vault.

The User Guide

Post-quantum file encryption.
With a secret trapdoor.

Built for skimming. Every chapter opens with a **TL;DR** and ends with a **Pocket this** recap. Procedures end with a **Done when** check, so you know it worked. The red boxes are the ones that can lose you data: read every one.

FrodoKEM-640 + ML-KEM-768 • decoy passphrase • one binary, no dependencies

uncomplex. Complicated things. Thoughtfully simplified.

Version 0.6.1-4-g5c470ab • October 2026

Hi. Start here.

This guide is built for skimming. Genuinely. You are allowed to jump around, read the boxes and ignore the paragraphs, or read chapter 5 first because that's the one with the commands in it. Nobody's checking.

Every chapter opens with a **TL;DR**. Read that and you've got the gist. The rest is there for when you want the detail, or when something goes wrong at 11pm and you need to know *exactly* what a message means.

The boxes, decoded

TL;DR

The whole chapter in about three lines. If you only read one thing, read this.

Quick win

A small thing you can do right now that makes life better.

Heads up

Something that will trip you up if you're not expecting it.

Here be dragons

Something that can lose you data. These are rare. Read every one of them.

Squirrel! *(a tangent; skip it if you like)*

A tangent. Interesting, possibly, but you can skip it and lose nothing.

Why though?

The reasoning behind a design decision, for the curious.

Pocket this

The end-of-chapter recap. A checklist of the things worth keeping in your head.

The absolute fastest route

Got two minutes? Here's all of it:

```
$ vault genkey            # make your key (pick a good passphrase)
$ vault init              # make your vault
$ vault add taxes.pdf     # put something in
$ vault ls                # see what's in there
$ vault get taxes.pdf     # get it back out
```

That's the tool. Everything after this page is detail, safety rails and the fun decoy trick in chapter 6. ☺

Contents

Start here i

1 Meet vault 1

what it is, what it isn't, and a 30-second mental model

1.1 What it actually does . . . 1

1.2 The three things that make it different . . . 1

1.3 What vault is *not* . . . 2

2 Install it 3

grab a binary, check it, put it on your PATH

2.1 The easy way on Linux: my package repository . . . 3

2.2 Pick your file . . . 3

2.3 Check the file hasn't been tampered with . . . 4

2.4 Put it on your PATH . . . 5

2.5 Did it work? . . . 5

2.6 Building it yourself . . . 7

3 Your key 8

the one file that matters most, and the passphrase that guards it

3.1 Making one . . . 8

3.2 Choosing a passphrase that doesn't suck . . . 9

3.3 Where it lives . . . 9

3.4 Back. It. Up. . . . 10

4 Your first vault 11

one command, and a peek inside

4.1 Create it . . . 11

4.2 Wait, why is there already stuff in it? . . . 11

4.3 More filler, for bigger secrets . . . 12

4.4 More than one vault . . . 12

5 Everyday moves 13

add, ls, get, mv, cp, rm: the commands you'll actually use

5.1 Putting things in: `add` . . . 13
5.2 Seeing what's inside: `ls` and `list` . . . 16
5.3 Getting things out: `get` . . . 16
5.4 Renaming, moving and copying: `mv` and `cp` . . . 17
5.5 Deleting: `rm` . . . 18
5.6 Not typing your passphrase every time: `unlock` . . . 19
5.7 Flags work anywhere . . . 20

6 The decoy passphrase 22

one key, two passphrases, two completely different vaults

6.1 The idea . . . 22
6.2 How it looks . . . 22
6.3 Setting it up . . . 23
6.4 Why it holds up . . . 23
6.5 Making it believable: the playbook . . . 24
6.6 The honest limits . . . 24

7 Scripts and automation 25

environment variables, exit codes, and a few recipes

7.1 The environment variables . . . 25
7.2 Exit codes and output . . . 26
7.3 Recipes . . . 26

8 Locking, syncing and backups 28

two terminals, two machines, and keeping copies

8.1 Locking is automatic . . . 28
8.2 Syncing between machines . . . 28
8.3 Backups . . . 29

9 Under the hood 30

the cryptography, the file layout, and exactly what a thief learns

9.1 The cryptography . . . 30
9.2 What's in the folder . . . 31
9.3 Why big files are cut up . . . 31
9.4 What an attacker learns . . . 32
9.5 What happens during `add` . . . 32

10 When things go sideways **34**

every message vault can show you, and what to do about it

10.1 The big table . . . 35

10.2 “My files have vanished!” . . . 37

10.3 Interrupted in the middle of something? . . . 37

11 Cheat sheet **39**

the whole tool at a glance. print it, pin it up

A Glossary **41**

B About this guide **42**

Meet vault

what it is, what it isn't, and a 30-second mental model

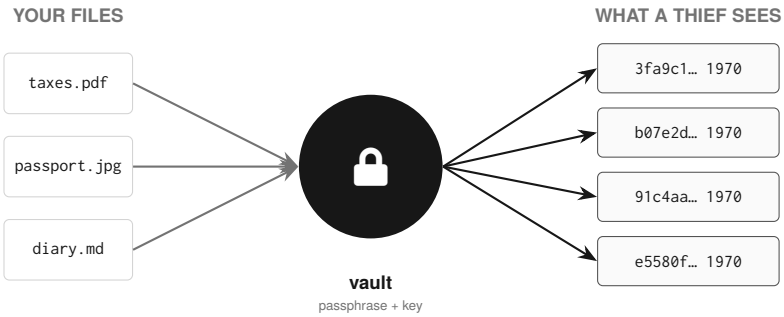
TL;DR

vault is one small program that keeps your files encrypted in a folder. It uses post-quantum cryptography, hides your file names and sizes, and has an optional decoy passphrase that shows a harmless set of files instead of the real ones.

1.1 What it actually does

You give vault a file. It encrypts the file and drops the result into your *vault*, which is just a folder. Later you ask for the file back and vault decrypts it. That's the core loop. Put in, take out.

The interesting part is what the folder looks like from the outside. Nothing. Well, not *nothing*. A pile of randomly named files, all dated 1 January 1970, all in a handful of suspiciously round sizes, none bigger than 4 MiB, scattered across folders named `00` to `ff`. No names. No real sizes. No dates. None of *your* folders. Big files are cut into pieces that look like everything else. Someone who steals the folder learns roughly how much stuff you have, and that's about it.



1.2 The three things that make it different

It's post-quantum

Most encryption you use today would, in theory, fall to a big enough quantum computer. Nobody has one yet. But someone could copy your encrypted files *today* and sit on them until they do. It's called *harvest now, decrypt later*, and it's a real strategy.

vault uses two post-quantum algorithms at once: **FrodoKEM-640** and **ML-KEM-768**. An attacker has to break *both*. Chapter 9 has the gory details.

It hides metadata, not just contents

Plenty of tools encrypt what's *in* your files but happily leave `divorce-papers-final-v3.pdf` sitting there in plain sight. vault encrypts the names too. And the sizes. And the timestamps.

It can lie for you

Optionally, your key can have a second, **decoy** passphrase. Type it, and vault shows a completely separate set of files. Your real files don't appear, don't error, don't leave a hint. They just aren't there, as far as anyone watching can tell. That's chapter 6, and yes, it's the fun one.

1.3 What vault is *not*

Honesty time. Knowing the edges matters as much as knowing the middle.

- **Not a sync service.** It's a folder. You can sync the folder yourself (chapter 8), but vault doesn't do it for you.
- **Not a backup by itself.** One copy of anything is zero copies. Back up the vault folder *and* your key.
- **Not magic against malware.** If something nasty is running on your computer while you type your passphrase, it can watch you type it. No file-encryption tool fixes a compromised machine.
- **Not recoverable without the passphrase.** There is no reset link. There's no "forgot password". That's the whole point, and it cuts both ways.

> **Squirrel!** *(a tangent; skip it if you like)*
> Why is it called *vault* and not something cooler? Because you type it a lot, and five letters you already know how to spell beat a clever name every single time. Also, all the cool names were taken.

> **Pocket this**
>
> - [ ] vault encrypts files into a folder and hides their names, sizes and dates.
> - [ ] The encryption is post-quantum: two algorithms, attacker must break both.
> - [ ] An optional decoy passphrase shows a different, harmless set of files.
> - [ ] Lose the passphrase and the files are gone. No reset. Ever.

Install it

grab a binary, check it, put it on your PATH

TL;DR

Download the one file for your system, check its SHA-256, rename it to `vault` (or `vault.exe`), and put it somewhere on your `PATH`. Or build every platform yourself with `./build.sh`.

2.1 The easy way on Linux: my package repository

On Debian, Ubuntu, Fedora, RHEL, openSUSE or Arch, add my signed repository once and install the `dixon-vault` package with your usual package manager. You get the manual pages too, and new releases arrive with your normal system updates. On Debian or Ubuntu:

```
$ sudo install -d -m 0755 /etc/apt/keyrings
$ curl -fsSL https://repo.dixon.cx/dixon.gpg | sudo tee /etc/apt/keyrings/dixon.gpg >/dev/null
$ echo "deb [signed-by=/etc/apt/keyrings/dixon.gpg] https://repo.dixon.cx/apt stable main" |
    sudo tee /etc/apt/sources.list.d/dixon.list
$ sudo apt update && sudo apt install dixon-vault
```

The commands for dnf, zypper and pacman are at `dixon.cx/repo`. The package is called `dixon-vault` because plain `vault` is HashiCorp Vault's name; the command it installs is still `vault`. Everyone else, read on.

2.2 Pick your file

vault is a single self-contained program. No installer, no runtime, no dependencies. You just need the right file for your operating system and CPU.

YOU HAVE	WHICH MEANS	DOWNLOAD
Windows PC	most laptops and desktops	`vault-windows-amd64.exe`
Windows on ARM	Surface Pro X, Snapdragon laptops	`vault-windows-arm64.exe`
Mac, 2020 or later	Apple silicon (M1, M2, M3…)	`vault-darwin-arm64`
Mac, older	Intel	`vault-darwin-amd64`
Linux PC or server	almost everything	`vault-linux-amd64`
Linux on ARM64	Pi 4/5 (64-bit OS), Graviton, Ampere	`vault-linux-arm64`
Raspberry Pi, 32-bit	Pi 2, 3, 4 on a 32-bit OS	`vault-linux-armv7`
Raspberry Pi Zero / 1	the tiny older ones	`vault-linux-armv6`
Android	in Termux: the Linux ARM64 file	`vault-linux-arm64`
FreeBSD	amd64 or 386	`vault-freebsd-amd64 etc.`
Something exotic	RISC-V, POWER (little-endian), s390x, LoongArch	`see dist/`

Squirrel! *(a tangent; skip it if you like)*
The release covers sixteen builds for Linux, Windows, macOS and FreeBSD, from Windows on ARM to IBM mainframes. The older Go builds reached further, to Plan 9 and WebAssembly; `docs/PLATFORMS.md` says why those went. Will you ever need `vault-linux-s390x`? Almost certainly not. Is it pleasing that it exists? Extremely.

Heads up
Not sure which Linux you have? Run `uname -m`. `x86_64` means amd64, `aarch64` means arm64, `armv7l` means armv7, and `armv6l` means armv6.

2.3 Check the file hasn't been tampered with

Every release comes with a `SHA256SUMS` file. It takes ten seconds to check, and it's the difference between running *vault* and running *something pretending to be vault*. For an encryption tool, that matters.

```
# Linux
$ sha256sum -c SHA256SUMS --ignore-missing

# macOS
$ shasum -a 256 -c SHA256SUMS --ignore-missing

# Windows (PowerShell): compare the output with the line in SHA256SUMS
PS> Get-FileHash .\vault-windows-amd64.exe -Algorithm SHA256
```

You want to see `OK`. Anything else: stop, and download it again from the source you trust.

2.4 Put it on your PATH

Linux and macOS

```
$ chmod +x vault-linux-amd64
$ sudo mv vault-linux-amd64 /usr/local/bin/vault
$ vault help
```

No admin rights? Use `~/.local/bin` instead, which most modern distributions already have on the `PATH`.

> ⚠ **Heads up**
> **macOS says the file "cannot be opened"?** It's not signed by Apple, so Gatekeeper quarantines it. Once you've checked the SHA-256, clear the flag: `xattr -d com.apple.quarantine /usr/local/bin/vault`.

Windows

1. Make a folder, say `C:\Tools`, and move the file in, renamed to `vault.exe`.
2. Start menu, search for "**environment variables**", open **Edit the environment variables for your account**.
3. Edit **Path**, add `C:\Tools`, and click OK.
4. Open a *new* terminal and run `vault help`.

Android (Termux)

```
$ curl -LO https://.../vault-linux-arm64
$ chmod +x vault-linux-arm64 && mv vault-linux-arm64 $PREFIX/bin/vault
```

2.5 Did it work?

Run `vault help`. You should see the full command list:



`vault help`: everything vault can do, on one screen

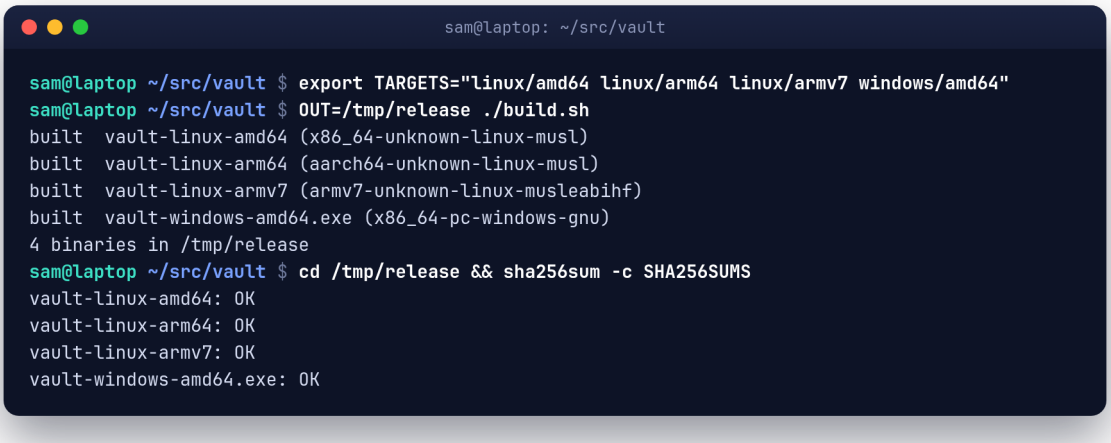
Done when `vault help` prints the command list from a new terminal, in any folder.

2.6 Building it yourself

Prefer to build from source? You need Rust, through `rustup`, which fetches the exact compiler version the project pins. Then:

```
$ cargo build --release --locked   # just your platform: target/release/vault
$ ./build.sh                       # every platform it can, into ./dist, with SHA256SUMS
```

`build.sh` cross-compiles Linux and Windows with `cargo-zigbuild` and `zig`, macOS on a Mac, and FreeBSD with `cross` and Docker; it says which targets your machine can't build. 32-bit ARM Linux gets two builds, armv6 and armv7. Want just a few? Set `TARGETS`. Want them somewhere else? Set `OUT`.



A partial build, followed by checking every binary against SHA256SUMS

Why though?

The Linux binaries are *static*: they need no system libraries at all. Every build uses the pinned compiler and the locked versions of every dependency, with build paths stripped, so the result doesn't depend on whose machine or which folder built it. Built with the same compiler for the same target, the same commit should give the same file, so you can check a release against your own build.

Pocket this

- [ ] One file per platform. Rename it `vault` and put it on your `PATH`.
- [ ] Always check the SHA-256 against `SHA256SUMS`.
- [ ] macOS: clear the quarantine flag after checking.
- [ ] `./build.sh` builds every platform your machine can, from source.

Your key

the one file that matters most, and the passphrase that guards it

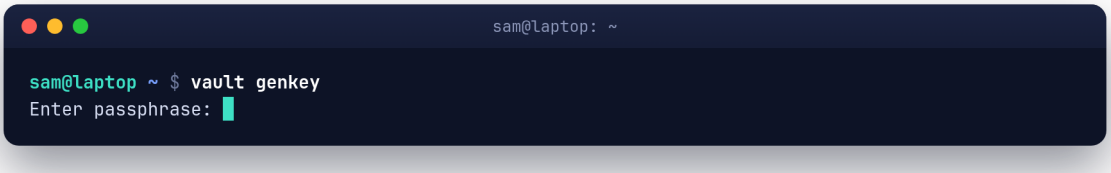
TL;DR

`vault genkey` creates your key at `~/.vault/key`, protected by a passphrase. It can also set a decoy passphrase. **Back the key file up.** Without it, and its passphrase, your files cannot be recovered by anyone.

3.1 Making one

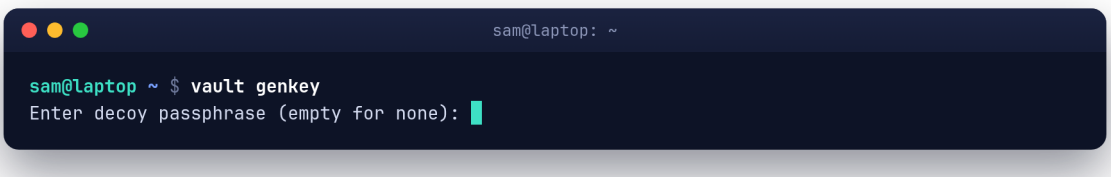
```
$ vault genkey
```

vault asks for a passphrase, then asks you to type it again. Nothing appears on screen as you type. That's deliberate, not broken.



Waiting for your passphrase. Typing is invisible on purpose.

Next it offers the **decoy passphrase**. Press Enter to skip it, or set one now. We'll cover what it's for in chapter 6. The short version: it opens a separate, fake-but-plausible set of files.



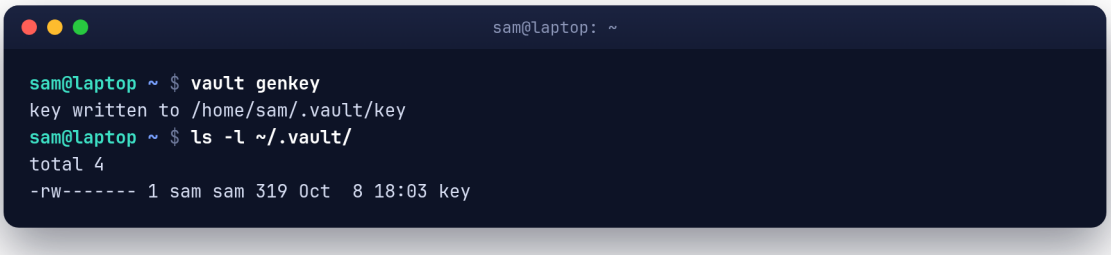
The optional decoy passphrase. An empty answer means "no decoy".

Heads up

The decoy can only be set here, when the key is made. There's no "add a decoy later" command. If you think you might ever want one, set one now. It costs nothing, and nobody can tell

it's there.

When it's done, you get one line and one small file:



Your key: 319 bytes, readable only by you

Done when vault says `key written to …` and `ls -l ~/.vault/` shows a 319-byte `key` that only you can read. Then back it up (see below).

3.2 Choosing a passphrase that doesn't suck

The key file is encrypted with your passphrase using **Argon2id**, a deliberately slow, memory-hungry function. Each guess costs an attacker real time and 64 MiB of memory. Slow guessing only helps, though, if there are a lot of guesses to make.

- ☑ **Use four or more random words.** *correct horse battery staple* beats *P@ssw0rd!* by a mile, and you can actually remember it.
- ☑ **Make it unique.** Not your email password. Not your bank's.
- ☑ **Make the decoy believable.** It should feel like something you'd genuinely pick. More in chapter 6.

> **Quick win**
> Use a password manager's *passphrase* generator (most have one) to make the words, then practise typing it five times. Muscle memory is a legitimate security tool.

3.3 Where it lives

By default your key is `~/.vault/key`. On Windows, that's `%USERPROFILE%\.vault\key`. Want it somewhere else, like a USB stick? Two options:

```
$ vault genkey --key /media/usb/vault.key     # this once
$ export VAULT_KEY=/media/usb/vault.key       # every command from now on
```

> **Squirrel!** *(a tangent; skip it if you like)*
> Keeping the key on a USB stick and the vault on your laptop is a neat trick. Pull the stick out and the vault is inert, even if someone knows your passphrase. Two things to steal instead of one.

3.4 Back. It. Up.

Here be dragons

If you lose the key file, or forget the passphrase, your files are gone. Not “hard to get back”. Gone. vault has no recovery mechanism, no master key, no back door. That’s what makes it trustworthy, and it’s also what makes a lost key final.

The key file is encrypted, so a copy is safe to keep somewhere a little less private than your laptop. Some decent options:

- A USB stick in a drawer. Or two, in two different drawers.
- Your password manager, as an attached file.
- Printed out. It’s only a few lines of text. Paper survives surprisingly well.

Why though?

What’s actually in the key file? Two equal-sized encrypted slots, in random order. Each passphrase opens exactly one. When there’s no decoy, the spare slot is just random bytes, and nobody can tell random bytes from an encrypted slot. So the file gives away nothing about whether a decoy exists.

Pocket this

- [ ] `vault genkey` makes `~/.vault/key`. It won’t overwrite an existing key.
- [ ] Four-plus random words. Unique. Practise it.
- [ ] Decide on a decoy now. It can’t be added later.
- [ ] Back up the key file. Somewhere else. Today.

Your first vault

one command, and a peek inside

TL;DR

`vault init` creates an empty vault at `~/.vault/store`. It's just a folder. Inside it seeds some random filler, which is normal and part of the decoy trick.

4.1 Create it

```
$ vault init
```

That's it. No passphrase needed, because an empty vault has nothing to protect yet.



A fresh vault. Two sub-folders, a marker, and… records already?!

Done when vault says `vault created at …` and the folder holds `.vault`, `objects` and `records`.

Inside `records/` and `objects/` are 256 folders each, named `00` to `ff`. Every encrypted file lives in the one matching the first two characters of its name. That just keeps any one folder from getting huge, which some file systems and sync services dislike. All 256 exist from the start, so the layout itself gives nothing away.

4.2 Wait, why is there already stuff in it?

Good catch. A brand new vault holds somewhere between 8 and 32 entries of random filler. They're encrypted to a key that was thrown away the moment it was used, so *nobody* can open them. Not you, not us, not anyone.

It looks weird, but it's what makes the decoy passphrase believable. Every vault, decoy or not, always contains entries you can't open. So when someone with your decoy passphrase sees unopenable entries, that's completely unremarkable. It's just filler. (Some of it might not be. They can't tell.)

> **Heads up**
> By default the filler takes a few tens of megabytes. That's the price of plausible deniability, and it's paid once.

4.3 More filler, for bigger secrets

A few tens of megabytes of filler can only cover for a few files. If you plan to hide more than that behind a decoy, ask for more when you create the vault:

```
$ vault init --filler 2G
```

vault keeps writing filler until there's at least that much. Sizes take `K`, `M`, `G` or `T`. The filler entries are built exactly like real ones, from a mix of small and large random "files", so a big file of yours sits among big files of filler. It costs the disk space and a few seconds per gigabyte, once. It can only be chosen at `init`: added later, the filler would show up as a burst of new entries, which is exactly what it's meant to hide.

> **Quick win**
> Rule of thumb: set `-filler` to at least as much as you ever expect to hide from the decoy passphrase. Without a decoy, you don't need it.

4.4 More than one vault

Want a separate vault for work stuff? Point `-vault` anywhere:

```
$ vault init --vault ~/work-vault
$ vault add contract.pdf --vault ~/work-vault
```

Or set `VAULT_DIR` so every command uses it. The same key works with any number of vaults.

> **Quick win**
> Add `export VAULT_DIR=…` to your shell profile and you'll never type `-vault` again.

> **Pocket this**
> - ☐ `vault init` makes `~/.vault/store`. `-vault DIR` to put it elsewhere.
> - ☐ Filler entries are normal. Nobody can open them. That's the point.
> - ☐ Hiding a lot behind a decoy? `vault init -filler 2G` (or however much you'll hide).
> - ☐ One key, as many vaults as you like.

Everyday moves

add, ls, get, mv, cp, rm: the commands you'll actually use

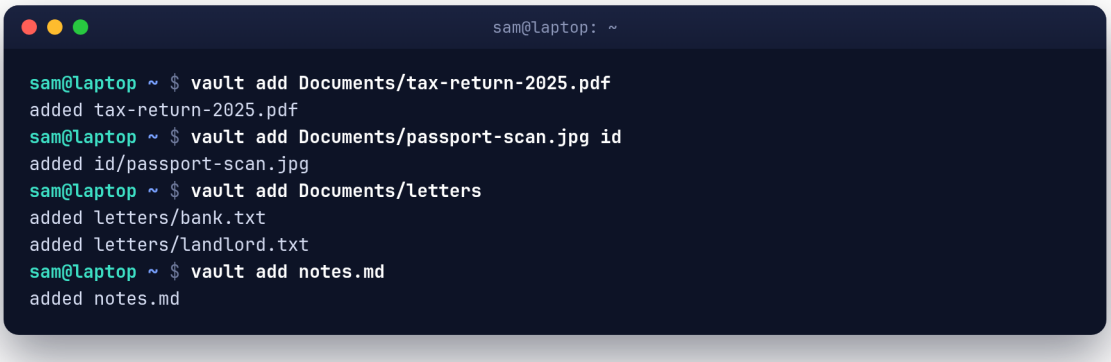
TL;DR

`add` puts files in (optionally into a folder), `ls` shows them a folder at a time, `get` takes them back out, `mv` and `cp` move and copy, `rm` deletes (folders with `-r` throughout). Each asks for your passphrase, unless you've run `unlock`. That's the whole daily workflow.

5.1 Putting things in: `add`

```
$ vault add FILE-OR-FOLDER-OR-URL [FOLDER-IN-VAULT]
```

Give it a file, and it goes in under its own name. Give it a second argument, and it goes into that folder inside the vault. Give it a whole folder, and everything inside comes along too, subfolders and all.



Adding a file, a file into a folder, a whole folder, and another file

Notice what happened there:

- `Documents/tax-return-2025.pdf` became plain `tax-return-2025.pdf`. vault keeps the *file name*, not the path you typed.
- With a second argument, `id`, the passport scan landed in `id/passport-scan.jpg`.
- Adding the folder `Documents/letters` kept its own name and structure: `letters/bank.txt`, `letters/landlord.txt`.
- Adding `.` (or `..`) brings in that folder's *contents*, subfolders and all, without a folder name of its own.

> ⚠ **Heads up**
> **Folders inside a vault aren't real folders.** They're just part of each file's name, like `letters/bank.txt`. So there's no `mkdir`, and a folder disappears when its last file goes. No housekeeping required.

Adding something that's already there

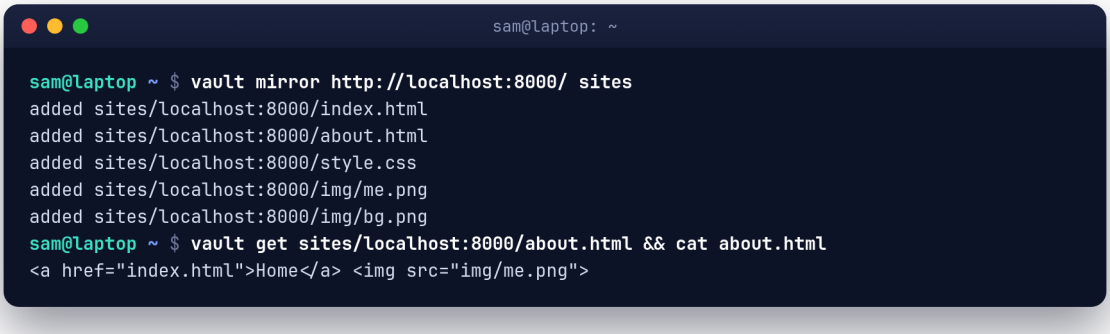
Adding a name that already exists **replaces** it, and vault tells you so. The old version is deleted *after* the new one is safely stored, so a crash halfway through never leaves you with neither.

Straight from the web

Give `add` an `http://` or `https://` address and vault downloads it (as Chrome, so fussy sites play along) and stores it under the last part of the address, such as `report.pdf`. The download sits in a private temporary file only until it's encrypted.

Copying a website: `mirror`

`mirror` takes an `http://` or `https://` address and copies the whole site, one request at a time, into a folder named after its host. Links inside pages and stylesheets are rewritten to relative paths, so the copy still browses once you get it out. Only the same host is followed.



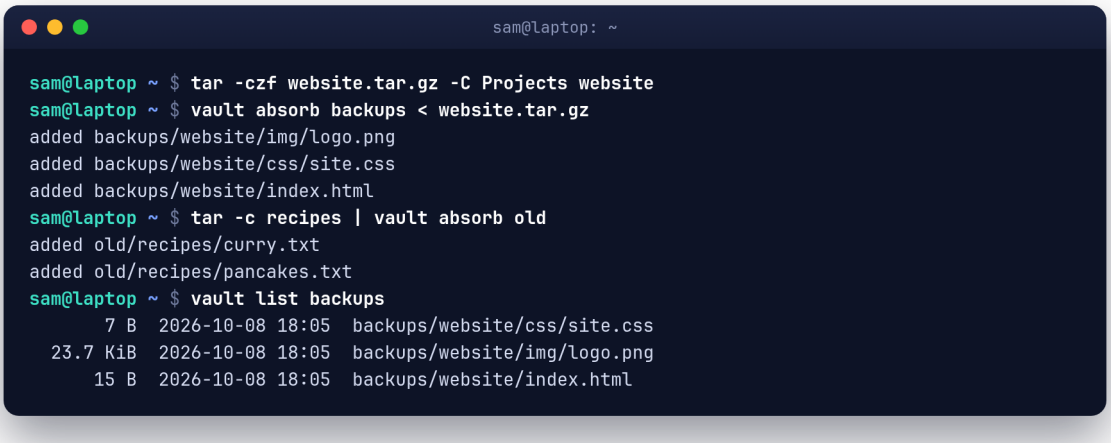
📷 Mirroring a small site. The links in each page now point at the copy.

From a tar archive: `absorb`

`absorb` reads a tar archive, gzipped or not, from standard input and stores every plain file in it, optionally under a folder:

```
tar -cz photos | vault absorb backup
ssh server 'tar -c /etc' | vault absorb server1
vault absorb old < archive.tar.gz
```

Each file is encrypted straight from the stream, so nothing is unpacked to disk first. Directories need no entry of their own; links, devices and any name that tries to climb out with `..` are skipped with a note. vault asks for your passphrase only once the archive starts arriving, so when it comes out of something that asks first, like `gpg -decrypt secrets.tar.gpg | vault absorb`, gpg's prompt comes and goes before vault's appears. If the archive stops halfway through a file, that file isn't stored and vault stops; everything before it is already safely in.

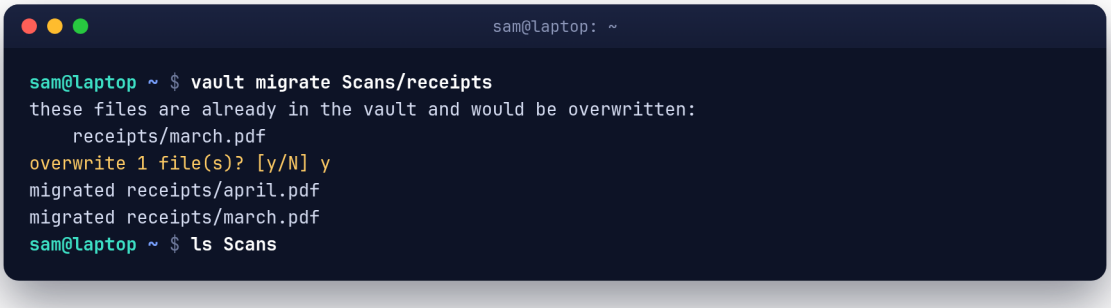


A gzipped tarball and a plain tar stream, both absorbed.

Moving in: `migrate`

`migrate` works exactly like `add`, but deletes each original once it's safely stored, then tidies away any folders left empty. It won't take a URL; there's no original to delete.

Because the originals go, `migrate` checks first: if anything would **replace** a file already in the vault, it lists those names and asks before touching anything. Answer anything but `y` and nothing changes. With no terminal to ask on, the answer is no. For scripts, `-yes` answers yes up front.



Moving a folder in. One file was already there, so vault asked first.

What gets skipped

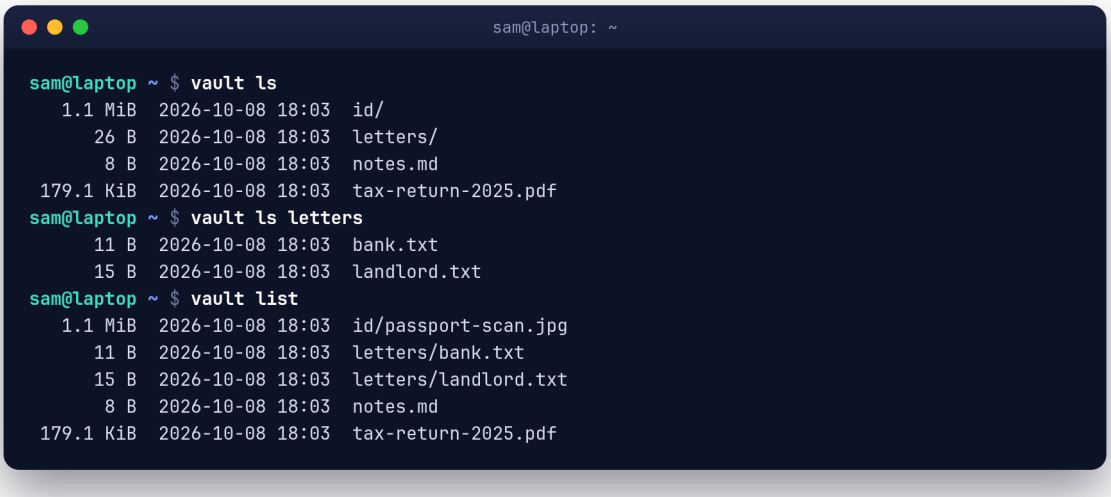
When adding a folder, anything that isn't a plain file, such as symbolic links, sockets or devices, is skipped with a note. So are the vault and key themselves, should they live inside the folder, so `migrate` can never eat them. Files larger than 1 TiB are refused. (If you have a single 1 TiB file, we have questions.)

5.2 Seeing what's inside: `ls` and `list`

```
$ vault ls              # the top level: folders (ending in /) and files
$ vault ls letters      # one level down, inside letters/
$ vault list            # every file, all the way down, full names
```

`ls` works like the shell's: it shows one folder at a time. Each sub-folder appears once, ending in `/`, with the total size of everything inside it and its newest date. Files show their size and their *own* last-modified time from when you added them. Folders come first, then files. Keep going down with `vault ls letters/2025`.

`list` is the whole catalogue in one go: every file under a folder, with its full name, sorted. It's the one to pipe into other commands, and `vault ls -r` does the same.



One level at a time with `ls`, then everything with `list`

Why though?

Why does `list` need my passphrase? Because the names are encrypted too. Without the key, vault can't read the catalogue any better than a thief can.

5.3 Getting things out: `get`

```
$ vault get NAME              # writes the file here, under its own name
$ vault get NAME -o OUTPUT    # writes it wherever you say
$ vault get -r FOLDER         # a whole folder, into ./FOLDER
```



Getting files back, including vault refusing to overwrite

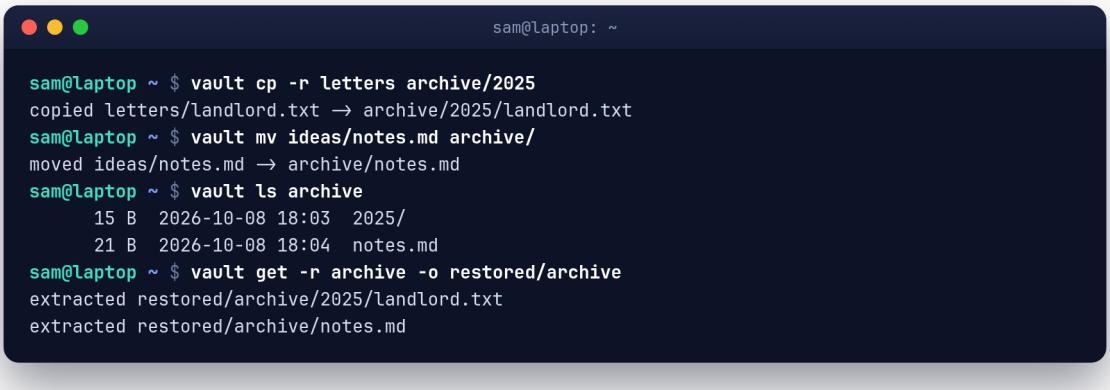
Three safety habits are built in, so you don't need to think about them:

- ☑ **It never overwrites.** If the output exists, vault stops. Move the old one or use `-o`.
- ☑ **Nothing appears until everything checks out.** vault writes to a hidden temporary file, verifies every single byte, and only then gives it its real name. A tampered or damaged file never shows up half-written.
- ☑ **It restores the details.** You get back the original permissions and modification time.

5.4 Renaming, moving and copying: `mv` and `cp`

```
$ vault mv OLD NEW         # rename a file or a whole folder
$ vault mv notes.md ideas/ # move into a folder, keeping the name
$ vault cp OLD NEW         # copy a file
$ vault cp -r OLD NEW      # copy a folder
```

They work like the everyday `mv` and `cp`: if `NEW` ends in `/` or is a folder already, `OLD` goes inside it. Neither will overwrite an existing name, and neither re-encrypts anything, so both are instant however big the files are. A copy shares the original's encrypted data, so it takes no extra space, and deleting one leaves the other intact. `get -r` brings a whole folder back out, keeping its shape.

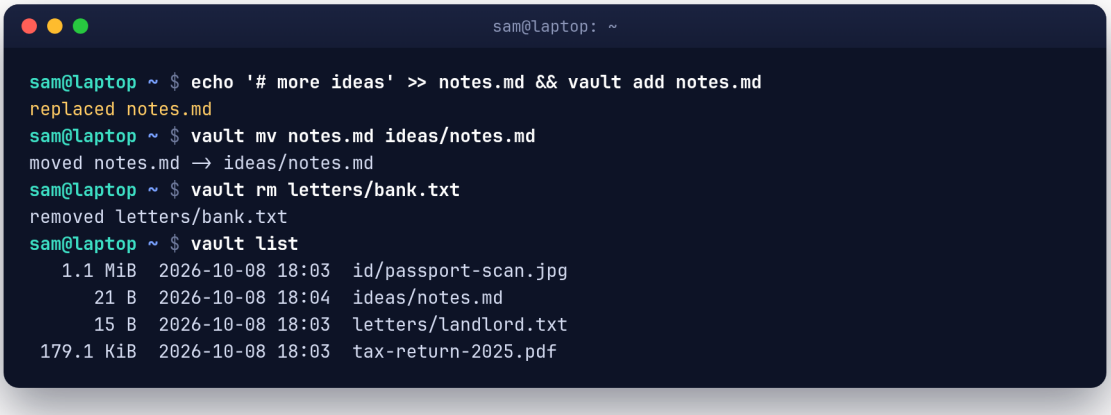


A folder copied, a file moved into it, and the lot brought back out

5.5 Deleting: `rm`

```
$ vault rm NAME          # one file
$ vault rm -r FOLDER     # a folder and everything in it; asks first
$ vault rm -rf FOLDER    # the same, without asking
```

Removes the file's catalogue entry and every encrypted piece of it, so the space comes back straight away.

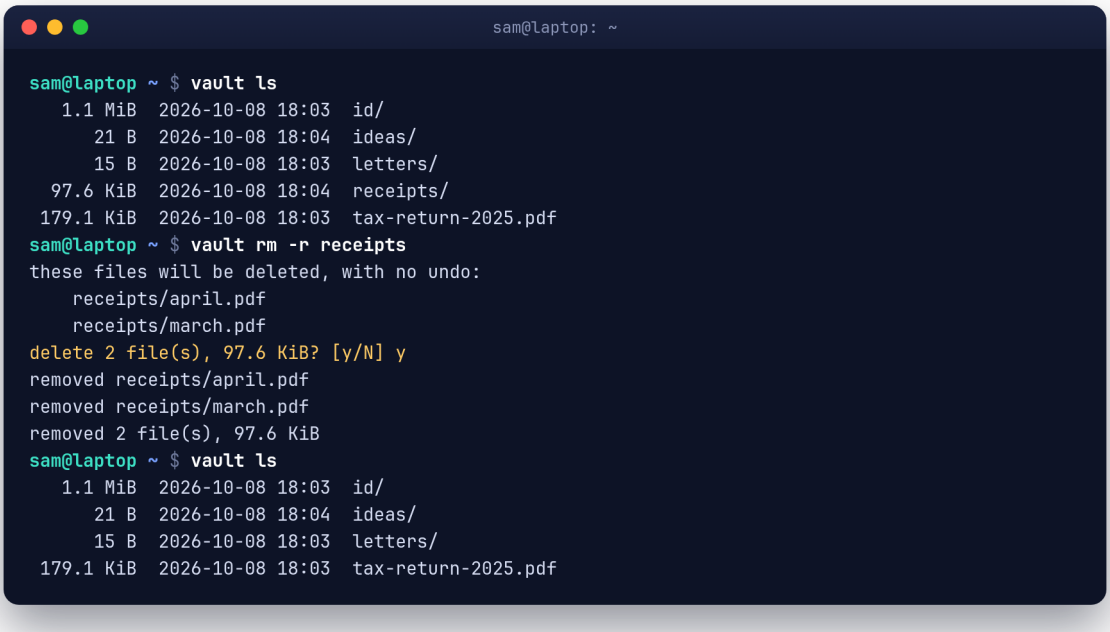


Replace, move, delete, and the result

> **Here be dragons**
>
> **`rm` is immediate and there's no undo.** No recycle bin. If it matters, make sure it's backed up somewhere first.

Deleting a whole folder

`rm -r` deletes a folder and everything under it. Because that can be a lot, it first lists what's going (the first 20 names, then a count) with the total size, and asks. Anything but `y` deletes nothing, and so does having no terminal to ask on. `-f` skips the question, and also means "don't complain if there's nothing there", so `rm -rf` is the one for scripts. `vault rm -r /` is refused rather than emptying the vault.

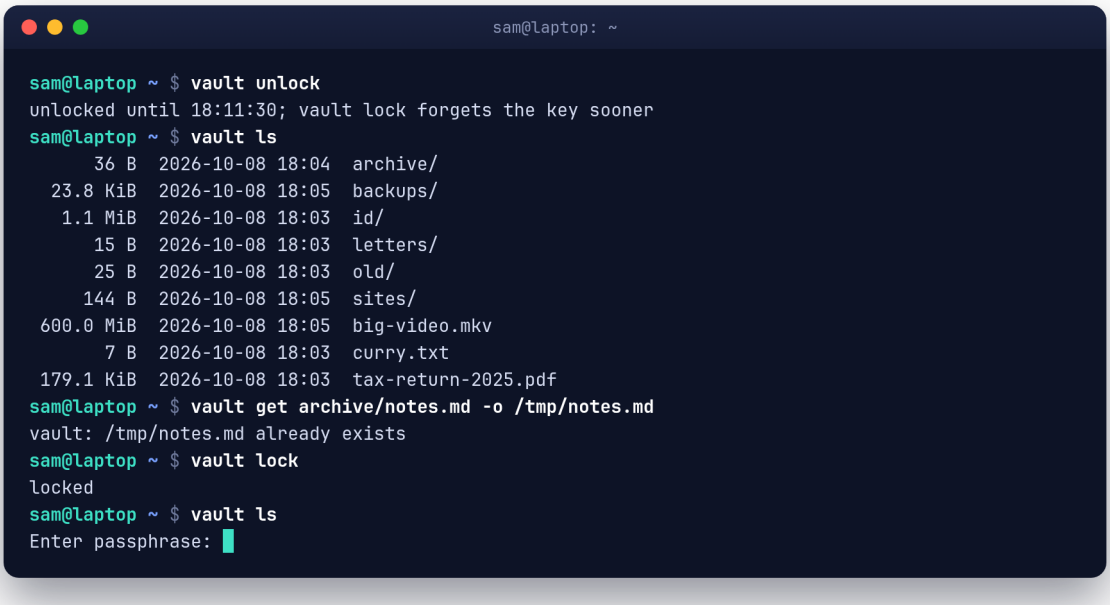


A folder deleted: vault lists it, asks, then frees the space

5.6 Not typing your passphrase every time: `unlock`

```
$ vault unlock             # ask once, remember the key for 5 minutes
$ vault unlock --for 30m   # or for as long as you say
$ vault unlock --idle      # 5 minutes since you last used it
$ vault lock               # forget it right now
```

After `unlock`, every command skips the passphrase prompt until the time is up. Normally that's a hard deadline, counted from when you unlocked, so a laptop left open still locks itself; run `unlock` again for a fresh countdown. If you'd rather it stayed unlocked while you're busy, `--idle` counts from the last command that used the key instead, so it locks only after that long without use.



Unlock once, work without prompts, lock when you're done

How it keeps the key safe

`unlock` starts a small helper process, much like `ssh-agent`, that holds your unlocked key in memory and nowhere else:

- **Never on disk.** The key goes from `unlock` to the helper through a pipe. Its memory is locked so it can't be swapped out, and on Linux the helper can't be debugged or core-dumped, even by you.
- **Only you can reach it.** It answers on a socket inside a folder only your user account can open (in `$XDG_RUNTIME_DIR`, or your temporary folder). vault checks that folder's owner and permissions every time.
- **It wipes itself.** At the deadline, or on `vault lock`, the key is overwritten and the helper exits. Rebooting ends it too.
- **An explicit passphrase wins.** If `VAULT_PASSPHRASE` is set, vault uses that and ignores the helper, so scripts behave exactly as before.

> **Heads up**
> **While it's unlocked, anyone at your keyboard is you.** Anything running as your user can use the key without a passphrase. Lock before you walk away, and before handing your machine to anyone, decoy or not (see *The honest limits* in the next chapter).

5.7 Flags work anywhere

Every command understands `-key PATH` and `-vault DIR`, and they can go before, between or after the other arguments. These all do the same thing:

```
$ vault add --vault ~/work report.pdf
$ vault add report.pdf --vault ~/work
$ vault add report.pdf --vault=~/work
```

Pocket this

- ☐ `add FILE [FOLDER]`: keeps the file's name, optionally in a folder. Folders add recursively.
- ☐ Adding an existing name replaces it, safely.
- ☐ `ls [FOLDER]` one level at a time, `list [FOLDER]` everything.
- ☐ `get [-r] NAME [-o OUT]`, `mv OLD NEW`, `cp [-r] OLD NEW`, `rm NAME`, `rm -r FOLDER`.
- ☐ `unlock` remembers the key for 5 minutes; `lock` forgets it.
- ☐ `get` never overwrites and never publishes a half-checked file.
- ☐ `rm` has no undo.

The decoy passphrase

one key, two passphrases, two completely different vaults

TL;DR

Set a decoy passphrase when you run `vault genkey`. Every command run with the decoy sees only decoy files. Your real files are invisible to it: they don't list, can't be fetched, and look exactly like the random filler every vault has. Make the decoy believable, and keep it at least as big as the real thing.

6.1 The idea

Sometimes "I won't tell you my passphrase" isn't an option. A border crossing. A pushy partner. Someone standing over you. The decoy passphrase is for that moment. You type it, the vault opens, and it's full of genuinely private-looking stuff: bank letters, some personal notes, a few scans. Everything behaves normally. It's just not your *real* stuff.

6.2 How it looks

Here's one vault, opened twice, with the two different passphrases:

real passphrase

```
sam@laptop ~ $ vault list
   1.1 MiB  2026-10-08 18:03  id/passport-scan.jpg
      21 B  2026-10-08 18:04  ideas/notes.md
      15 B  2026-10-08 18:03  letters/landlord.txt
 179.1 KiB  2026-10-08 18:03  tax-return-2025.pdf
```

decoy passphrase

```
sam@laptop ~ $ vault list
       7 B  2026-10-08 18:03  recipes/curry.txt
      18 B  2026-10-08 18:03  recipes/pancakes.txt
      21 B  2026-10-08 18:04  shopping/notes.md
```

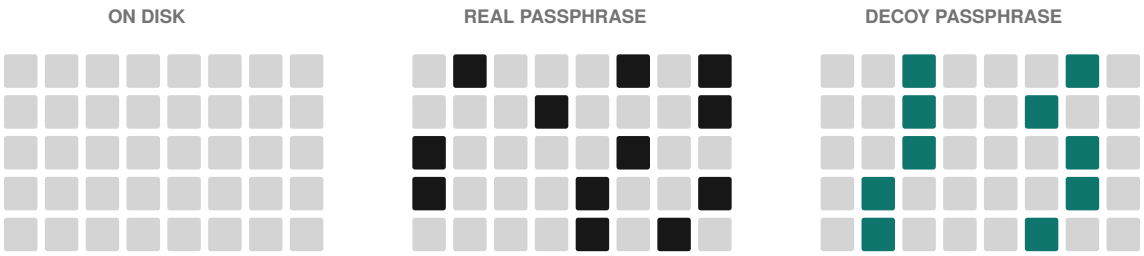
Same key file, same vault folder, two passphrases, two worlds

Nothing on the right hints that the left exists. Ask the decoy for a real file and you get the same message you'd get for a typo. The folder's entry count doesn't give it away either, because filler is always there:

sam@laptop: ~

```
sam@laptop ~ $ vault get ideas/notes.md -o /tmp/x.md
vault: ideas/notes.md: not in the vault
sam@laptop ~ $ find ~/.vault/store/records -type f | wc -l
19
```

The decoy can't see real files, and a few dozen entries on disk is unremarkable



Every square is an encrypted entry. Each passphrase lights up only its own. Grey is filler, or the other side.

6.3 Setting it up

The decoy is chosen when the key is made, and only then:

1. Run `vault genkey`. Type your *real* passphrase, twice.
2. At **Enter decoy passphrase (empty for none)**, type the decoy. Twice.
3. Run `vault init`, adding `-filler` with at least as much as you plan to hide, say `-filler 5G`.
4. Add some decoy files, typing the *decoy* passphrase when asked.
5. Add your real files, typing the *real* passphrase.

The two must be different (vault checks). In scripts, set `VAULT_DECOY_PASSPHRASE` alongside `VAULT_PASSPHRASE` before `genkey`. Chapter 7 has more on scripting.

Done when `vault list` with each passphrase shows only that passphrase's files, as in the screenshot above.

> **Heads up**
> **Already have a key without a decoy?** You'll need a new key: run `genkey` with a different `-key`, make a new vault, `get` everything out of the old one and `add` it to the new. The key file format deliberately gives no way to tell whether a decoy slot is in use, so there's nothing to safely "upgrade".

6.4 Why it holds up

- **The key file can't tell.** It always has two slots of identical size in random order. With no decoy, one slot is random noise, which looks the same as an encrypted slot. Both are tried every time, so even the timing is the same.
- **The vault can't tell.** Real and decoy entries are sealed to different keys, but they're the same size, the same shape, and all dated 1970. Nothing on disk says which key an entry belongs to.
- **The filler covers for you.** Every vault starts with at least 8 to 32 entries nobody can open, and as much more as you ask for with `-filler`. So "some entries I can't open" is true of *every* vault, decoy or not.
- **The errors are identical.** A real file asked for by name with the decoy passphrase gives exactly the same "not in the vault" as a typo.

6.5 Making it believable: the playbook

A decoy only works if it looks like the real thing. An empty decoy vault, or one with two files called `test.txt`, is a neon sign.

- ☑ **Fill it with plausibly private stuff.** Bank letters, payslips, a diary-ish notes file, scans of documents. The kind of thing someone *would* encrypt.
- ☑ **Keep it bigger than your real collection**, or at least as big. See the next section for why.
- ☑ **Keep it alive.** Add to it now and then, so the dates in `list` look like a real person uses it.
- ☑ **Know it cold.** Practise the decoy as much as the real one. Hesitating on the "real" passphrase is a tell.
- ☑ **Never mention it.** Not in notes, not in a README, not in a script sitting on the same machine.

6.6 The honest limits

> ⚠ **Heads up**
> **Numbers can give you away.** Someone holding the decoy passphrase can count the records and pieces they can't open. Without `-filler` there are only a few tens of megabytes of filler, so if `records/` holds 400 entries, or `objects/` holds a 300 MB video's 72 pieces, the filler can't explain the difference. They can't see *what* the leftovers are, but they can see there's more than filler. Create the vault with `-filler` at least as large as what you'll hide, and keep the decoy collection believable next to it.

> ⚠ **Heads up**
> **Growth over time is visible.** If someone copies your vault folder today and again next month, they can see it grew, even if the decoy didn't change. They can't see *what* was added. They can see *that* something was.

- ⊗ It won't help if someone saw you type the real passphrase, or has malware on your machine.
- ⊗ It won't help if the real key is still unlocked. A command run while `vault unlock` holds it uses the real key, whatever passphrase you're asked to give. Run `vault lock` first.
- ⊗ It doesn't stop anyone deleting the raw files. It hides; it doesn't guard.
- ⊗ It isn't legal advice. Some places treat misleading an official as an offence in its own right. Know where you are.

Pocket this

- ☐ The decoy is set at `genkey` time only.
- ☐ Decoy and real are two separate worlds in one folder. Neither can see the other.
- ☐ Filler, identical entry sizes and a two-slot key file make the hiding work.
- ☐ Keep the decoy big, believable and alive. Keep quiet about it.

Scripts and automation

environment variables, exit codes, and a few recipes

TL;DR

Set `VAULT_PASSPHRASE`, `VAULT_KEY` and `VAULT_DIR` and vault never asks anything. Output you'd pipe (`list`, `help`) goes to standard output; everything else goes to standard error. Exit code 0 means it worked.

7.1 The environment variables

VARIABLE	WHAT IT DOES
`VAULT_PASSPHRASE`	The passphrase. Skips the prompt, and any key held by `vault unlock`. With `genkey`, it's the new key's passphrase.
`VAULT_DECOY_PASSPHRASE`	`genkey` only: the decoy passphrase. If `VAULT_PASSPHRASE` is set and this isn't, no decoy is made.
`VAULT_KEY`	The key file. Default `~/.vault/key`. Same as `-key`.
`VAULT_DIR`	The vault folder. Default `~/.vault/store`. Same as `-vault`.



With the variable set, no prompts. A failed command exits with status 1.

Here be dragons

Don't type a passphrase straight into a command line. It lands in your shell history. Read it without echo instead:

```
read -rs VAULT_PASSPHRASE && export VAULT_PASSPHRASE
```

And remember that other programs running as you can read your environment. For everyday use, just let vault prompt you.

7.2 Exit codes and output

CODE	MEANING
0	It worked.
1	Something went wrong. The reason is on standard error, starting `vault:`.
2	You ran `vault` with no command at all. The usage text is on standard error.

`list` and `help` write to standard output. Progress lines like `added …`, `replaced …` and every error go to standard error. So `vault list > files.txt` gives you a clean file list.

> **Squirrel!** *(a tangent; skip it if you like)*
> The `list` columns are fixed-width: size in 10 characters, two spaces, the date and time in 16, two spaces, then the name. The name therefore always starts at character 31, which is why `cut -c31-` works even for names with spaces in.

7.3 Recipes

Nightly: stash a folder

```
#!/bin/sh
# ~/bin/stash-documents: re-adds everything in ~/Documents/private
export VAULT_PASSPHRASE="$(cat ~/.config/vault-pass)"   # chmod 600 this file!
vault add ~/Documents/private archive || exit 1
```

Then run it from cron, say at 2am: `0 2 * * * $HOME/bin/stash-documents`. Re-adding replaces each file, so it always holds the latest copy.

> **Heads up**
> Keeping the passphrase in a file trades safety for convenience. Anyone who can read that file and the key can open the vault. Only do it on a machine you trust completely, and never for the decoy.

Get everything back out

```
#!/bin/sh
# restore the whole vault into ./restore, keeping its folders
vault list | cut -c31- | while IFS= read -r name; do
  mkdir -p "restore/$(dirname "$name")"
  vault get "$name" -o "restore/$name"
done
```

PowerShell on Windows

```
PS> $env:VAULT_DIR = ”D:\vault”
PS> vault add .\report.docx
PS> vault list
```

Pocket this

- ☐ `VAULT_PASSPHRASE`, `VAULT_KEY`, `VAULT_DIR`: no prompts, no flags.
- ☐ Exit 0 is success, 1 is an error, 2 is “no command”.
- ☐ `list` goes to stdout, so it pipes cleanly. Names start at column 31.
- ☐ Keep passphrases out of your shell history.

Locking, syncing and backups

two terminals, two machines, and keeping copies

TL;DR

vault locks itself: one writer at a time, and readers wait for writers. You don't have to do anything. Syncing the folder between machines works if only one machine changes it at a time. Back up the folder and the key, and you're covered.

8.1 Locking is automatic

Commands that change the vault (`add`, `migrate`, `mirror`, `absorb`, `mv`, `rm`) take an **exclusive** lock. Commands that only read (`list`, `get`) take a **shared** one, so several can read at once. If a command has to wait, it says so, then carries on by itself:



The second writer waits politely for the first to finish

The lock is taken *after* you've typed your passphrase, so a colleague who's slow at typing never blocks you.

Why though?

What if vault crashes while holding the lock? On Linux, macOS, the BSDs and Windows, the operating system drops the lock the moment the process dies. Nothing to clean up. Older Go builds of vault use the very same locks, so mixing versions is safe. If one's ever left behind, vault tells you after a minute exactly which file to delete.

8.2 Syncing between machines

A vault is just a folder, so Syncthing, Dropbox, iCloud Drive, OneDrive, `rsync` or a USB stick will all carry it between computers. Copy your key file to each machine too, over something private.

Heads up

Locks don't travel through sync services. Two machines writing at the same time won't wait for each other. The rule: **one machine writes at a time, and let the sync finish before you write on the other one.**

If you do slip up, it's rarely a disaster:

- Sync "conflicted copy" files have names vault doesn't recognise, so it simply ignores them.
- If both machines added the same name, `list` shows the newest, and the next `add` or `rm` of that name tidies up the other.

Network file systems are fine too. NFS honours vault's locks on Linux and the BSDs, and Windows file shares honour them from Windows.

8.3 Backups

Two things to back up: the **vault folder** and the **key file**. Ideally in different places. Any backup tool works, because it's all ordinary files.

> **Quick win**
> vault never edits a file in place. Every entry is written once and later deleted, never changed. That makes incremental backups (`rsync`, restic, borg, Time Machine) fast and tidy: each run only copies what's new.

```
$ rsync -a --delete ~/.vault/store/ /media/backup/vault-store/
```

> **Heads up**
> A backup of the vault is only as good as your ability to open it. Test it: point `-vault` at the backup copy and run `vault list` now and then.

Done when `vault list -vault /media/backup/vault-store` (or wherever your copy is) shows the same files as the original.

> **Pocket this**
> - ☐ Locking is automatic. Writers take turns; readers share.
> - ☐ Sync across machines freely, but write on one machine at a time.
> - ☐ Back up the folder and the key, separately. Test the backup.

Under the hood

the cryptography, the file layout, and exactly what a thief learns

TL;DR

Each file gets its own random 256-bit key, wrapped with a hybrid of FrodoKEM and ML-KEM, and is encrypted in authenticated 64 KiB chunks. Catalogue entries are padded to one fixed size, file contents are cut into 4 MiB pieces with the last padded to a power-of-two bucket, and every timestamp on disk is set to 1970. An attacker learns roughly how much you store. They can delete or roll back entries, but not read or forge them.

9.1 The cryptography

JOB	HOW
Protecting your key	Argon2id (3 passes, 64 MiB, 4 lanes) turns the passphrase into a key that seals your identity with XChaCha20-Poly1305.
Your identity	A 32-byte seed from which the FrodoKEM-640 and ML-KEM-768 key pairs are derived.
Per-file key	256 random bits, fresh for every single entry.
Wrapping that key	FrodoKEM-640 *and* ML-KEM-768 encapsulations, combined with HKDF-SHA-256 into one key-encryption key, then ChaCha20-Poly1305.
Header integrity	HMAC-SHA-256 under the per-file key.
The contents	ChaCha20-Poly1305 in a STREAM construction: 64 KiB chunks, each authenticated, with the last one marked so truncation is caught.

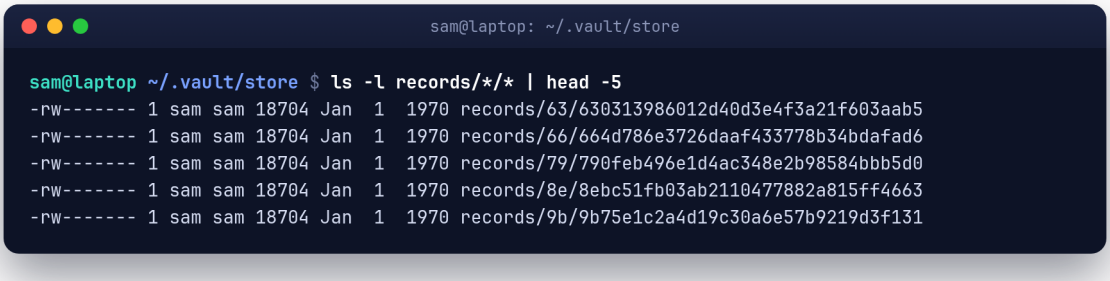
Why though?

Why two post-quantum algorithms? ML-KEM is the new standard, efficient and heavily analysed. FrodoKEM is slower and bulkier, but rests on older, more conservative maths. Combining them means a breakthrough against either one alone isn't enough.

9.2 What's in the folder

```
~/.vault/store/
├── .vault                marker: "VAULT 2"
├── .lock                 lock file, always empty
├── records/0f/           one encrypted catalogue entry per file
│   └── 0f1170a6…         every one exactly 18,704 bytes
└── objects/3e/           file contents, in 4 MiB pieces
    └── 3e78275e…         64 KiB, 128 KiB… 4 MiB, never in between
```

A **record** holds one file's name, size, permissions, dates and a secret random **seed**. It's padded to 4096 bytes before encryption, so every record is the same size whatever the name. The file's contents are cut into **objects** of just under 4 MiB each. Every object starts with its own ID, the last one is padded with random bytes up to the next power of two (at least 64 KiB), and each is encrypted separately. The objects' names are worked out from the seed, so only someone who can open the record can tell which objects belong together.



Random names, identical sizes, and all "modified" at the dawn of Unix time

9.3 Why big files are cut up

With one object per file, a 300 MB video would sit in the folder as a single 512 MiB blob. Everyone could see there's one large file, roughly how large, and, since filler is never that big, that it can't be filler. Cut into pieces, the same video is 72 objects of exactly the same size as the largest filler and as every other big file's pieces. Nothing on disk says which pieces belong together, or where one file ends and the next begins.

It's not magic, and some padding and fluff is still doing the work:

- **Small files are still bucketed.** A file under 4 MiB is one object padded to 64 KiB, 128 KiB, 256 KiB and so on, exactly as before. The last piece of a big file is padded the same way, so no file wastes more than 4 MiB, where a single padded blob could waste nearly half its size.
- **Filler is still the cover story.** The random entries `init` writes are the reason "objects I can't open" is unremarkable. By default there are only a few tens of megabytes of them, enough to hide a few files, not gigabytes. `vault init -filler SIZE` writes more, shaped like a real collection of files, at the cost of that much disk.
- **Totals still show.** The number of objects times 4 MiB is a fair estimate of how much you store.

- **Timing still shows.** Someone who watches the folder change, or reads a sync service's upload log, sees 72 objects appear together and can guess they're one file. The pieces hide a file's size from a snapshot, not from someone watching you add it.

9.4 What an attacker learns

SOMEONE WITH A COPY OF YOUR VAULT FOLDER…	CAN THEY?
Read any file's contents	✗ no
Learn file names or folder names	✗ no
Learn exact file sizes	✗ no
Learn when a file was created or changed	✗ no
Tell real entries from decoy or filler	✗ no
Alter a file without it being detected	✗ no
Swap one file's contents for another's	✗ no
Tell which pieces make up one file, from a single copy	✗ no
Count files, and see roughly how much you store	✓ yes
See that the folder changed between two copies	✓ yes
Delete entries	✓ yes
Put back an older copy of an entry (roll back)	✓ yes

> ⚠ **Heads up**
> **Deletion and rollback aren't detectable.** vault can prove a file is exactly what you stored, but not that it's the *latest* thing you stored. If the storage isn't under your control, keep an independent backup.

9.5 What happens during `add`

1. Pick a random seed. Read your file 4 MiB at a time and encrypt each piece, the last with random padding, to a hidden temporary file in its `objects/` folder. Several pieces are written at once.
2. Give each piece the name the seed dictates. They're now safely stored, but nothing refers to them.
3. Encrypt a record holding the seed, the same way, into `records/`.
4. Only then, delete any older version of the same name.

Interrupt it anywhere and the worst case is some orphaned encrypted objects that nothing points to. You never end up with a record pointing at nothing, or without either version.

> 🐾 **Squirrel!** *(a tangent; skip it if you like)*
> vault's crypto core comes from *Fury*, a larger post-quantum encryption toolkit. Peek inside any entry and the first line still reads `fury-encryption.org/v1`. That's the format header, kept on purpose: changing working, well-tested cryptographic formats for aesthetic reasons is how bugs get in.

Pocket this

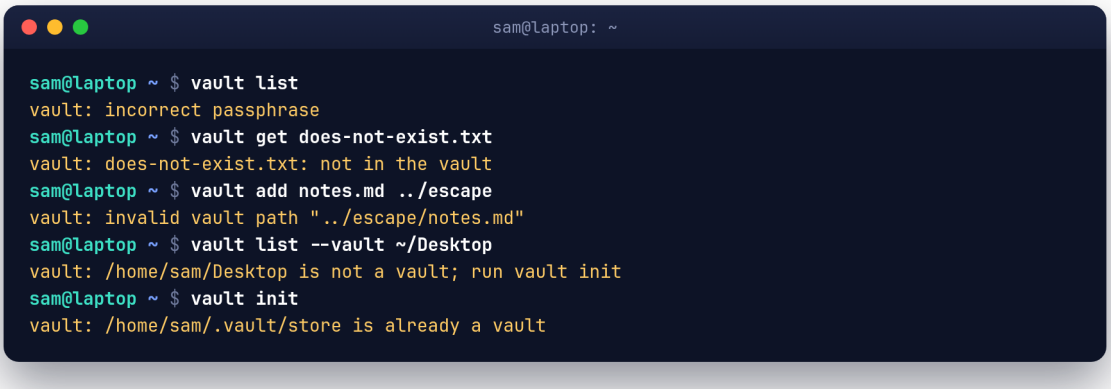
- ☐ Hybrid post-quantum key wrapping, per-file keys, authenticated streaming encryption.
- ☐ Fixed-size records, 4 MiB pieces, power-of-two tails, 1970 timestamps: metadata stays private.
- ☐ Pieces hide a file's size in any one copy of the folder, not from someone watching it change.
- ☐ Tampering is caught. Deletion and rollback are not, so keep backups.

When things go sideways

every message vault can show you, and what to do about it

TL;DR

Almost every error is one of four things: wrong passphrase, wrong `-key` or `-vault`, a name typo, or a file already in the way. Check the table. Your data is still fine.



The usual suspects

10.1 The big table

MESSAGE	WHAT IT MEANS, AND THE FIX
`incorrect passphrase`	Typo, Caps Lock, or the wrong key file. Check `-key` and `VAULT_KEY`.
`not a vault key file`	`-key` points at something that isn't a vault key, or the key file is damaged. Use your backup.
`open …/key: no such file or directory`	No key there yet. Run `vault genkey`, or point `VAULT_KEY` at yours.
`… is not a vault; run vault init`	Wrong `-vault` or `VAULT_DIR`, or you haven't run `init` yet.
`… is already a vault`	You ran `init` on an existing vault. Nothing was changed.
`NAME: not in the vault`	Check the exact name with `vault list`: it's case-sensitive and includes the folder. Or you typed the other passphrase.
`… already exists`	`get`: the output file exists, so use `-o`. `mv`, `cp`: the new name is taken. `genkey`: you already have a key.
`invalid vault path "…"`	Names can't start with `/` or `../`, end in `/`, or contain backslashes or control characters.
`…: not a regular file of at most 1 TiB`	You tried to add a device, a socket or a gigantic file.
`… changed while being added`	The file was being written while vault read it. Try again once it's finished.
`… is a folder; use -r`	`cp` copies a folder only with `-r`.
`cannot put … inside itself`	You tried to move or copy a folder into itself.
`refusing to remove the whole vault`	`rm -r /` (or an empty folder name). Name the folder you mean.
`rm cancelled; nothing changed`	You didn't answer `y` to `rm -r`, or there was no terminal to ask on. `-f` skips the question.
`skipping …: not a regular file`	Adding a folder: a symlink or special file was skipped. Everything else went in.

Passphrases, the unlock helper, and damage

MESSAGE	WHAT IT MEANS, AND THE FIX
`passphrases do not match`	The two entries differed. Nothing was created; run it again.
`passphrase must not be empty`	Blank passphrases aren't allowed, for good reason.
`decoy passphrase must differ from the passphrase`	The decoy can't be the same as the real one.
`VAULT_PASSPHRASE is set but empty`	Unset it, or give it a value.
`could not read passphrase (set VAULT_PASSPHRASE…)`	No terminal to ask on, typically cron or CI. Set the variable.
`… is not a private directory of this user`	The folder for `unlock`'s socket belongs to someone else or is open to others. Someone may be up to something; check who made it. Commands still work, asking for the passphrase.
`agent did not start: …`	`unlock` couldn't start its helper. The reason follows; commands still work, asking for the passphrase.
`nothing was unlocked`	`vault lock` found nothing to lock. Not an error.
`waiting for another vault process to finish`	Not an error. Another command is writing; this one continues when it's done.
`skipping unreadable record …`	A catalogue entry is damaged. Everything else still works; restore `records/` from backup to recover that file.
`object does not match its catalogue record`	The encrypted contents were swapped or damaged. Nothing was written. Restore `objects/` from backup.
`object size does not match its catalogue record`	Same as above: damage or tampering, caught before any output appeared.
`cannot find home directory; set VAULT_KEY`	Unusual environments only. Set `VAULT_KEY` and `VAULT_DIR`.
`unknown command / wrong number of arguments`	A typo in the command itself. The usage text follows.

Files, the lock and downloads

MESSAGE	WHAT IT MEANS, AND THE FIX
`cannot lock the vault: …`	The lock file couldn't be opened or locked; the reason follows. Check the vault folder's permissions.
`… changed while being added`	The file was being written while vault read it. Nothing was stored; try again when it's finished.
`… changed after it was stored; not deleting it`	`migrate` stored the file, but it changed before it could be deleted. The original is kept; the vault has the version it read.
`… already exists`	`get` never overwrites, not even a symlink. Pick another name with `-o`.
`… refusing to extract through it`	`get -r` found a symlink (or a file) where a folder should be inside the output folder, and stopped rather than follow it.
`cannot lock agent memory …, cannot make the agent undumpable …`	`unlock`'s helper couldn't protect the key in memory, so it refused it. Commands still work, asking for the passphrase.
`vault unlock is not supported on this platform…`	Windows: every command asks for the passphrase.
`download exceeds … bytes`	A download passed its size limit. Raise it with `-max-bytes` if you meant it.
`redirected to another host: …`	`mirror` only follows redirects within the site it is copying.
`mirror incomplete: stored …, skipped …`	Some pages failed; the reasons were printed as they happened. Exit status 3.

10.2 "My files have vanished!"

Deep breath. Nine times out of ten, one of these:

- ☑ **You typed the other passphrase.** The decoy and real sides are separate worlds. Try the other one.
- ☑ **You're looking at a different vault.** Check `-vault` and `VAULT_DIR`. Is this the vault you think it is?
- ☑ **You're looking in a folder.** `vault list letters` only shows what's under `letters/`.

10.3 Interrupted in the middle of something?

Power cut, closed laptop, Ctrl + C. It's fine. vault writes new data first and removes old data last, so an interruption leaves the vault consistent. At worst there's an orphaned encrypted object, or a temporary file starting `.pending-`. Both are harmless and both are safe to delete.

> **Here be dragons**
> **The one thing that can't be fixed:** a lost key file or a forgotten passphrase. Nothing in this chapter helps with that. Back up your key.

Pocket this

- ☐ Most errors mean a wrong passphrase, key, vault or name.
- ☐ “Missing” files usually mean the other passphrase or another vault.
- ☐ Interruptions are safe. Lost keys are not.

Cheat sheet

the whole tool at a glance. print it, pin it up

`vault genkey`
Make a key. Asks for a passphrase and an optional decoy.

`vault init`
Make an empty vault, with filler. `-filler 2G` for more.

`vault add FILE|DIR|URL [FOLDER]`
Encrypt in, keeping the name. Replaces an existing name.

`vault migrate FILE|DIR [FOLDER]`
Like add, then delete the originals. Asks before replacing.

`vault mirror URL [FOLDER]`
Crawl a website in, links rewritten to work offline.

`vault absorb [FOLDER] < TAR`
Every file in a tar or .tar.gz from standard input.

`vault ls [FOLDER]`
One folder: sub-folders (ending in /) and files.

`vault list [FOLDER]`
Every file under it, full names. Pipes cleanly.

`vault get [-r] NAME [-o OUT]`
Decrypt out, or a folder with -r. Never overwrites. Verifies first.

`vault mv OLD NEW`
Rename or move a file or folder. Instant.

`vault cp [-r] OLD NEW`
Copy a file, or a folder with -r. Instant, no extra space.

`vault rm [-r] [-f] NAME`
Delete a file, or a folder with -r. Frees the space. No undo.

`vault unlock [-for 5m] [-idle]`
Remember the key, so commands stop asking. Fixed deadline, or since last use.

`vault lock`
Forget the remembered key now.

`vault help`
Show the command summary.

`man vault`
A manual page for vault and for every command.

Options and variables

FLAG	VARIABLE	DEFAULT
`-key PATH`	`VAULT_KEY`	`~/.vault/key`
`-vault DIR`	`VAULT_DIR`	`~/.vault/store`
(prompt)	`VAULT_PASSPHRASE`	asked on the terminal
(genkey prompt)	`VAULT_DECOY_PASSPHRASE`	asked; empty means none
`-o OUT`	—	the file's own name, here
`-filler SIZE`	—	a few tens of MB, for `init`
`-yes`	—	ask, for `migrate`
`-r`	—	`rm`, `get`, `cp`: one file; `ls`: one level
`-for DURATION`	—	`5m`, for `unlock`
`-idle`	—	`unlock`: count from the unlock, not the last use
`-f`	—	`rm`: ask before `-r`, fail if missing

Numbers worth knowing

THING	VALUE
Largest file	1 TiB
Longest stored name	1024 bytes
Smallest object on disk	64 KiB
Largest object on disk	4 MiB
Folders under `records/` and `objects/`	256 each
Filler entries in a new vault	8 to 32, or more with `-filler`
Key file size	319 bytes
Exit codes	0 ok, 1 error, 2 no command

APPENDIX A

Glossary

Argon2id
A deliberately slow, memory-hard function that turns your passphrase into a key, making guessing expensive.

Catalogue
The encrypted list of what's in your vault, stored as one record per file.

Decoy passphrase
A second passphrase that opens a separate, harmless set of files.

Entry
One record plus its objects. Real, decoy and filler entries all look identical on disk.

Filler
Entries sealed to a key that was immediately destroyed. Nobody can open them. They make the decoy plausible.

FrodoKEM, ML-KEM
Post-quantum key-encapsulation algorithms. vault uses both at once.

Key file
The small file holding your identity, encrypted under your passphrase (and decoy).

Object
One encrypted piece of a file, 4 MiB, or padded to a smaller power of two if it's the last piece.

Post-quantum
Designed to stay secure against attackers with large quantum computers.

Record
The encrypted catalogue entry for one file: its name, size, dates and the seed its objects' names come from.

Vault
The folder where your encrypted files live.

APPENDIX B

About this guide

This guide lives in `docs/guide` and builds itself.

```
$ ./build-pdf.sh    # makes vault-guide.pdf (downloads Tectonic if needed)
$ ./shots.py        # re-shoots every screenshot from the real vault binary
```

Every screenshot is real output from the `vault` program built from the same source code, run in a throwaway home folder. The only touch-ups: the home folder is shown as `/home/sam`, some lines are highlighted in amber so they're easier to spot, and the two "waiting for your passphrase" shots are frozen at that moment.

That's it. Go encrypt something.
(And back up your key.)

---

uncomplex.

Complicated things. Thoughtfully simplified.