, `$(…)` and quotes have no special meaning).
- Every value must match its pattern completely (`beta|stable` allows exactly those two words), may not be longer than 256 characters, contain line breaks, or start with `-` (unless you allow that with `--allow-dash NAME`). `--optional NAME` lets a parameter be left out.
- Secrets come in as with `exec`: `-g` groups, `-e` values, `-f` files. On-demand rules and blocks apply as usual.
- `--confirm` makes the server ask **you** before every run — the MCP client shows the question with the exact command line; the AI cannot answer it.
- Default time limit 10 minutes (`--timeout`, at most 24h); on timeout the whole process group is killed.

```
mlcsecure recipe ls                # name, parameters, description
mlcsecure recipe show deploy-barcode
mlcsecure recipe rm deploy-barcode
```

Recipes belong to one computer (`~/.config/mlcsecrets/recipes.json`): paths differ between Linux and Mac, and what an agent may run is decided per machine. They are not synced.

Connecting a client

`mlcsecure setup --mcp --yes` registers the server with the agents on this computer (see setup). By hand:

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

The client starts the server itself as `mlcsecure mcp`; it runs over stdio only — no network port. It decrypts through the unlock agent and never asks for a passphrase: start the agent (`mlcsecure agent start`) before you let an AI work.

Tool	What it does
`status`	is the agent unlocked, until when; active grants and blocks
`list_secrets`	names, descriptions and rules (optionally below a group) — never values
`show_secret`	one secret like `mlcsecure show`: kind, rule, last change and the key names in a stored `.env`, INI, TOML or JSON file — never values; needs the agent
`list_recipes`	your recipes with their parameters and the secrets they receive
`run_recipe`	runs a recipe; returns exit code and output, secrets **always** masked, at most 64 KiB per stream

A prompt `use_secrets` guides a model through a task. The full reference with every parameter is generated from the running server: docs/mcp-tools.md.

Every run — and every refusal — is written to `~/.config/mlcsecrets/mcp-audit.jsonl` (time, recipe, arguments, outcome, exit code, duration; never values).

> The MCP server protects the **MCP path**. An AI that also has a shell on your account can edit the recipe file or call `mlcsecure` directly — for such tools the rules in SKILLS.md and the other protections (masking, exec-only, on-demand) apply.

Backup and sync with a remote repository

The store only ever contains ciphertext, so it can live on a server you control as a plain git repository:

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

Working on one computer only? The remote can just as well be a bare repository on a local disk, for example a second drive:

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

It is then your only copy besides the store itself, so **include it in your regular backup** — a remote on the same disk as the store protects against mistakes, not against a broken disk.

Changes are committed locally right away; `sync` exchanges them with the remote. Run it after changing secrets and before using them on another computer. Changes to different secrets on two computers are combined automatically. If the **same** secret was changed on both, `sync` changes nothing, names the secret and stops:

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

Decide which value wins with `mlcsecure sync --prefer local` or `--prefer remote`. The other value is not lost: it stays in `history`.

Automatic sync

Instead of running `sync` yourself, let each computer do it:

```
mlcsecure config set sync.auto write    # sync after every change
mlcsecure config set sync.auto always   # …and also pull before reading
mlcsecure config                        # show this computer's settings
```

`sync.auto`	Effect
`off` (default)	only `mlcsecure sync` talks to the remote
`write`	after every successful change (`set`, `add-file`, `import-env`, `mv`, `edit`, `rm`, `rollback`, `describe`, `policy set`, `recipients add/rm`)
`always`	as `write`, plus one pull before `get`, `ls`, `exec`, `history` and before every change — the other computer's latest values are always there, at the cost of one round trip to the remote per command

Automatic sync never makes a command fail: if the remote is unreachable or there is a conflict, the change stays committed locally and a warning on stderr tells you to run `mlcsecure sync` later. Output on stdout is not affected, so scripts that read `ls` keep working. Without a remote nothing happens.

Settings belong to one computer and live next to its identity (`~/.config/mlcsecrets/config`, plain `key = value` lines).

What the remote can see: the names of your secrets, when they changed, and the author names from your git configuration. Not the values.

A second computer

Checklist: a new computer

The whole round, with the details below. *Old* is a computer that can already decrypt, *new* the one being added. How the pieces fit together — with diagrams — is in ARCHITECTURE.md.

- ☐ Install: on *old* (with the source) `task deploy HOST=new` — or `task install` on *new*
- ☐ On *new*: `mlcsecure setup --yes` — the skill for its AI agents
- ☐ On *new*, in a terminal: `mlcsecure init --clone URL` — choose this computer's passphrase
- ☐ On *old*: `mlcsecure recipients add "$(ssh new mlcsecure key)" --name new`, type `yes`, then `mlcsecure sync`
- ☐ On *new*: `mlcsecure sync`, then `mlcsecure ls -l` — values are readable

- [ ] On *new*: `mlcsecure config set sync.auto write` if you want automatic sync
- [ ] On *new*: `mlcsecure agent start` when an AI should use secrets

Step by step

Each computer has its own identity (key + passphrase); the private key never leaves it. A second computer joins by cloning the store and being added as a *recipient* on a computer that can already decrypt:

```
# 1. on the first computer (already set up, with a remote):
mlcsecure sync

# 2. on the new computer:
mlcsecure init --clone mlc@nas.local:/volume1/homes/mlc/repositories/secrets-store.git
#    → asks for a passphrase for this computer and prints its public key

# 3. back on the first computer — fetches the key over ssh, no copying:
mlcsecure recipients add "$(ssh macstudio mlcsecure key)" --name macstudio   # type "yes"
mlcsecure sync

# 4. on the new computer:
mlcsecure sync
mlcsecure ls -l                                  # works now
```

`recipients add` re-encrypts **every** secret so the new key can read it, in a single commit. Because that hands out access to everything, it must be confirmed on the terminal. Older versions in the history stay encrypted for the computers of their time; `rollback` re-encrypts for today's set.

`mlcsecure key` prints a computer's public key at any time; it is not secret and needs no passphrase. (Set up before 0.11? Then it asks for the passphrase once and remembers the key in `identity.age.pub`.) If `ssh` cannot find `mlcsecure` on the other computer, use its full path, e.g. `ssh macstudio .local/bin/mlcsecure key`.

```
mlcsecure recipients                 # list: name and key
mlcsecure recipients rm macstudio    # stop encrypting for it (confirm by typing the name)
```

After `recipients rm`, new versions are unreadable for that computer, but it can still read **old versions** in the git history it already has. Change the secrets it must no longer know. The last recipient cannot be removed.

Recovery key

Each computer's key is protected by its passphrase, and there is no reset. If every computer is lost — or every passphrase forgotten — the store cannot be decrypted any more. A **recovery key** kept offline prevents that:

```
mlcsecure key generate --out recovery.txt         # prints the public key age1…
mlcsecure recipients add age1… --name recovery    # type "yes"; re-encrypts everything
mlcsecure sync
```

Now put `recovery.txt` into your password manager (or print it) and delete it from the disk. It is a normal age key: even without MLC Secrets, `age -d -i recovery.txt <file>.age` decrypts a stored file.

To recover on a new computer:

```
mlcsecure key import recovery.txt                 # becomes this computer's identity (new passphrase)
mlcsecure init --clone mlc@nas.local:/volume1/homes/mlc/repositories/secrets-store.git
```

The recovery key can read everything, like any computer. Treat it like the master key it is — and consider `recipients rm recovery` plus a new one if it might have been exposed.

Exit codes

Code	Meaning
0	success
1	error

Code	Meaning
2	wrong usage (unknown command or flag, wrong number of arguments, bad path or revision, no terminal, refused output, target exists, too large, exec-only, on-demand without grant, blocked)
3	locked — wrong passphrase, no identity, agent not running (`agent status`), or no agent and no terminal to ask for the passphrase (scripts, AI agents)
4	secret not found
5	`check`: the live file differs from the backup or is missing

Scripts can rely on these. `exec` instead returns the exit status of the command it ran once the command has started.

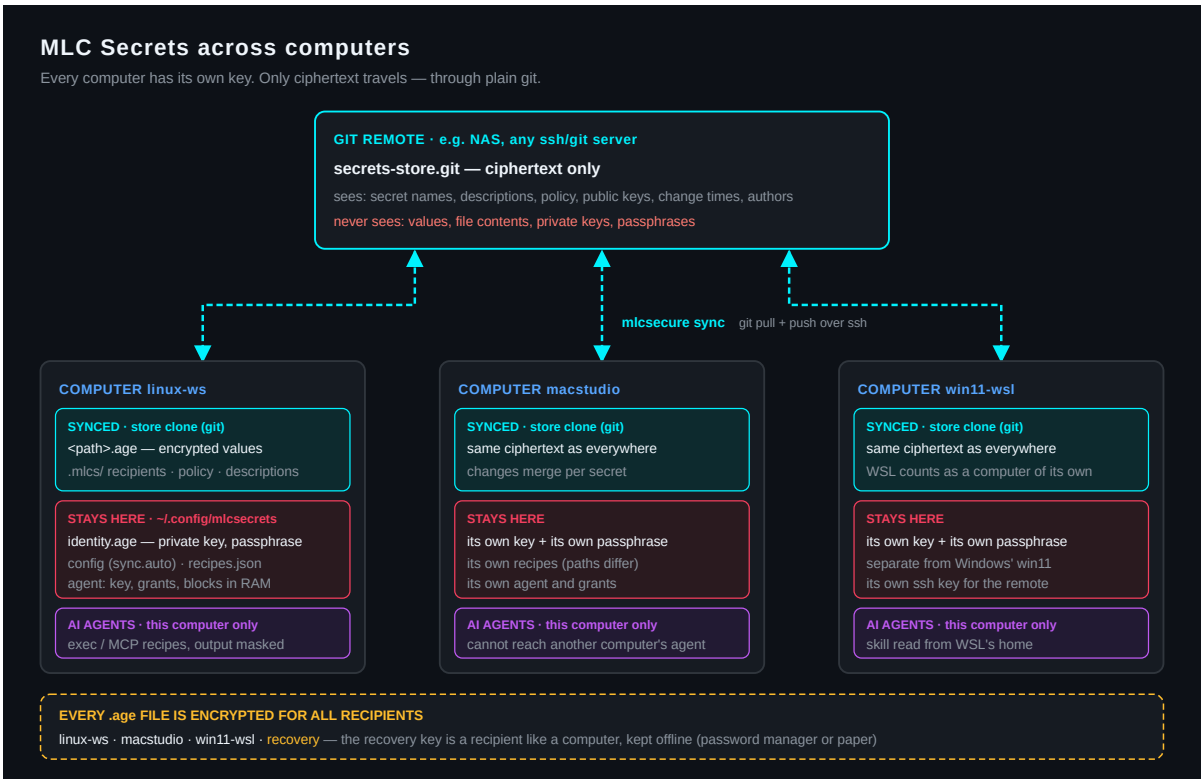
Security notes

- Encryption is age (X25519); each computer's identity file is protected with scrypt. Plaintext is never written into the store.
- The remote sees secret **names**, descriptions, rules, the computers' public keys and change times — never values or file contents.
- While the agent is unlocked, any program running as your user can decrypt through it. Masking, `exec-only` and `on-demand` stop accidents and honest tools, not a determined program running as you.
- A removed computer keeps the old versions it already has.

The full picture — what is protected, who sees what, what MLC Secrets does **not** defend against, and your part — is in SECURITY.md.

Architecture: several computers

How MLC Secrets works when you use it on more than one computer: what stays on each computer, what travels between them, and who can read what along the way. The commands are in USAGE.md; the threat model is in SECURITY.md.



The two halves of every computer

Each computer has two directories. One stays on that computer. The other is a git repository that every computer shares through the remote.

Directory	Holds	Leaves the computer?
`~/.config/mlcsecrets`	`identity.age` (private key, encrypted with this computer's passphrase), `identity.age.pub`, `config` (e.g. `sync.auto`), `recipes.json`	never
`~/.local/share/mlcsecrets/store`	one `<path>.age` file per secret, plus `.mlcs/recipients`, `.mlcs/policy`, `.mlcs/descriptions`	yes, as git commits through the remote
memory of the agent	the unlocked key, grants and blocks	never; gone when the agent locks

There is no shared master key. Each computer has its own key pair and its own passphrase, and changing the passphrase on one computer changes nothing on the others.

The store is a git repository

Every change (`set`, `mv`, `rm`, `rollback`, `recipients add` …) is a commit in the local store at once, with or without a network. `mlcsecure sync` does a pull and then a push against the remote. The remote can be any git server you reach over ssh, such as a NAS. It needs no MLC Secrets of its own. With a single computer, a bare repository on a local disk works too; then it belongs in your backup.

- Changes to **different** secrets on two computers are merged automatically.

- If the **same** secret changed on both, `sync` stops, names it, and leaves the choice to you (`--prefer local` or `--prefer remote`). The losing value stays in `history`.
- `sync.auto` decides per computer whether this happens by itself: `off`, `write` (after every change) or `always` (also a pull before reading).

Because the values are encrypted before they are committed, the remote only ever stores ciphertext. It does see the structure: secret names, descriptions, group rules, the recipients' public keys, change times and git author names.

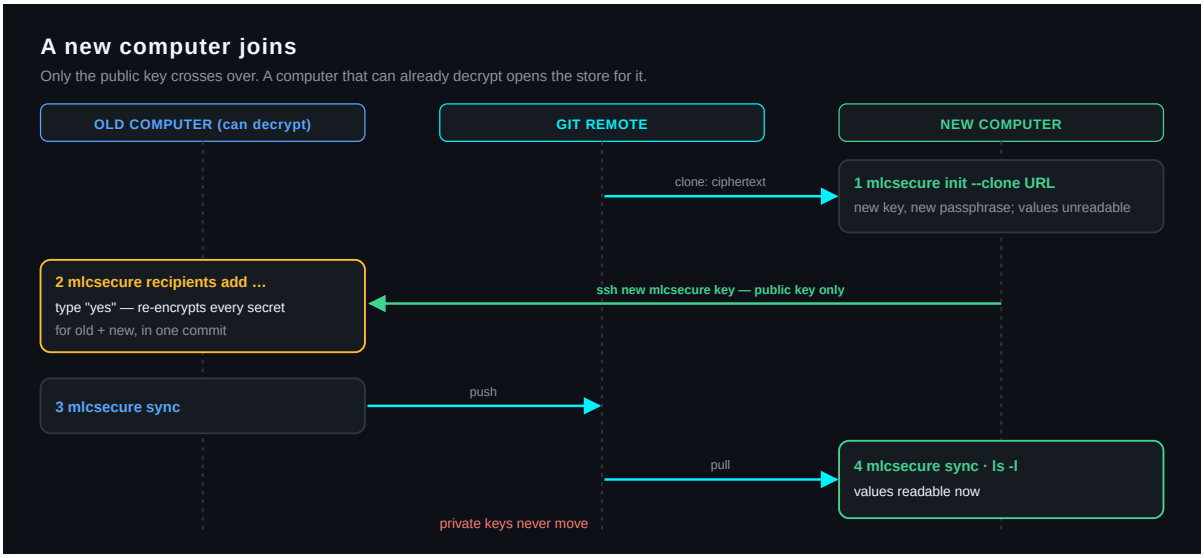
Recipients: who can decrypt

`.mlcs/recipients` lists the public key of every computer that may read the store. Each `.age` file is encrypted with age for **all** of them at once, so any one of them can decrypt it with its own private key, and none needs another computer's key.

Action	Effect on the store
`recipients add`	re-encrypts every secret for the new list, in one commit; must be confirmed on the terminal
`recipients rm`	new versions leave the removed computer out; it can still read old versions in the history it already has
`rollback`	restores an old value and encrypts it for **today's** recipients

The **recovery key** is one more recipient, like a computer that is switched off: an age key you keep offline (password manager or paper). It is the way back in when every computer or every passphrase is lost. See Recovery key.

A new computer joins



1. The new computer clones the store and creates its own key with `mlcsecure init --clone URL`. It has the ciphertext but cannot read any of it.
2. A computer that can already decrypt fetches the new **public** key (`ssh new mlcsecure key`) and runs `recipients add`, which re-encrypts everything.
3. That computer pushes the commit with `sync`.
4. The new computer pulls it with `sync`, and the values are readable from then on.

No private key and no passphrase moves between computers at any point. The complete checklist, including installing and the agent skill, is in A second computer.

WSL is a computer of its own: it has its own key, passphrase, agent and recipient name (e.g. `win11-wsl`), separate from the Windows side of the same machine.

What stays per computer

Some things are deliberately not synced, because they belong to one machine:

Not synced	Why
Recipes (`recipes.json`)	paths differ between computers, and what an AI may run is decided per machine
`config` (`sync.auto`)	each computer decides how often it talks to the remote
Agent, grants and blocks	they live in the agent's memory and end with it
AI agents' access	an AI agent can use only the agent and the MCP server of the computer it runs on

So an AI agent on one computer can never reach the secrets on another. It reaches them only through that computer's own unlocked agent, within that computer's grants and recipes.

Where plaintext exists

Only on a computer that holds a recipient key, and only for a short time:

- inside the agent (the key) and the `mlcsecure` process doing the work;
- in the program started by `exec` or a recipe, as environment variables, pipes or temporary files that are deleted afterwards (tmpfs on Linux);
- on your terminal, if you ask for it with `get`.

Never in the store, never on the remote, never in the git history, and never in the output an AI agent sees (values there are masked as `***path***`). Details and limits: SECURITY.md.

Security model

What MLC Secrets protects, against whom — and where it stops. Read this before you trust it with something you cannot replace.

What is protected, and how

Asset	Protection
Secret values and files	encrypted with age (X25519) for every computer listed in `.mlcs/recipients`; the store only ever holds ciphertext
Metadata of a secret (kind, file name, origin of a backed-up server file)	inside the same encrypted envelope
A computer's private key	`~/.config/mlcsecrets/identity.age`, encrypted with your passphrase (scrypt); never in the store, never sent anywhere
Plaintext while in use	handed to programs through environment variables, pipes, or files in memory (tmpfs on Linux; on macOS and Windows a private per-user temp folder, which is on disk) that are deleted afterwards

Who sees what

Who	Sees	Does not see
The git remote (e.g. your NAS)	secret **names**, descriptions, group rules (`policy`), the list of computers' public keys, change times, git author names	values, file contents, file origins
Anyone with a copy of your identity file but not the passphrase	nothing more than the remote	values
A removed computer (`recipients rm`)	old versions it already has in its git history	anything changed after its removal — so change secrets it must no longer know
AI agents using `mlcsecure exec` / the MCP server	names, descriptions, and program output with secret values masked as `***path***`	values (unless a program transforms and prints them, see below)

What it defends against

- **Leaks by accident**: secrets in shell history, in `.env` files next to the code, in chat transcripts of AI assistants, in logs. Values are never accepted as arguments, `get` refuses to print into pipes, and output of `exec` and of MCP recipes is masked (raw, base64, URL and hex forms).
- **Honest tools that go too far**: `exec-only` groups are never shown; `on-demand` groups work only while you grant them; temporary blocks refuse anything. Everything that widens access — granting, lifting a block, adding a computer, changing a policy, defining a recipe — must be confirmed by a person at the terminal (or, for MCP recipes with `--confirm`, in the MCP client), so a script or AI agent cannot do it on its own.
- **AI clients without a shell** (MCP): they can only list names and run the recipes you defined; parameters are checked against patterns, no shell is involved, output is always masked, every run is logged.
- **Loss of one computer**: every computer has its own key; the store is synced through git. With a **recovery key** kept offline you can recover even if all computers are lost.

What it does not defend against

- **A process with full access to your user account while the agent is unlocked.** Such a process can talk to the agent and decrypt anything, set grants, edit `.mlcs/policy` or `recipes.json`. Masking, `exec-only` and `on-demand` stop accidents and well-behaved tools — not a determined attacker running as you. Keep the agent's time short (`agent start --ttl`) and lock it when you are done (`agent lock`).
- **Programs that leak what they are given.** A recipe or `exec` command is only as safe as the program it runs. Masking catches values printed as they are or in common encodings, not values that were reversed, split or re-encrypted.
- **root on your computer**, a compromised kernel, or malware that reads memory or keystrokes (your passphrase).
- **Copies your editor makes.** `edit` keeps its plaintext in memory and starts vim and neovim without swap, undo, backup and viminfo files, but VS Code, JetBrains IDEs, Sublime Text and others keep their own copies (local history, backups of unsaved changes) in your profile, out of mlcsecure's reach. `edit` names such editors; USAGE.md, "Editors and plaintext copies", says what to switch off.
- **Values shorter than 4 characters** are not masked (they would replace ordinary words); `exec` and the MCP server name them.
- **On Windows**, terminal confirmations are weaker: a program running as you can inject keystrokes into a console window, so "a person at the keyboard" is not proven as strongly as by `/dev/tty` on Linux and macOS. The agent's socket and temp folder are protected by an access list that admits only your user and SYSTEM; the identity file and the store rely on your profile folder's permissions (it is passphrase-encrypted anyway).

Your responsibilities

1. Choose a long passphrase on every computer; there is no reset.
2. Create a recovery key and keep it offline: `mlcsecure key generate --out recovery.txt`, add it with `mlcsecure recipients add age1… --name recovery`, `mlcsecure sync`, then store `recovery.txt` in a password manager or on paper and delete it from the disk. Without it, losing all computers' passphrases loses everything.
3. Keep the agent's lifetime short and lock it when an AI no longer needs it.
4. Never put a secret into a description — descriptions are readable.
5. Edit secrets in an editor that keeps no copies, or switch its copies off (USAGE.md, "Editors and plaintext copies").

Reporting a problem

MLC Secrets is maintained by Michael Lechner. Please report security issues privately rather than in a public issue.