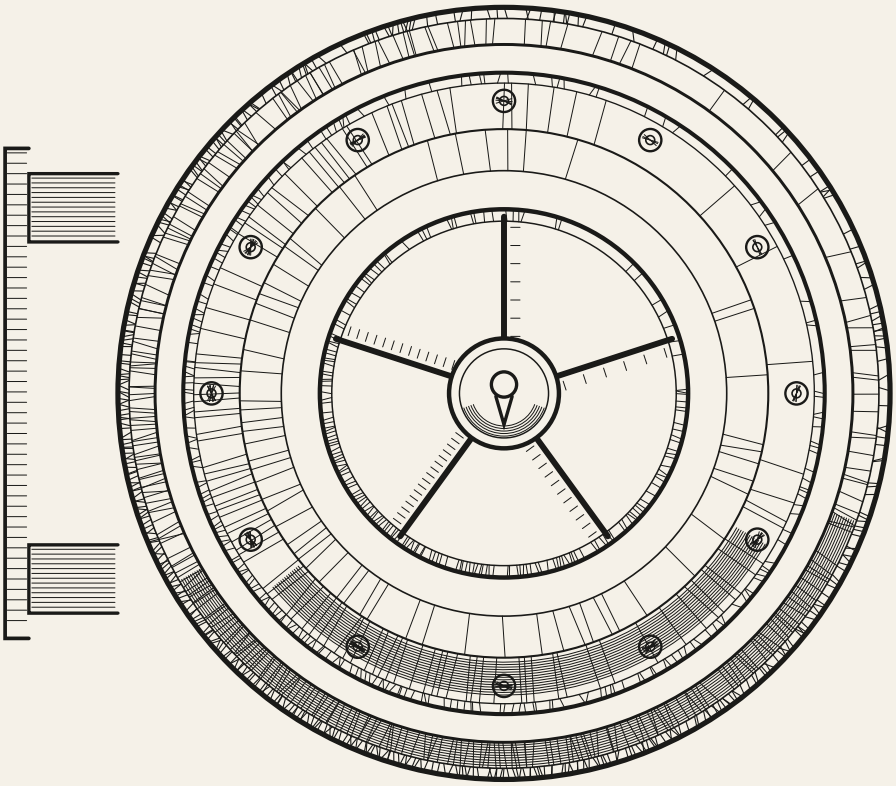
IN PRACTICE



Vault in Practice

Hands-on labs for HashiCorp Vault and OpenBao

24 chapters, 24 labs, and three that ask you to break things

Covers the Vault Associate exam objectives in full

Thomas Zachmann

BOOK ONE

Vault in Practice

A Hands-On Lab Guide to HashiCorp Vault and OpenBao

Thomas Zachmann

Contents

Vault in Practice

A Hands-On Lab Guide to HashiCorp Vault and OpenBao

With full coverage of the Vault Associate exam objectives

Thomas Zachmann

Copyright

Trademarks

Disclaimer

The information in this book is provided on an "as is" basis, without warranty of any kind, express or implied. Neither the author nor any distributor shall be liable for any damages arising directly or indirectly from the use of the information or the code examples contained herein.

The commands and configurations in this book are written for a **local laboratory environment**. Several of them deliberately weaken security — TLS verification is disabled, self-signed certificates are used, credentials appear in plain text on the command line, and in one chapter you will be asked to destroy your own data on purpose. **Do not run these commands against a production system.** Each chapter states explicitly which settings must change before anything resembling this setup goes near real data.

Software changes. Every command in this book was verified against the software versions listed in Chapter 2. Where a command has changed in recent releases, the book says so. Where behaviour depends on a version, the version is named. If something does not work as printed, check the version first.

Edition

First edition, 2026.

Acknowledgements

To my wife, and to my family, for the evenings this took.

And to the colleagues who have shaped this career — the ones who showed me how things are actually built, and the ones who, just as often and just as usefully, brought me back down to earth.

About the Author

Thomas Zachmann has spent more than twenty years building systems, and the last several of them building Kubernetes platforms for organisations where security and auditability are the requirement rather than the aspiration: on-premises, on bare metal, in air-gapped networks, and in European sovereign clouds. His work is almost entirely in regulated environments — federal

printing and identity services, public-sector IT providers, cooperative banking, and security technology vendors — under European regulatory and audit regimes.

His work is the hardening of container platforms and everything that has to hold up when an auditor asks: policy enforcement with Kyverno, identity and access with Keycloak, supply-chain evidence with Harbor, Trivy and Dependency Track — and secrets management with HashiCorp Vault, which he has been deploying and operating since 2017.

He works in the public clouds as well — AWS, Azure and Google Cloud — which is what makes the on-premises argument in these books a comparison rather than a preference. Most regulated estates are hybrid, and knowing what a hyperscaler does well is the only honest way to say where it does not fit.

He is certified as a Certified Kubernetes Security Specialist (CKS), Certified Kubernetes Administrator (CKA) and Certified Kubernetes Application Developer (CKAD), and holds cloud certifications from AWS and Google Cloud.

Those twenty years are software engineering — performance and systems programming at SAP in C and C++, database performance analysis at IBM, and Go since. That background is why the automation in his platforms is written rather than bought, and why this book asks you to type rather than to read.

This book exists because of a pattern he has watched repeatedly: teams understand Vault within an hour and take months to run it well. The gap is never conceptual. It is that nobody ever made them break a cluster and put it back together.

He is based in Hamburg and works across the DACH region.

He works as an independent consultant and takes on engagements directly. That includes implementing Vault end to end — architecture, installation, hardening, high availability, backup and restore, the integrations in Part V of this book, and the handover — as well as reviewing an installation somebody else built, and training a team on the material in these chapters.

If any of that is useful to you, get in touch. Questions about the book are welcome too, and so are corrections.

Contact: thomas@zachmann.work **Web:** thomaszachmann.de · github.com/thomaszachmann

How to Read This Book

This is a workbook. You type it into a terminal. Reading it in an armchair will teach you what Vault does; it will not teach you to run Vault, and only one of those is worth money.

What this book is not

It is not the documentation in a different binding. HashiCorp's documentation is good, it is free, and an assistant will summarise it for you in seconds — a book that competes with that has no reason to exist.

What this book does instead is take the questions that actually come up in production and answer them **against a running cluster**, with the output printed. Not "Vault seals on restart" but a restart, timed, with the HTTP 503 it returned and how long it stayed sealed. Not "rekey needs a quorum" but a rekey that fails because somebody used the wrong share, and the message it produced.

Every one of those was run before it was written down. None of it is transcribed from a manual.

And then it asks you to do it. That is the part that matters, and it is the part a reader can skip and get nothing. Every chapter has a lab you type, failures you cause on purpose, and tasks with no walkthrough. The knowledge this book is trying to give you is not the kind you can be told — it is the kind you have after you broke something at half past two in the morning and put it back.

So: keep a terminal open, run every command, and cause every failure. An afternoon of that is worth more than the whole book read through.

Why this book is shorter than it used to be

An earlier draft of this book explained things. It explained what a secrets engine is, why short-lived credentials are better than static ones, and what each flag in each command does. Then the world changed underneath it.

You have an AI assistant in the terminal next to this book. It will explain any flag, any concept and any error message, instantly, in your own words, at whatever depth you ask for. It is genuinely good at that. A chapter that competes with it is a chapter you will skim.

So this book does not compete with it. It does the two things an assistant cannot:

It knows what actually happened. Every number, every error message and every duration printed here was measured on a real machine, and the book says which. An assistant will tell you that an unsealed Vault answers requests; this book tells you that a restarted Vault answered in 1152 ms with HTTP 503 and stayed sealed for as long as nobody came with a key. The first is true. The second is what you plan around.

It makes you do it wrong on purpose. Every chapter has a lab where you break something, read the real message, and fix it. You can ask an assistant what `http: server gave HTTP response to HTTPS client` means. You cannot ask it to give you the experience of having caused it at half past two in the morning, which is the thing that makes you recognise it in one second rather than ten minutes.

The practical consequence: **keep an assistant open while you work through this.** Ask it the "why" questions this book no longer answers at length. Have it write the notes and the runbook as you go — that is how most people now learn a platform, and it produces something your colleagues can use. This book supplies what the assistant does not have: a sequence that works, the numbers that are real, and the failures worth causing.

What you need

A laptop with 8 GB of RAM, a container runtime, and roughly thirty minutes per chapter. Docker, Podman, or nerdctl — the commands are the same. Appendix A installs all of them. No cloud account, no credit card, no spare server, no Kubernetes cluster until Chapter 16, and that one runs in a container too.

The companion repository

```
git clone https://github.com/thomaszachmann/books.git
cd books/vault-in-practice
```

Work inside `books/vault-in-practice`. Every path printed here is relative to that directory. The book calls it `~/vault-lab`; link it if you want the name literally:

```
ln -s "$(pwd)" ~/vault-lab
```

The repository is organised by chapter. `chapters/ch06/policies/` holds Chapter 6's policies, including the deliberately broken one. `chapters/ch10/setup-database.sh` builds Chapter

10's database engine in one command. Each chapter directory has a `README.md` stating the state that chapter expects.

Use it to check your work and to recover. Do not use it instead of typing — that is the one reliable way to get nothing out of this book.

How the chapters work

Every chapter has the same parts, in the same order.

Why this chapter exists. The concrete problem, in a paragraph. No chapter opens with a definition.

Concepts. Only what you need to do the lab. Short by design; ask your assistant for the long version.

Lab. Numbered steps you type, with the expected output printed so you can tell whether it worked. Every lab starts from a defined state and tells you how to get back to it.

Break it on purpose. Two to four failures per chapter. You cause each one, read the real message, and fix it. These are the messages you will meet again, and having caused them is what makes them recognisable.

What you should now be able to do. The checklist.

Do this without looking. Five tasks with no walkthrough, inside the lab rather than after it, because a task placed after the recap is a task nobody does.

Worked solutions. For the tasks above, at the end of the chapter. Read them after you have tried, not instead.

Review questions. In **Appendix L**, grouped by chapter, written to the shape of the certification exam — 256 of them. Each chapter points you there. Appendix J is a different thing: one timed practice exam you sit end to end when you think you are ready.

Exam objectives covered. At the *end* of each chapter, once you have done the work. Individual facts that appear on the exam are marked **Exam-relevant** where they come up. Appendix F maps every published objective to the chapter and lab that covers it.

Where OpenBao behaves differently from Vault, a box says so and names the versions it was measured against:

OpenBao differs. …

There are fewer of these than you might expect. Chapter 18 prints the measurement rather than asserting compatibility.

The chapters that will annoy you

Five chapters ask you to break things properly. In Chapter 3 you lose unseal keys on purpose and discover there is no way back. In Chapter 21 you kill enough nodes to lose quorum. In Chapter 22 you configure Vault to unseal itself using itself and watch it refuse to start. In Chapter 23 you break the only audit device and watch a healthy Vault refuse every request. In Chapter 24 you revoke the last token that can administer your installation, and get back in with the unseal keys.

These are the most valuable chapters here. Nobody who has recovered a sealed cluster at three in the morning learned it from documentation.

If you are studying for the certification

Work through the book, use Appendix L after each chapter, then Appendix F as a coverage checklist and Appendix J as the timed rehearsal. You will be better prepared than somebody who memorised answers, because you will have typed the commands and caused the errors.

Conventions

Commands you type appear with a `$` prompt:

```
$ vault status
```

Output appears without one:

```
Key             Value
---             -----
Sealed          false
```

Long commands are broken with a backslash. Type them as one line or paste them including the backslashes — both work.

Type them rather than pasting them. This book deliberately avoids clever one-liners for that reason: `jq` pipelines and shell loops can be copied but not learned, and the commands here are

meant to end up in your fingers. You will need them under pressure, and possibly in an exam room, where there is nothing to copy from.

Where indentation matters, do not copy out of the PDF at all. Extracting text from a PDF frequently discards leading spaces. HCL and most shell input do not care; YAML does, and a flattened `docker-compose.yml` fails with a parser error that says nothing about copying. Every file in this book is in the companion repository, verbatim, under `chapters/<n>/`. Take indented files from there and type the rest.

```
$ vault write auth/approle/role/payments \
    token_policies="payments-read" \
    token_ttl=1h
```

Placeholders to replace with a value from your own system appear in angle brackets: `<unseal-key>`, `<lease-id>`. These are never literal.

Chapter 1 — The Problem With Secrets

Why this chapter exists

A database password sits in a deployment manifest:

```
env:
  - name: DB_PASSWORD
    value: "Mer1dian!Freight2019"
```

The manifest is in Git. Git is mirrored to three places, cloned to nineteen laptops, and backed up nightly. Search the organisation for that string and it appears in fourteen repositories. A contractor pasted it into a Jira ticket in 2021 and left in 2022.

That is **secret sprawl**, and it costs you four specific things:

You cannot	Because
say who has access	fourteen files, unknown readers
rotate	four teams, and something will break
attribute	every service connects as `meridian_app`
revoke	there is no list to revoke from

Visibility, rotation, attribution, revocation. Everything Vault does aims at one of those four. Hold them; the rest of the book is their implementation.

What you can do with Vault

Not what it *is* — what you will actually make it do:

- **Store a secret once** and hand it out under policy, with every read recorded

- **Issue a database user on demand** that lives one hour and deletes itself — Chapter 10
- **Encrypt application data** without the application holding a key, and without Vault storing the data — Chapter 12
- **Issue TLS certificates** from your own CA — Chapter 13
- **Take any of it back** with one command — Chapter 11

The last one is what separates Vault from a password manager. A password manager stores. Vault stores, *issues*, and *withdraws* — and most of what it issues did not exist until it was asked for.

Write a secret and read it back, then break it three ways. The four questions and where the data lives are after.

Lab 1 — Your first secret

The development server: one process, memory storage, no TLS, starts unsealed. It exists so you can learn without configuring anything, and it is unsuitable for anything real. Chapter 2 builds the version that is not a toy.

Step 1 — A container runtime

```
$ docker --version
Docker version 27.3.1, build ce12230
```

Docker is not required. Any of these works, and every command in this book runs unchanged on all of them:

Runtime	Note
Docker / Docker Desktop	what the examples show
Podman	rootless by default; `alias docker=podman`
nerdctl + containerd	closest to what Kubernetes runs
the `vault` binary alone	no container at all, Appendix A

On Podman with SELinux — Rocky, RHEL, Fedora — bind mounts need `:z`. That does not matter yet; it matters in Chapter 2 and it is the single most common reason a Vault container cannot read its own certificate.

If nothing is installed, Appendix A covers all four.

Step 2 — Start it

First check that nothing already holds port 8200. Every chapter in this book uses it, and a container that cannot bind fails in a way that looks like Vault being broken:

```
$ ss -lntp | grep 8200
```

On macOS, or on a machine without `ss`:

```
$ lsof -nP -iTCP:8200 -sTCP:LISTEN
```

No output means the port is free. If something answers, it is usually a Vault from an earlier attempt:

```
$ docker ps --filter publish=8200
$ docker rm -f vault-dev
```

If it is not a container, the second column of the `lsof` output is the process id, and `kill <pid>` ends it. Check what it is before you do that.

Now start Vault:

```
$ docker run --rm -d --name vault-dev \
    -p 8200:8200 \
    -e VAULT_DEV_ROOT_TOKEN_ID=root \
    -e VAULT_DEV_LISTEN_ADDRESS=0.0.0.0:8200 \
    --cap-add=IPC_LOCK \
    hashicorp/vault:1.18
```

`docker run -d` prints a container id and nothing else, which tells you it started and not that it is running. Confirm:

```
$ docker ps --filter name=vault-dev
```

```
CONTAINER ID   IMAGE                  STATUS         NAMES
c4f81a2b93e7   hashicorp/vault:1.18   Up 4 seconds   vault-dev
```

`Up` is the word to look for. If the container is missing from the list it exited, and `docker logs vault-dev` says why.

`VAULT_DEV_ROOT_TOKEN_ID=root` fixes the administrative token instead of generating a random one. Convenient here, catastrophic anywhere else.

`--cap-add=IPC_LOCK` lets Vault call `mlock` so its memory is never written to swap. Without it, secrets can land on disk in the swap file. Chapter 2 measures this.

Step 3 — Install the client, or borrow it

You need the `vault` command. Three ways, in order of how long they take:

```
# macOS
$ brew install hashicorp/tap/vault

# Linux, no package manager needed
$ curl -sLO https://releases.hashicorp.com/vault/1.18.1/\
vault_1.18.1_linux_amd64.zip
$ unzip -q vault_1.18.1_linux_amd64.zip
$ sudo mv vault /usr/local/bin/

# or use the one inside the container you just started
$ docker exec -e VAULT_ADDR=http://127.0.0.1:8200 \
    -e VAULT_TOKEN=root vault-dev vault status
```

The third works immediately and is enough for this chapter. Install the binary before Chapter 2 — from there on you will be talking to a server that is not in your container.

Step 4 — Point the client at it

```
$ export VAULT_ADDR='http://127.0.0.1:8200'
$ export VAULT_TOKEN='root'
```

Do not skip the first line. `VAULT_ADDR` defaults to **HTTPS**, and the development server speaks HTTP. You will meet that error on purpose in Lab 1B.

Step 5 — Confirm it is running

```
$ vault status
```

```
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    1
Threshold       1
Storage Type    inmem
Version         1.18.1
```

Four lines you will read for the rest of your career:

`Sealed false` — the barrier is open. Check this field first when anything is broken. `Initialized true` — the keys exist. Happens once per cluster. `Total Shares 1` / `Threshold 1` — dev splits into one piece and needs one. Production defaults are 5 and 3; Chapter 3. **Exam-relevant.** `Storage Type inmem` — stop the container and everything is gone.

Step 6 — Open the web interface

The development server has the UI enabled already. Open:

```
http://127.0.0.1:8200/ui/
```

`http://127.0.0.1:8200/` redirects there, so either works.

Sign in with **Method: Token**, token `root`. That is the same token your shell is using — the UI and the CLI are two clients of one API, with no separate account system. Chapter 4 makes that concrete.

This is not the default for a real server. Your own Vault in Chapter 2 needs one line in its configuration:

```
ui = true
```

Without it, `/ui/` returns **404** while the API works perfectly — the service is fine and only the interface is missing. Measured on the Chapter 2 server: `404` before the line, `200` after. You can run Vault entirely without the UI, and many people do.

Step 7 — Write a secret

```
$ vault kv put secret/meridian/tracking \
    db_user=tracking_svc \
    db_password=correct-horse-battery-staple
```

```
====== Secret Path ======
secret/data/meridian/tracking

======= Metadata =======
created_time       2026-08-19T09:41:12.442Z
version            1
```

Two things in that output.

You wrote `secret/meridian/tracking`; Vault reports `secret/data/meridian/tracking`. That extra `data` segment is the single most common cause of broken Vault policies. Lab 1B makes you meet it. **Exam-relevant.**

And `version 1` — Vault kept a version. Write again for version 2, with version 1 still readable.

In the UI: `secret/` in the left navigation, then `meridian/`, then `tracking`. The values are masked; the eye icon reveals them. Note that the UI shows `secret/meridian/tracking` — it hides the `data` segment that the API requires, which is exactly why the next lab exists.

Step 8 — Read it back

```
$ vault kv get secret/meridian/tracking
```

```
====== Data ======
db_password    correct-horse-battery-staple
db_user        tracking_svc
```

For scripts, one field, no decoration:

```
$ vault kv get -field=db_password secret/meridian/tracking
correct-horse-battery-staple
```

Lab 1B — Break it on purpose

Three failures. Cause each one, read the actual message, fix it. These are the three you will cause again next month, and recognising the message is faster than reasoning about the cause.

Break 1 — Forget VAULT_ADDR

```
$ unset VAULT_ADDR
$ vault status
```

```
Error checking seal status: Get
"https://127.0.0.1:8200/v1/sys/seal-status":
http: server gave HTTP response to HTTPS client
```

Read it backwards: the client spoke **HTTPS**, the server answered **HTTP**. The default is HTTPS and the development server is not.

Fix:

```
$ export VAULT_ADDR='http://127.0.0.1:8200'
```

The same message with the words reversed — *HTTPS response to HTTP client* — is the opposite mistake, and you will cause that one in Chapter 2 the first time you point at a real server without changing the scheme back.

Break 2 — Use a token that is not valid

```
$ VAULT_TOKEN=falsch vault kv put secret/test a=b
```

```
Error making API request.

URL: GET http://127.0.0.1:8200/v1/sys/internal/ui/mounts/secret/test
Code: 403. Errors:
```

403, not 401. Vault does not distinguish “no such token” from “this token may not do that” in the status code, which matters later: a policy problem and a token problem produce the same code. The URL in the error tells you which call failed, and it is usually the mount lookup rather than the write.

Fix:

```
$ export VAULT_TOKEN='root'
```

Break 3 — The `/data` path

Use the raw `read` command instead of `kv get`:

```
$ vault read secret/meridian/tracking
```

```
WARNING! The following warnings were returned from Vault:

  * Invalid path for a versioned K/V secrets engine. See the API
    docs for the correct path.
```

No data, and a warning rather than an error. The KV version 2 engine puts data under `data/` and metadata under `metadata/`; `vault kv get` inserts that segment for you and `vault read` does not.

Fix — say the real path:

```
$ vault read secret/data/meridian/tracking
```

```
Key         Value
---         -----
data        map[db_user:tracking_svc]
metadata    map[created_time:... version:1]
```

Remember the shape of that output: `data` and `metadata` as two map fields. When you write your first policy in Chapter 6 and it does not work, this is why — a policy on `secret/meridian/*` matches nothing, because the actual path is `secret/data/meridian/*`.

Clean up

```
$ docker rm -f vault-dev
```

Everything is gone, because storage was `inmem`. That is the point of the development server, and the reason Chapter 2 exists.

The four questions

Every request to Vault answers these, in order:

```
    Who are you?      ->  Authentication  ->  you get a token
    What may you do?  ->  Policies        ->  attached to the token
    What do you get?  ->  Secrets engines ->  stored or generated
    For how long?     ->  Leases          ->  and then revoked
```

Chapters 5 to 11 are these four, one at a time.

Where the data lives

Vault writes secrets **encrypted** to a storage backend — a directory, a Raft cluster, a Consul cluster. The backend never sees plaintext. Steal the disk and you get an encrypted blob. Encryption is AES-256-GCM, and the boundary is called the **barrier**.

The key lives in memory only. On start, Vault has the blob and no way to read it — that state is **sealed**, and a sealed Vault refuses everything. Somebody must supply key material to reconstruct the key. That is **unsealing**, Chapter 3.

The consequence to absorb now: **restarting Vault seals it.** A stolen disk, a stolen backup and a stolen VM image are equally useless.

What you should now be able to do

- Name the four failures of secret sprawl and which Vault feature answers each
- Start a development server and write and read a secret, from the CLI and from the UI
- Read `vault status` and say whether the barrier is open
- Explain why restarting Vault seals it
- Recognise three error messages by sight and fix each in one command
- Say why `secret/meridian/x` and `secret/data/meridian/x` are both correct, in different places

What a Vault expert knows without looking it up

Not exercises — the things that should be automatic. Cover the right column and work down.

Question	Answer
Default `VAULT_ADDR` scheme	HTTPS
Field to check first when broken	`Sealed`
Production key share defaults	5 shares, threshold 3
Where the barrier key lives	memory only
Encryption algorithm	AES-256-GCM
Real path of `secret/x` in KV v2	`secret/data/x`
Metadata path of the same	`secret/metadata/x`
Status code for a bad token	403
Config line that enables the UI	`ui = true`
Capability needed for `mlock`	`IPC_LOCK`

If any right-hand cell was not immediate, that row is your next five minutes.

Review questions

Chapter questions are collected in **Appendix L**, grouped by chapter and written to the shape of the certification exam. Do Chapter 1's seven questions there now — they take four minutes and they will tell you whether the table above went in.

Exam objectives covered

Objective	
9a	Describe the encryption of data stored by Vault
5b	Contrast dynamic secrets and static secrets (introduced)
9k	Explain the value of short-lived credentials (introduced)

Appendix F maps every objective to the chapter, lab and question that covers it.

Chapter 2 — Building Your Lab

Why this chapter exists

The development server from Chapter 1 keeps everything in memory. Close the container and the secrets are gone. That is documented behaviour, and it is useless for anything past a first look.

Two questions follow. If Vault had kept those secrets, where would it have put them? And what would stop somebody from reading that file directly?

By the end of this chapter you will have a Vault that writes to disk, serves TLS with a certificate you generated, and comes back after a reboot — the environment every remaining chapter in this book assumes.

Why the development server is not a small production server

It is tempting to think of `vault server -dev` as production Vault with the difficulty turned down. It is not. It is a different thing wearing the same name, and four of its differences matter.

Storage is in memory. Nothing is written anywhere. This is why the secrets from Chapter 1 evaporated.

It starts unsealed. The whole encryption model of Chapter 3 is bypassed. You never see an unseal key because there is nothing to unseal.

There is no TLS. Traffic is plaintext HTTP, which is why Chapter 1 spent so long on `VAULT_ADDR`.

It hands you a root token on stdout. Convenient. Also the credential equivalent of leaving the key under the mat.

The result is that everything in this book that actually matters — initialisation, unsealing, storage, high availability, audit devices, auto-unseal — either cannot be demonstrated on a

development server or behaves differently there. So we build the real thing now, once, and use it for twenty chapters.

What the configuration file looks like

Production Vault is configured by a file in HCL. A minimal one has three blocks and two settings, and you type it in Step 3:

```
ui            = true
disable_mlock = false

storage "file" {
  path = "/vault/data"
}

listener "tcp" {
  address       = "0.0.0.0:8200"
  tls_cert_file = "/vault/tls/vault-cert.pem"
  tls_key_file  = "/vault/tls/vault-key.pem"
}

api_addr     = "https://127.0.0.1:8200"
cluster_addr = "https://127.0.0.1:8201"
```

Every line earns its place, and the lab shows you what each one does by taking it away. What each directive means, in detail, is after the lab — where it will mean something.

Software versions

Every command in this book was verified against these versions.

Software	Version
Vault	1.18.x
Docker Engine	27.x
Docker Compose	v2.29+
PostgreSQL (Chapter 10)	16
kind (Chapter 16)	0.24+

Vault is pinned to a specific minor version throughout. Using `:latest` in a book is how you produce a book that stops working. Where behaviour changed in a recent release, the text says so.

Lab 2 — The lab you will keep

Step 1 — Create the directory layout

```
$ mkdir -p ~/vault-lab/{config,data,logs,tls}
$ cd ~/vault-lab
$ ls
config  data  logs  tls
```

Four directories, and each one maps to something in the configuration: `config` holds the HCL file, `data` is the storage backend, `logs` will hold audit logs from Chapter 23, and `tls` holds the certificate.

This is also the layout of the companion repository. If you would rather clone than type:

```
$ git clone https://github.com/thomaszachmann/books.git
$ cd books/vault-in-practice
```

The steps below then confirm what is already there rather than creating it. Type them anyway the first time — and note that the repository directory is where `~/vault-lab` points in the rest of the book.

Step 2 — Generate a TLS certificate

This step needs `openssl`, and the steps after it need Docker Compose v2. Both are in Appendix A if `openssl version` or `docker compose version` does not answer.

Check where you are before you type anything. The command below writes into `tls/` relative to your current directory, and Step 4 mounts that exact directory into the container:

```
$ cd ~/vault-lab
$ pwd
```

```
/home/you/vault-lab
```

If that is not what it says, the certificate lands somewhere the container will never look, and you will spend an hour on a mount that is empty.

Self-signed, valid for a year, and — this is the part people get wrong — carrying the right Subject Alternative Names.

```
$ openssl req -x509 -newkey rsa:2048 -sha256 -days 365 \
    -nodes -keyout tls/vault-key.pem \
    -out tls/vault-cert.pem \
    -subj "/CN=vault.lab.local" \
    -addext "subjectAltName=DNS:vault.lab.local,DNS:vault,\
DNS:localhost,IP:127.0.0.1"
```

Check what you produced:

```
$ openssl x509 -in tls/vault-cert.pem -noout -text \
    | grep -A1 "Subject Alternative Name"
```

```
X509v3 Subject Alternative Name:
    DNS:vault.lab.local, DNS:vault, DNS:localhost,
    IP Address:127.0.0.1
```

```
$ ls tls/
```

```
vault-cert.pem   vault-key.pem
```

Two files, in `~/vault-lab/tls/`. Step 4 mounts that directory as `/vault/tls`, and Step 3's configuration refers to `/vault/tls/vault-cert.pem` — the *container* path. Putting the host path into the configuration is the other way to lose an hour here.

On Podman with SELinux the mount also needs `:z`, and the directory must be traversable — a `700` directory produces the same "permission denied" as a wrong SELinux label, so check the mode before blaming SELinux.

How this is done outside a lab. Nobody self-signs in production. Three normal answers: an internal CA — which you will run yourself in Chapter 13, and which is what most on-premises estates use; ACME, which Appendix A works through with a DNS-01 example, including the case where the server is not reachable from the internet at all; or certificates issued by Vault itself once a first Vault exists, which is the chicken-and-egg problem Chapter 14 solves.

The middle one surprises people: with the **DNS-01** challenge the certificate authority never contacts your server. It reads a TXT record in public DNS instead, so a Vault behind a firewall with no inbound rule can hold a publicly trusted certificate — and a wildcard, which no other challenge can issue. Appendix A measures the whole flow.

`IP:127.0.0.1` is the one that matters. Modern TLS clients ignore the Common Name entirely and validate against the SANs. If you connect to `https://127.0.0.1:8200` and the certificate has no IP SAN, verification fails — with an error message that names the problem clearly but only if you know to read it. You will see that error on purpose later in this chapter.

Step 3 — Write the configuration

```
$ cat > config/vault.hcl <<'EOF'
ui            = true
disable_mlock = false

storage "file" {
  path = "/vault/data"
}

listener "tcp" {
  address       = "0.0.0.0:8200"
  tls_cert_file = "/vault/tls/vault-cert.pem"
  tls_key_file  = "/vault/tls/vault-key.pem"
}

api_addr     = "https://127.0.0.1:8200"
cluster_addr = "https://127.0.0.1:8201"
EOF
```

Check what you wrote:

```
$ vault operator diagnose -config=config/vault.hcl 2>&1 | head -5
```

Errors about reaching a server are expected — nothing is running yet. What you are looking for is that the file **parsed**. A syntax error here produces a container that starts and immediately exits, and the reason is in `docker compose logs` rather than on your screen.

> **If you are copying out of the PDF, check the indentation.** Text extraction from a PDF frequently discards leading spaces, which HCL tolerates and YAML does not — Step 4 is

where that bites. The companion repository has every file in this chapter verbatim, and copying from there is the reliable route: `chapters/ch02/config/vault.hcl`.

The paths *inside* the file are container paths (`/vault/...`), not paths on your machine. The next step maps one to the other, and mixing them up is the second most common mistake in this chapter.

Step 4 — Write the Compose file

```
$ cat > docker-compose.yml <<'EOF'
services:
  vault:
    image: hashicorp/vault:1.18
    container_name: vault
    ports:
      - "8200:8200"
    cap_add:
      - IPC_LOCK
    volumes:
      - ./config:/vault/config:ro
      - ./data:/vault/data
      - ./logs:/vault/logs
      - ./tls:/vault/tls:ro
    command: vault server -config=/vault/config/vault.hcl
    restart: unless-stopped
EOF
```

Verify it before you start anything. YAML is significant to the space, and this is the file that suffers if the indentation did not survive the journey from page to terminal:

```
$ docker compose config
```

That prints the file back, normalised, if it parsed. If instead you see

```
services.vault must be a mapping
```

or

```
yaml: line 3: did not find expected key
```

then the indentation is wrong. Retype it, two spaces per level, or take `chapters/ch02/docker-compose.yml` from the companion repository.

The file belongs in `~/vault-lab/docker-compose.yml` — next to the four directories, not inside them. Docker Compose looks for it in the directory you run the command from, which is why every `docker compose` command in this book assumes you are in `~/vault-lab`. Run one from somewhere else and you get `no configuration file provided: not found`.

Two details worth pointing at.

`:ro` on the config and TLS mounts. Vault has no reason to write to either, so it cannot.

`restart: unless-stopped`. The container will come back after a Docker restart — and Vault will come back **sealed**, which is exactly the behaviour Chapter 3 is about. Leaving this in means you will experience it rather than read about it.

Step 5 — Start it

```
$ docker compose up -d
$ docker compose ps
```

```
NAME    STATUS          PORTS
vault   Up 3 seconds    0.0.0.0:8200->8200/tcp
```

Check the log before assuming it worked:

```
$ docker compose logs vault --tail 15
```

You are looking for a start-up banner ending in:

```
==> Vault server started! Log data will stream in below:
```

How long that takes: 714 ms, measured on a 4-vCPU virtual machine with `file` storage. If it is not there within a couple of seconds, it is not coming and something is wrong — Vault does not start slowly, it either starts or fails.

The other line you will see, and it is your own doing. Sooner or later the log fills with this:

```
vault | [INFO] http: TLS handshake error from 172.21.0.1:41288:
  client sent an HTTP request to an HTTPS server
```

Nothing is broken. `172.21.0.1` is the Docker bridge gateway — which is to say, your own machine — and something on it spoke plain HTTP to a listener that only speaks HTTPS. It

is almost always one of three things: `VAULT_ADDR` still set to `http://`, a browser opened at `http://localhost:8200`, or a health check somebody added without the scheme.

Vault logs it at `INFO` and carries on, which is correct: from its side a client made a mistake and was told so. Cause it on purpose now, so the log line has a memory attached to it:

```
$ VAULT_ADDR=http://127.0.0.1:8200 vault status
```

```
Error checking seal status: Get
  "http://127.0.0.1:8200/v1/sys/seal-status":
  http: server gave HTTP response to HTTPS client
```

Two messages, one mistake, seen from both ends: yours says the server answered wrongly, the server's says the client asked wrongly. Neither mentions a scheme, which is why this costs people time.

Lines above the banner that look alarming and are not. With the config and TLS directories mounted read-only, the container's entry point tries to take ownership of them and cannot:

```
chown: /vault/config/vault.hcl: Read-only file system
chown: /vault/tls: Read-only file system
Could not chown /vault/config (may not have appropriate permissions)
```

Three or four of those, every start, and they are the intended consequence of `:ro` in Step 4. Vault reads both directories and never writes to them. Ignore them.

What is *not* harmless is anything containing `x509`, `bind`, or `failed to`. Those stop the server, and you will cause each of them deliberately in Step 11.

Step 6 — Point your client at it

This is different from Chapter 1 in two ways. The scheme is now `https`, and the client needs to be told which certificate to trust.

```
$ export VAULT_ADDR='https://127.0.0.1:8200'
$ export VAULT_CACERT=~/vault-lab/tls/vault-cert.pem
```

`VAULT_CACERT` points at a **file**. There is a companion variable, `VAULT_CAPATH`, which points at a **directory** of certificates. Mixing them up is a small, persistent annoyance; the mnemonic is that CERT is a certificate file and PATH is a path to a folder.

Make these permanent, because you will open new terminals for the next twenty chapters:

```
$ cat >> ~/.zshrc <<'EOF'
export VAULT_ADDR='https://127.0.0.1:8200'
export VAULT_CACERT="$HOME/vault-lab/tls/vault-cert.pem"
EOF
```

Use `~/.bashrc` if you are on bash.

Step 7 — Look at the status

```
$ vault status
```

```
Key                Value
---                -----
Seal Type          shamir
Initialized        false
Sealed             true
Total Shares       0
Threshold          0
Unseal Progress    0/0
Storage Type       file
HA Enabled         false
```

This looks like a broken server and is not. Read it carefully, because it is the state every real Vault begins in.

`Initialized false` — the encryption keys do not exist yet. Nobody has ever set this Vault up.

`Sealed true` — the barrier is closed. Every request except status will be refused.

`Total Shares 0` — there are no unseal keys, because there is nothing to unseal yet.

`Storage Type file` — the configuration took effect.

`HA Enabled false` — file storage cannot do high availability. Expected, and it changes in Chapter 20.

Prove that it really is closed:

```
$ vault kv put secret/test foo=bar
```

```
Error writing data to secret/test: Error making API request.

URL: PUT https://127.0.0.1:8200/v1/secret/test
Code: 503. Errors:

* Vault is sealed
```

Status code 503, "Vault is sealed." Commit that pairing to memory now. When a service reports 503 from Vault, it is not a networking problem and not a permissions problem — somebody restarted something.

Step 8 — Look at what is on disk

```
$ ls -R data/
```

```
data/:
core  sys
```

Nearly empty, because nothing has been written yet. Vault created its skeleton and stopped. In the next chapter you will initialise it, write a secret, and come back to this directory to search it for the plaintext — which will not be there.

Step 9 — Write the reset script

Every chapter in this book starts from a known state, and several of them ask you to destroy things. You need a way back.

```
$ cat > scripts/reset-lab.sh <<'EOF'
#!/usr/bin/env bash
set -euo pipefail

cd "$(dirname "$0")/.."

echo "This deletes all Vault data in $(pwd)/data"
read -r -p "Type RESET to continue: " confirm
[ "$confirm" = "RESET" ] || { echo "Aborted."; exit 1; }

docker compose down
rm -rf data/* logs/* .env
docker compose up -d vault

echo "Lab reset. Vault is uninitialized and sealed."
echo "Run: vault status"
EOF
$ chmod +x scripts/reset-lab.sh
```

The repository ships this as `scripts/reset-lab.sh`, and the `Makefile` wraps it:

```
$ make reset
```

The confirmation prompt is not decoration. You will run this script when frustrated, and at that moment you want one thing standing between you and an accidental wipe of a chapter's work.

Do not run it now. You have nothing to reset.

Step 10 — Confirm persistence

The whole point of this chapter:

```
$ docker compose restart
$ sleep 3
$ vault status | grep -E "Initialized|Sealed"
```

```
Initialized     false
Sealed          true
```

Unchanged across a restart — the state came from disk rather than being recreated. In Chapter 1 the equivalent action erased everything.

Step 11 — Break it on purpose

Five failures. Cause each one, read the message, fix it. Every one of these will happen to you on a real installation, and the only thing that shortens it from an afternoon to a minute is having seen it.

Type the five breaks below yourself. That is the exercise, and the next few pages are all you need.

The repository also contains a script that runs the same five in sequence, against a throwaway Vault under `/tmp` so the one you built keeps running. It is there for two purposes: checking that your machine produces the same messages this book printed, and re-running the set later without working through the pages again. Ignore it on your first pass.

```
$ ./chapters/ch02/reproduce-errors.sh
```

Break 1 — A certificate with no SAN.

A certificate says which names it is valid for. Step 2 put those names in the **Subject Alternative Name** extension, using `-addext`. Leave that flag off and the certificate names nothing — the `/CN=` in the subject line looks like a name, and no modern client reads it.

Cause it. This is the Step 2 command with the `-addext` line deleted:

```
$ openssl req -x509 -newkey rsa:2048 -sha256 -days 365 -nodes \
    -keyout tls/vault-key.pem -out tls/vault-cert.pem \
    -subj "/CN=vault.lab.local"
```

Confirm that you made a certificate with no names in it:

```
$ openssl x509 -in tls/vault-cert.pem -noout -ext subjectAltName
```

```
No extensions in certificate
```

Now hand it to Vault and connect:

```
$ docker compose restart && vault status
```

```
Error checking seal status: ... tls: failed to verify certificate:
  x509: certificate relies on legacy Common Name field,
  use SANs instead
```

Read what it says: not *wrong name*, but *the Common Name is not a name*. Go removed Common Name fallback in 1.15, and every modern client did the same. Older Go versions worded it as *doesn't contain any IP SANs*; same fault.

This is the single most common certificate error in this book's subject, and it appears again in Chapter 13 when you issue certificates yourself.

Fix: regenerate with the full `-addext` line from Step 2, check the extension is there, then `docker compose restart`.

Break 2 — Forget `VAULT_CACERT`.

```
$ unset VAULT_CACERT && vault status
```

```
Error checking seal status: Get "https://127.0.0.1:8200/v1/sys/
  seal-status": tls: failed to verify certificate:
  x509: certificate signed by unknown authority
```

The certificate is fine; your client does not trust the CA that signed it. Note how close this looks to Break 1 and how different the cause is — *relies on legacy Common Name* is the server's certificate being wrong, *signed by unknown authority* is your client not being told.

Fix:

```
$ export VAULT_CACERT=~/vault-lab/tls/vault-cert.pem
```

`VAULT_SKIP_VERIFY=true` also makes it go away, by disabling verification entirely. It has a way of travelling from a laptop into a deployment script and then into production. Do not start.

Break 3 — Make the data directory unwritable.

```
$ chmod 500 data && docker compose restart
$ docker compose logs vault --tail 5
```

```
failed to initialize barrier: permission denied
```

The container runs as an unprivileged user and the host directory belongs to you.

Fix:

```
$ chmod 777 data logs && docker compose restart
```

`777` is acceptable in a lab. On a real host you create a dedicated user and set ownership — and this is exactly the class of thing that makes a first production install take longer than planned.

Break 4 — Remove `IPC_LOCK`. Delete the `cap_add` block from `docker-compose.yml` and `docker compose up -d`:

```
Error initializing core: failed to lock memory:
  cannot allocate memory
```

Fix: put `cap_add: [IPC_LOCK]` back, or set `disable_mlock = true` in the configuration. If you choose the second, know what you gave up: Vault's memory can now reach swap.

Break 5 — Leave Chapter 1's container running.

```
$ docker run -d -p 8200:8200 hashicorp/vault:1.18
```

```
Error: cannot listen on the TCP port: listen tcp4 :8200:
  bind: address already in use
```

Fix:

```
$ docker ps -a | grep 8200
$ docker rm -f vault-dev && docker compose up -d
```

Put everything back before Step 12 — or run `./reset.sh` from Step 9, which is what it is for.

What a Vault expert knows without looking it up

Everything above told you exactly what to type. These do not, and they are the part that shows whether the chapter worked. Worked solutions are at the end of the chapter — for after you have tried, not instead of trying.

2.1 Stop the lab with `docker compose down` and start it again. Predict the values of `Initialized` and `Sealed` before you run `vault status`, then check.

2.2 Change the listener to port 8300 instead of 8200. What else has to change for `vault status` to work, and what breaks if you change only the configuration file?

2.3 Generate a second certificate deliberately missing the IP SAN, point Vault at it, and reproduce the SAN error. Then fix it. Write down the exact error text — you are building a personal index of error messages, and it will be more useful than any set of notes.

2.4 Set `disable_mlock = true`, restart, and confirm Vault still starts. Then explain in one sentence what an attacker gains from that change, and under what circumstance it is nonetheless the correct setting.

2.5 Without looking it up: which two storage backends are both supported by HashiCorp and capable of high availability? Then check your answer against the table earlier in this chapter.

The configuration file, line by line

`storage`

Where the encrypted data goes. Vault supports several backends and this book uses two of them:

Backend	Persists	Supports HA	Used in
`inmem`	no	no	dev server only
`file`	yes	**no**	Chapters 2–18
`raft`	yes	yes	Chapters 19–22
`consul`	yes	yes	discussed, Chapter 20

`file` is the right choice for a single-node lab: simple, visible, and you can look at what it writes. It cannot do high availability, which is the entire reason Chapter 19 replaces it with Raft.

`path` can point anywhere the container can reach, including an external disk or a NAS mount — `/mnt/vault-data`, an NFS export, an iSCSI LUN. Two conditions, and both are hard requirements rather than preferences:

- The filesystem must support **POSIX locking**. Vault takes a lock on its data directory, and an NFS export mounted without locking gives you a Vault that starts and then corrupts itself under concurrent writes. NFSv4 with `lock` is fine; NFSv3 with `nolock` is not.
- Latency goes straight into every request. Vault writes synchronously, so a NAS on the other side of a slow link makes every secret read slow.

For a homelab NAS: usable, and worth measuring before you trust it. For production: HashiCorp's answer is Raft on local disks rather than shared storage, because Raft replicates the data instead of relying on the filesystem to be available. Chapter 20 measures the difference.

The two backends HashiCorp supports **and** which do high availability are Raft and Consul. Everything else in the list either does not persist, does not cluster, or is community-maintained. That sentence is worth remembering; it answers a question people spend surprising amounts of time on.

`listener`

The network socket. `0.0.0.0:8200` means "all interfaces, port 8200" — necessary inside a container, since binding to `127.0.0.1` would make Vault unreachable from the host.

You can bind to one address instead — `address = "10.0.5.12:8200"` — and on a real server that is the better choice, because it stops Vault appearing on a management interface nobody meant to expose. **But the address must exist on that machine.** Bind to an address the host does not have and Vault refuses to start with `listen tcp 10.0.5.12:8200: bind: cannot assign requested address`. Inside a container the address also has to be the *container's*, not the host's, which is why the lab uses `0.0.0.0` and lets the port mapping do the restricting.

Port **8200** is the API and UI. You will also see **8201** referenced as `cluster_addr`; that is the port Vault nodes use to talk to each other. It does nothing on a single node, and it becomes essential in Chapter 21. Set it now so you do not have to think about it later.

`tls_cert_file` and `tls_key_file`

You could write `tls_disable = 1` and skip the certificate. Plenty of tutorials do, and it would make the next twenty minutes shorter.

We are not going to, for one reason: **disabling TLS hides an entire class of problem that you will meet on your first real installation.** Certificate SAN mismatches, missing CA bundles, `VAULT_CACERT` — these produce errors that are baffling the first time and obvious the second. This is the cheapest possible place to have that first time.

`ui`

`ui = true` serves the web interface at `/ui/`. Leave it out and that path returns **404** while the API keeps working perfectly — measured on this exact configuration: `404` without the line, `200` with it.

You can run Vault entirely from the CLI and many people do; the UI is convenient for looking at a secret, and useless for anything you want repeatable. If you set it, remember that it is reachable by anyone who can reach port 8200, which is an argument for the specific-address binding above.

`disable_mlock`

`mlock` is a system call that asks the kernel never to write a process's memory to swap. Vault uses it so that decrypted secrets cannot end up in a swap file on disk.

It requires a capability the container does not have by default. Two ways out:

- Grant it: `cap_add: [IPC_LOCK]` in Docker
- Turn the feature off: `disable_mlock = true`

Which is correct depends on where you are. On a virtual machine, grant it — swapping secrets to disk is a genuine risk. In a container or on Kubernetes, the near-universal practice is `disable_mlock = true` combined with **disabling swap on the node**, because the container has no business locking host memory and there is no swap to protect against anyway.

We grant the capability, because we can, and because it means you see the setting doing something.

`api_addr`

The address at which clients can reach this node. On one node it is nearly cosmetic. In a cluster it is how a standby node tells a client where the active node is, and getting it wrong produces redirects to addresses nobody can reach.

What you should now be able to do

You can now:

- Explain the four ways a development server differs from a real one, and why each matters
- Write a Vault configuration file with `storage`, `listener`, `api_addr`, and `cluster_addr`
- Name which storage backends persist and which support high availability
- Explain what `mlock` protects against, and when `disable_mlock = true` is the right answer
- Generate a TLS certificate with the Subject Alternative Names a client will actually check
- Distinguish `VAULT_CACERT` from `VAULT_CAPATH`

- Recognise `Initialized false` / `Sealed true` as the normal starting state of a new Vault
- Identify HTTP 503 as “Vault is sealed”
- Reset the lab to a known state, deliberately
- Diagnose the five most common start-up failures from their error text

Review questions

Chapter questions are in **Appendix L**, grouped by chapter and written to the shape of the certification exam. Do Chapter 2’s there now.

Exam objectives covered

Objective	
9c	Describe storage backends
6f	Configure environment variables
9a	Describe the encryption of data stored by Vault