



THE FIELD GUIDE

vault

Post-quantum file encryption.
With a secret trapdoor.

 FrodoKEM + ML-KEM

 decoy passphrase

 one binary, 43 platforms

for humans who skim

VERSION 0.4.1 • 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

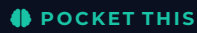
Something that can lose you data. These are rare. Read every one of them.

🐾 SQUIRREL! *(a tangent. skip it if you like)*

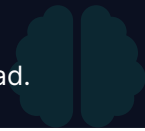
A tangent. Interesting, possibly, but you can skip it and lose nothing.

💡 WHY THOUGH?

The reasoning behind a design decision, for the curious.



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	ii
1 Meet vault	1
<i>what it is, what it isn't, and a 30-second mental model</i>	
1.1 What it actually does	1
1.2 The three things that make it different	1
1.3 What vault is <i>not</i>	2
2 Install it	3
<i>grab a binary, check it, put it on your PATH</i>	
2.1 Pick your file	3
2.2 Check the file hasn't been tampered with	4
2.3 Put it on your PATH	4
2.4 Did it work?	5
2.5 Building it yourself	6
3 Your key	8
<i>the one file that matters most, and the passphrase that guards it</i>	
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
<i>one command, and a peek inside</i>	
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	14
	<i>add, ls, get, mv, rm: the five commands you'll actually use</i>	
5.1	Putting things in: add	14
5.2	Seeing what's inside: ls and list	17
5.3	Getting things out: get	18
5.4	Renaming and moving: mv	18
5.5	Deleting: rm	18
5.6	Flags work anywhere	20
6	The decoy passphrase	21
	<i>one key, two passphrases, two completely different vaults</i>	
6.1	The idea	21
6.2	How it looks	21
6.3	Setting it up	22
6.4	Why it holds up	22
6.5	Making it believable: the playbook	23
6.6	The honest limits	23
7	Scripts and automation	25
	<i>environment variables, exit codes, and a few recipes</i>	
7.1	The environment variables	25
7.2	Exit codes and output	26
7.3	Recipes	26
8	Locking, syncing and backups	28
	<i>two terminals, two machines, and keeping copies</i>	
8.1	Locking is automatic	28
8.2	Syncing between machines	28
8.3	Backups	29
9	Under the hood	30
	<i>the cryptography, the file layout, and exactly what a thief learns</i>	
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
	<i>every message vault can show you, and what to do about it</i>	
10.1	The big table	35
10.2	“My files have vanished!”	36
10.3	Interrupted in the middle of something?	36
11	Cheat sheet	37
	<i>the whole tool at a glance. print it, pin it up</i>	
A	Glossary	39
B	About this guide	40

Meet vault

• ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ you are here

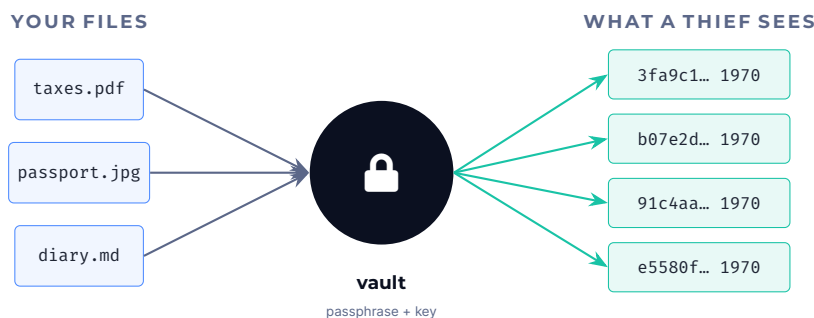
⚡ TL;DR

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

2

• • • • • • • • • • you are here

⚡ 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 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	<code>vault-windows-amd64.exe</code>
Windows on ARM	Surface Pro X, Snapdragon laptops	<code>vault-windows-arm64.exe</code>
Mac, 2020 or later	Apple silicon (M1, M2, M3...)	<code>vault-darwin-arm64</code>
Mac, older	Intel	<code>vault-darwin-amd64</code>
Linux PC or server	almost everything	<code>vault-linux-amd64</code>
Linux on ARM64	Pi 4/5 (64-bit OS), Graviton, Ampere	<code>vault-linux-arm64</code>
Raspberry Pi, 32-bit	Pi 2, 3, 4 on a 32-bit OS	<code>vault-linux-armv7</code>
Raspberry Pi Zero / 1	the tiny older ones	<code>vault-linux-armv6</code>
Android	in Termux	<code>vault-android-arm64</code>
FreeBSD, OpenBSD, NetBSD	amd64, arm64, 386, arm...	<code>vault-freebsd-amd64</code> etc.
Something exotic	RISC-V, POWER, s390x, MIPS, LoongArch	see dist/

🐿 SQUIRREL! (a tangent. skip it if you like)

The full release covers 43 platform builds, including AIX, Solaris, illumos, Plan 9 and even WebAssembly. Will you ever need `vault-plan9-arm`? 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.

2.2 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.

2.3 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-android-arm64
$ chmod +x vault-android-arm64 && mv vault-android-arm64 $PREFIX/bin/vault
```

2.4 Did it work?

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

```

sam@laptop: ~
sam@laptop ~ $ vault help
Usage:
  vault genkey                create a new passphrase-protected key
  vault init [--filler SIZE]  create a new, empty vault, with at least
                              SIZE of filler (e.g. 500M, 2G)
  vault add FILE|DIR|URL [FOLDER] encrypt a file, a directory (recursively)
                              or a downloaded URL into the vault,
                              optionally under a folder inside the vault
  vault migrate FILE|DIR [FOLDER] like add, then delete each original once
  [--yes]                    it is stored; asks before overwriting
                              anything already in the vault, unless
                              --yes is given
  vault mirror URL [FOLDER]   crawl a website (same host only, one request
                              at a time) into FOLDER/HOST, rewriting its
                              links to relative paths
  vault absorb [FOLDER]       read a tar archive (optionally gzipped) from
                              standard input and add every file in it,
                              optionally under a folder inside the vault
  vault ls [FOLDER]           list one folder: its files, and its
                              sub-folders (ending in /) with their
                              total size; -r lists everything under it
  vault list [FOLDER]         list every stored file under FOLDER,
                              with full names
  vault get NAME [-o OUTPUT]  decrypt a stored file (default: its base name)
  vault mv OLD NEW            rename a stored file
  vault rm NAME               delete a stored file
  vault rm -r FOLDER [-f]     delete a folder and everything under it;
                              asks first unless -f (or -rf) is given

Options (any command):
  --key PATH    key file (default $VAULT_KEY, else ~/.vault/key)
  --vault DIR   vault directory (default $VAULT_DIR, else ~/.vault/store)

The passphrase is read from the terminal, or from $VAULT_PASSPHRASE.

Decoy passphrase: genkey offers a second, optional passphrase
($VAULT_DECOY_PASSPHRASE when scripted). It unlocks a separate key, so
every command run with it sees and changes only the decoy files, and the
real files stay invisible. The key file does not reveal whether a decoy
exists, and init fills each vault with unreadable filler, so hidden
files look like filler to anyone holding only the decoy passphrase.

```

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

2.5 Building it yourself

Prefer to build from source? You need Go 1.26.6 or newer. Then:

```

$ go build .      # just your platform, produces ./vault
$ ./build.sh     # every platform, into ./dist, with SHA256SUMS

```

`build.sh` cross-compiles every target Go supports without C libraries. 32-bit ARM Linux gets two builds, armv6 and armv7. Want just a few? Set `TARGETS`. Want them somewhere else? Set `OUT`.

```
sam@laptop: ~/src/vault

sam@laptop ~/src/vault $ export TARGETS="linux/amd64 linux/arm darwin/arm64 windows/amd64"
sam@laptop ~/src/vault $ OUT=/tmp/release ./build.sh
built vault-linux-amd64
built vault-linux-armv6
built vault-windows-amd64.exe
built vault-darwin-arm64
built vault-linux-armv7
5 binaries in /tmp/release
sam@laptop ~/src/vault $ cd /tmp/release && sha256sum -c SHA256SUMS
vault-darwin-arm64: OK
vault-linux-amd64: OK
vault-linux-armv6: OK
vault-linux-armv7: OK
vault-windows-amd64.exe: OK
```

📷 A partial build, followed by checking every binary against SHA256SUMS

💡 WHY THOUGH?

The binaries are *static* and *reproducible*: no C code, no system libraries, file paths stripped, build IDs zeroed. Two people building the same commit get byte-identical files, 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 all 43 platform binaries from source.



Your key

3

• • • ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ you are here

⚡ 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.

```
sam@laptop: ~  
sam@laptop ~ $ vault genkey  
Enter passphrase: █
```

📷 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.

```
sam@laptop: ~  
sam@laptop ~ $ vault genkey  
Enter decoy passphrase (empty for none): █
```

📷 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:

```
sam@laptop: ~  
sam@laptop ~ $ vault genkey  
key written to /home/sam/.vault/key  
sam@laptop ~ $ ls -l ~/.vault/  
total 4  
-rw----- 1 sam sam 319 Oct  2 18:13 key
```

📷 Your key: 319 bytes, readable only by you

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.



4.1 Create it

```
sam@laptop: ~  
  
sam@laptop ~$ vault init  
vault created at /home/sam/.vault/store  
sam@laptop ~$ ls -A ~/.vault/store/  
.vault  
objects  
records  
sam@laptop ~$ find ~/.vault/store/records -type f | wc -l  
20
```

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?

11

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

5

• • • • • ○ ○ ○ ○ ○ you are here

⚡ TL;DR

`add` puts files in (optionally into a folder), `ls` shows them a folder at a time, `get` takes one back out, `mv` renames, `rm` deletes (a whole folder with `-r`). Each asks for your passphrase. 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.

```
sam@laptop: ~  
  
sam@laptop ~ $ vault add Documents/tax-return-2025.pdf  
added tax-return-2025.pdf  
sam@laptop ~ $ vault add Documents/passport-scan.jpg id  
added id/passport-scan.jpg  
sam@laptop ~ $ vault add Documents/letters  
added letters/bank.txt  
added letters/landlord.txt  
sam@laptop ~ $ vault add notes.md  
added notes.md
```

📷 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

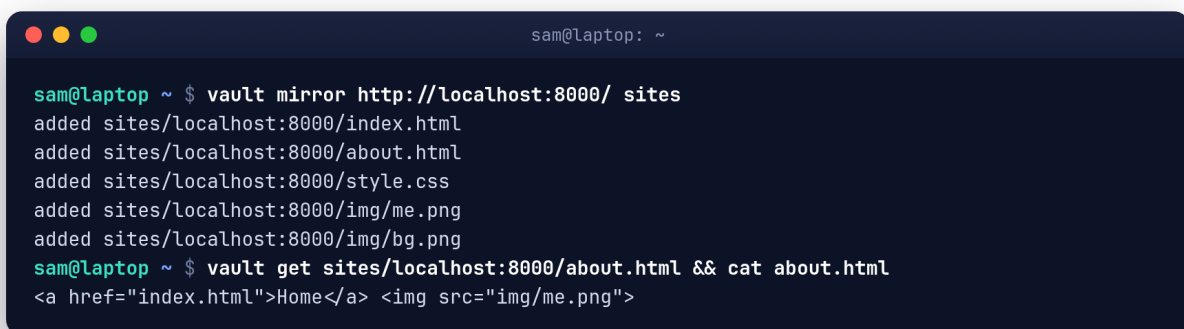
Adding a name that already exists **replaces** it, and vault tells you so. The old version is deleted *after* the new one is safely stored, so a crash halfway through never leaves you with neither.

■ 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.



```
sam@laptop: ~  
  
sam@laptop ~ $ vault mirror http://localhost:8000/ sites  
added sites/localhost:8000/index.html  
added sites/localhost:8000/about.html  
added sites/localhost:8000/style.css  
added sites/localhost:8000/img/me.png  
added sites/localhost:8000/img/bg.png  
sam@laptop ~ $ vault get sites/localhost:8000/about.html && cat about.html  
<a href="index.html">Home</a> 
```

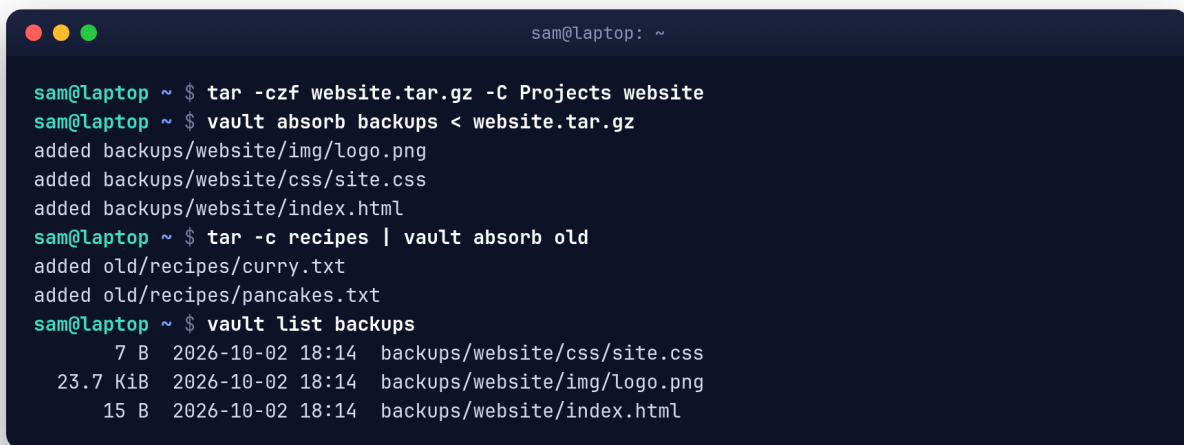
📷 Mirroring a small site. The links in each page now point at the copy.

■ 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.



```

sam@laptop: ~
sam@laptop ~ $ tar -czf website.tar.gz -C Projects website
sam@laptop ~ $ vault absorb backups < website.tar.gz
added backups/website/img/logo.png
added backups/website/css/site.css
added backups/website/index.html
sam@laptop ~ $ tar -c recipes | vault absorb old
added old/recipes/curry.txt
added old/recipes/pancakes.txt
sam@laptop ~ $ vault list backups
 7 B  2026-10-02 18:14  backups/website/css/site.css
23.7 KiB  2026-10-02 18:14  backups/website/img/logo.png
15 B  2026-10-02 18:14  backups/website/index.html

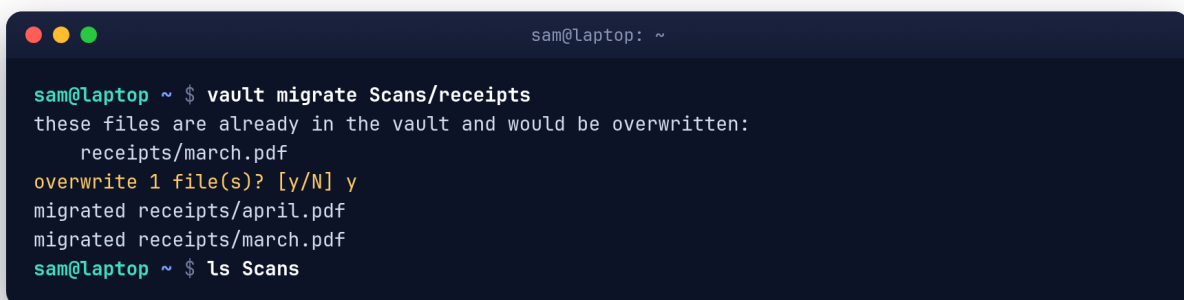
```

📷 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.



```

sam@laptop: ~
sam@laptop ~ $ vault migrate Scans/receipts
these files are already in the vault and would be overwritten:
  receipts/march.pdf
overwrite 1 file(s)? [y/N] y
migrated receipts/april.pdf
migrated receipts/march.pdf
sam@laptop ~ $ ls Scans

```

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

■ What gets skipped

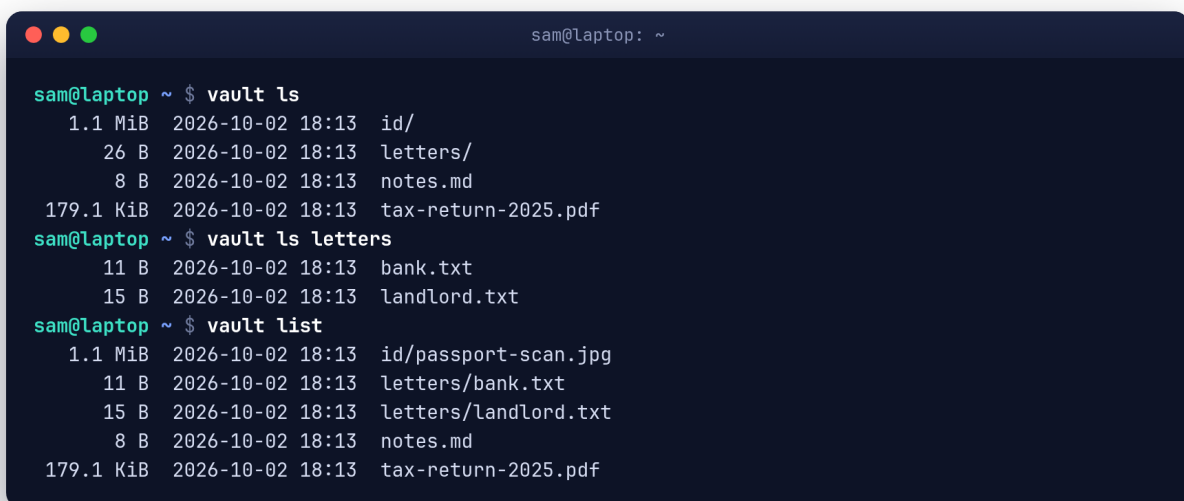
When adding a folder, anything that isn't a plain file, such as symbolic links, sockets or devices, is skipped with a note. So are the vault and key themselves, should they live inside the folder, so `migrate` can never eat them. Files larger than 1 TiB are refused. (If you have a single 1 TiB file,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



```
sam@laptop: ~
sam@laptop ~ $ vault ls
 1.1 MiB  2026-10-02 18:13  id/
   26 B   2026-10-02 18:13  letters/
    8 B   2026-10-02 18:13  notes.md
179.1 KiB 2026-10-02 18:13  tax-return-2025.pdf
sam@laptop ~ $ vault ls letters
 11 B  2026-10-02 18:13  bank.txt
 15 B  2026-10-02 18:13  landlord.txt
sam@laptop ~ $ vault list
 1.1 MiB  2026-10-02 18:13  id/passport-scan.jpg
 11 B     2026-10-02 18:13  letters/bank.txt
 15 B     2026-10-02 18:13  letters/landlord.txt
 8 B      2026-10-02 18:13  notes.md
179.1 KiB 2026-10-02 18:13  tax-return-2025.pdf
```

📷 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
```

```
sam@laptop: ~
sam@laptop ~ $ mkdir -p restored && cd restored && vault get id/passport-scan.jpg
sam@laptop ~ $ vault get tax-return-2025.pdf -o restored/tax.pdf
sam@laptop ~ $ vault get notes.md
vault: notes.md already exists
```

📷 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 and moving: `mv`

```
$ vault mv OLD NEW
```

`NEW` is the *full* new name, folder included. So to move `notes.md` into `ideas`, it's `vault mv notes.md ideas/notes.md`, not ... `ideas/`. It won't overwrite an existing name, and it doesn't re-encrypt the file, so it's instant however big the file is.

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.

```
sam@laptop: ~  
  
sam@laptop ~ $ echo '# more ideas' >> notes.md && vault add notes.md  
replaced notes.md  
sam@laptop ~ $ vault mv notes.md ideas/notes.md  
sam@laptop ~ $ vault rm letters/bank.txt  
removed letters/bank.txt  
sam@laptop ~ $ vault list  
1.1 MiB 2026-10-02 18:13 id/passport-scan.jpg  
21 B 2026-10-02 18:13 ideas/notes.md  
15 B 2026-10-02 18:13 letters/landlord.txt  
179.1 KiB 2026-10-02 18:13 tax-return-2025.pdf
```

📷 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.

```

sam@laptop: ~
sam@laptop ~ $ vault ls
 1.1 MiB  2026-10-02 18:13  id/
    21 B  2026-10-02 18:13  ideas/
    15 B  2026-10-02 18:13  letters/
 97.6 KiB  2026-10-02 18:14  receipts/
179.1 KiB  2026-10-02 18:13  tax-return-2025.pdf
sam@laptop ~ $ vault rm -r receipts
these files will be deleted, with no undo:
  receipts/april.pdf
  receipts/march.pdf
delete 2 file(s), 97.6 KiB? [y/N] y
removed receipts/april.pdf
removed receipts/march.pdf
removed 2 file(s), 97.6 KiB
sam@laptop ~ $ vault ls
 1.1 MiB  2026-10-02 18:13  id/
    21 B  2026-10-02 18:13  ideas/
    15 B  2026-10-02 18:13  letters/
179.1 KiB  2026-10-02 18:13  tax-return-2025.pdf

```

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

5.6 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 NAME [-o OUT], mv OLD NEW, rm NAME, rm -r FOLDER`.
- ✓ `get` never overwrites and never publishes a half-checked file.
- ✓ `rm` has no undo.

The decoy passphrase

6

• • • • • ○ ○ ○ ○ ○ you are here

⚡ 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 2026-10-02 18:13 id/passport-scan.jpg
21 B 2026-10-02 18:13 ideas/notes.md
15 B 2026-10-02 18:13 letters/landlord.txt
179.1 KiB 2026-10-02 18:13 tax-return-2025.pdf
  
```

```

decoy passphrase
sam@laptop ~ $ vault list
7 B 2026-10-02 18:13 recipes/curry.txt
18 B 2026-10-02 18:13 recipes/pancakes.txt
21 B 2026-10-02 18:13 shopping/notes.md
  
```

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

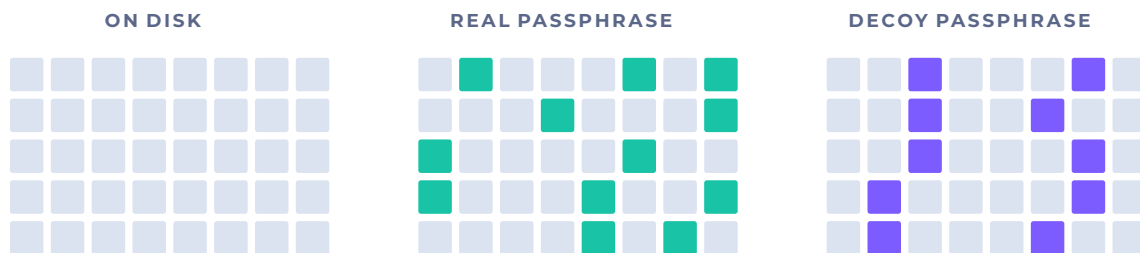
Nothing on the right hints that the left exists. Ask the decoy for a real file and you get the same message you'd get for a typo. The folder's entry count doesn't give it away either, because filler is always there:

```

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
27

```

📷 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_DECLOY_PASSPHRASE` alongside `VAULT_PASSPHRASE` before `genkey`. Chapter 7 has more on scripting.

⚠️ 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 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

7

• • • • • • • • • • you are here

⚡ 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
<code>VAULT_PASSPHRASE</code>	The passphrase. Skips the prompt. With <code>genkey</code> , it's the new key's passphrase.
<code>VAULT_DECOY_PASSPHRASE</code>	<code>genkey</code> only: the decoy passphrase. If <code>VAULT_PASSPHRASE</code> is set and this isn't, no decoy is made.
<code>VAULT_KEY</code>	The key file. Default <code>~/.vault/key</code> . Same as <code>-key</code> .
<code>VAULT_DIR</code>	The vault folder. Default <code>~/.vault/store</code> . Same as <code>-vault</code> .

```

sam@laptop: ~
sam@laptop ~ $ export VAULT_PASSPHRASE=...
sam@laptop ~ $ vault list ideas
21 B  2026-10-02 18:13  ideas/notes.md
sam@laptop ~ $ vault get nope 2>/dev/null; echo "exit code: $?"
exit code: 1

```

📷 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 every-

day 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 <code>vault:.</code>
2	You ran <code>vault</code> 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

8

• • • • • • • • • • you are here

⚡ 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:

```
terminal 1
sam@laptop ~ $ vault add big-video.mkv
added big-video.mkv
```

```
terminal 2
sam@laptop ~ $ vault add recipes/curry.txt
added curry.txt
```

📺 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. On Plan 9 and WebAssembly, which have no file locks, vault uses a `.lock` file instead. 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.

📁 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.



9.1 The cryptography

💡 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.

```

sam@laptop: ~/.vault/store

sam@laptop ~/.vault/store $ ls -l records/*/ | head -5
-rw----- 1 sam sam 18704 Jan 1 1970 records/0c/0c5eebb18fb94ca6317f6064d22532e7
-rw----- 1 sam sam 18704 Jan 1 1970 records/21/21b18b94173c2953f43abee0e78e2d43
-rw----- 1 sam sam 18704 Jan 1 1970 records/24/24f0cb576d6c7fc913b997454cf9f828
-rw----- 1 sam sam 18704 Jan 1 1970 records/27/27e126f997376c6cbeeac45d15a9a02f
-rw----- 1 sam sam 18704 Jan 1 1970 records/32/327e4f9cdc84d8f224422e84a7af8109

```

📷 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

10

• • • • • • • • • • you are here

⚡ 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.

```
sam@laptop: ~  
  
sam@laptop ~ $ vault list  
vault: incorrect passphrase  
sam@laptop ~ $ vault get does-not-exist.txt  
vault: does-not-exist.txt: not in the vault  
sam@laptop ~ $ vault add notes.md ../escape  
vault: invalid vault path "../escape/notes.md"  
sam@laptop ~ $ vault list --vault ~/Desktop  
vault: /home/sam/Desktop is not a vault; run vault init  
sam@laptop ~ $ vault init  
vault: /home/sam/.vault/store is already a vault
```

📷 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 <code>-key</code> and <code>VAULT_KEY</code> .
not a vault key file	<code>-key</code> 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 <code>vault genkey</code> , or point <code>VAULT_KEY</code> at yours.
... is not a vault; run vault init	Wrong <code>-vault</code> or <code>VAULT_DIR</code> , or you haven't run <code>init</code> yet.
... is already a vault	You ran <code>init</code> on an existing vault. Nothing was changed.
NAME: not in the vault	Check the exact name with <code>vault list</code> : it's case-sensitive and includes the folder. Or you typed the other passphrase.
... already exists	<code>get</code> : the output file exists, so use <code>-o</code> . <code>mv</code> : the new name is taken. <code>genkey</code> : you already have a key.
invalid vault path "..."	Names can't start with <code>/</code> or <code>../</code> , end in <code>/</code> , or contain backslashes or control characters.
...: not a regular file of at most 1TiB	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.
refusing to remove the whole vault	<code>rm -r /</code> (or an empty folder name). Name the folder you mean.
rm cancelled; nothing changed	You didn't answer <code>y</code> to <code>rm -r</code> , or there was no terminal to ask on. <code>-f</code> skips the question.
skipping ...: not a regular file	Adding a folder: a symlink or special file was skipped. Everything else went in.
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.
waiting for another vault process to finish	Not an error. Another command is writing; this one continues when it's done.
vault still locked after a minute...	Plan 9 and WebAssembly only. If nothing else is running, delete the file it names.
skipping unreadable record ...	A catalogue entry is damaged. Everything else still works; restore <code>records/</code> from backup to recover that file.
object does not match its catalogue record	The encrypted contents were swapped or damaged. Nothing was written. Restore <code>objects/</code> 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 <code>VAULT_KEY</code> and <code>VAULT_DIR</code> .
unknown command / wrong number of arguments	A typo 35 command itself. The usage text follows.

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

11

• • • • • • • • • • you are here

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 NAME [-o OUT]

Decrypt out. Never overwrites. Verifies first.

vault mv OLD NEW

Rename, using the full new name. Instant.

vault rm [-r] [-f] NAME

Delete a file, or a folder with `-r`. Frees the space. No undo.

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 <code>init</code>
-yes	—	ask, for <code>migrate</code>
-r	—	<code>rm</code> : one file; <code>ls</code> : one level
-f	—	<code>rm</code> : ask before <code>-r</code> , 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 <code>records/</code> and <code>objects/</code>	256 each
Filler entries in a new vault	8 to 32, or more with <code>-filler</code>
Key file size	319 bytes
Exit codes	0 ok, 1 error, 2 no command

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.

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.)