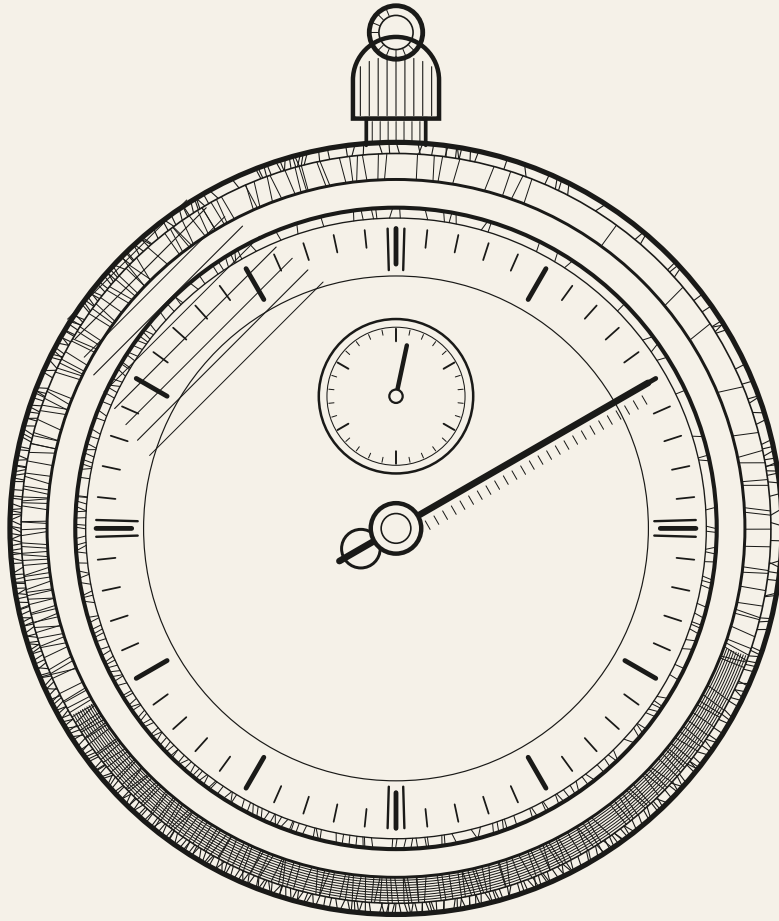




Vault in Production

Hands-on labs for Vault and OpenBao after day one

21 chapters, 20 labs, every one broken on purpose

Ends with the 66-point review you run before go-live

Thomas Zachmann

Vault in Production

Operating HashiCorp Vault and OpenBao After Day One

Thomas Zachmann

Contents

Vault in Production	1
Acknowledgements	1
About the Author	2
Who this book is for	3
What you need to have read	3
How the chapters work	3
Why this book is shorter than it used to be	4
How this book was written	4
The lab	5
Chapter 1 — What Day Two Actually Means	6
Why this chapter exists	6
The three questions Day 2 asks	6
Lab 1 — Measure what depends on you	7
What this book assumes	9
What replaced the exam boxes	9
The shape of the operational cost	10
Break it on purpose	10
Do this without looking	11
What you should now be able to do	11
Worked solutions	12
What a Vault operator knows without looking it up	12
Runbook fragment — Monthly measurement	13
Chapter 2 — Before the First Real Secret	14
Why this chapter exists	14
Three instances, three different jobs	14

Lab 2 — Fail your own acceptance check	15
What a development server actually is	18
The list	18
Break it on purpose	19
Do this without looking	20
What you should now be able to do	20
Worked solutions	21
What a Vault operator knows without looking it up	21
Runbook fragment — Go-live	22

Vault in Production

Operating HashiCorp Vault and OpenBao after day one

Thomas Zachmann

In Practice — Book 04

Copyright © 2026 Thomas Zachmann. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author, except in the case of brief quotations embodied in reviews and certain other non-commercial uses permitted by copyright law.

HashiCorp, Vault, Terraform, Consul, and Nomad are trademarks of HashiCorp, Inc. OpenBao is a project of the Linux Foundation. Kubernetes is a registered trademark of The Linux Foundation. Docker is a trademark of Docker, Inc. PostgreSQL is a trademark of the PostgreSQL Community Association of Canada. All other trademarks are the property of their respective owners. This book is not affiliated with, authorised by, or endorsed by any of them.

The commands and configurations in this book were run against the versions listed in Appendix A. Software changes. Verify anything that matters against the current documentation before applying it to a system you care about.

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 builds 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. Since 2020 he has worked almost exclusively in regulated environments — federal printing and identity services, public-sector IT providers, cooperative banking, and security technology vendors — under frameworks including BSI IT-Grundschutz, BSI C5, ISO 27001, BAIT and DORA.

That background is why this book insists on evidence: every number in it was produced by breaking something and writing down what happened, because that is the only form an auditor — or a colleague at three in the morning — has any reason to accept.

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.

Underneath that is twenty years of 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.

The first book in this series exists because teams understand Vault in an hour and take months to run it well. This one exists because of what comes after that: the installation that has been running for two years, that eleven services now depend on, and that nobody has restored from a snapshot since the day it was built.

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

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

Available for platform work, hardening reviews, and team enablement.

Who this book is for

Somebody who has been running Vault for six months and has found that the documentation stops where the questions start.

You have an installation. It works. What you do not have is an answer to what happens when it does not, what has been quietly accumulating inside it, and which of the numbers in your recovery plan anybody has ever measured.

There is a second reader: the person who has to approve the operational budget, and who needs to know what the second year costs before signing off on the first.

What you need to have read

The first book in this series, *Vault in Practice*, or equivalent experience. This one assumes you can initialise and unseal a Vault, write a policy that is not `secret/*`, explain the difference between a token and an entity, and say what a lease is.

Nothing from that book is repeated here. Where a mechanism was already explained, you get a pointer rather than a recap.

How the chapters work

Why this chapter exists. The concrete operational problem, in a paragraph or two.

Lab. Numbered steps you type, against a cluster you can break. Every lab starts from a fresh cluster and ends with `make reset`, so any chapter can be the one you open first.

Concepts. What is actually happening, after the lab and under their own headings. Short.

Break it on purpose. Each entry names a failure, gives you the command or the action that causes it, prints the message it produces, and then the fix. Cause them. On day two the difference between a twenty-minute outage and a two-hour one is almost entirely whether the person on call has seen the message before.

Do this without looking. Numbered tasks that repeat the lab without the text in front of you.

What you should now be able to do. What you can now do. If one line is not true yet, the lab is still open.

Worked solutions. Short answers to the tasks, for checking, not for reading first.

What a Vault operator knows without looking it up. A two-column table. Cover the right column and answer from memory.

Runbook fragment. A page to lift into your own documentation, with blanks where your names and numbers go. Deliberately incomplete — a runbook that arrives finished is one nobody reads. There are no exam objectives. There is no certification for this material; the tasks and tables are there so you notice what has not stuck yet.

Why this book is shorter than it used to be

An earlier draft explained things. Then the world changed underneath it: you have an AI assistant in the terminal next to this book, and it will explain any concept, flag or error message instantly, at whatever depth you ask for. A chapter that competes with that is a chapter you will skim.

So this book does the two things an assistant cannot. **It knows what actually happened** — every number and duration here was measured on a real cluster, and the book says which. And **it makes you do it wrong on purpose**, which is the only way to recognise a failure in one second rather than in twenty minutes.

Keep an assistant open while you work through this. Ask it the “why” questions this book no longer answers at length, and have it write your runbook as you go. This book supplies what it does not have: numbers that are real, and the failures worth causing.

How this book was written

Twenty years of building systems teaches one habit worth more than any particular technology: distrust the number you did not measure yourself, and distrust twice the number that came out convenient.

So every figure in these twenty-one chapters was produced by breaking something on purpose, on the two hosts described in Appendix A, and writing down what happened. Not one of them is

quoted from documentation. Where a measurement is a single run, Appendix C says so; where two runs disagreed, it prints both.

That method has a consequence for the tone. Several times in these chapters I got a result, believed it, and was wrong — a credential that appeared to survive an outage because my test never checked a password, a benchmark where the standby beat the leader because the standby was sealed, a monitoring check that reported “unknown” for every healthy node. Each of those is still in the book, in the chapter where it happened, because the mistake is more instructive than the corrected number and because you will make the same class of mistake on your own installation.

What is not in this book: anything I have not run. There are no worked examples from client engagements, no anonymised incidents, no “a customer once”. Years of work in regulated environments produce stories, and none of them belong in a book where the reader cannot check the claim. What you get instead is a lab you can run, and numbers you can reproduce or refute.

The lab

Docker or podman, and a machine with 16 GB of RAM. Most chapters run a three-node cluster from one `make up` ; Chapter 12 wants a Kubernetes cluster in kind; Chapter 13 adds a Postgres container. Nothing in this book needs a cloud account or costs money.

The cluster is HashiCorp Vault 1.18.5, the community edition — free to use, under the Business Source License, and not open source. OpenBao, the open-source fork, shares the mechanisms this book measures, but only one chapter was measured against it: Chapter 20 runs OpenBao outright, because the feature it measures — a PKCS#11 seal for a hardware security module — is Enterprise-only in Vault and part of the free build in OpenBao.

Chapter 3 has no lab at all, and says so.

Appendix A installs everything and lists the versions every measurement in this book was taken against.

Chapter 1 — What Day Two Actually Means

Why this chapter exists

The first book ended with a checklist and a Vault that passes it. Two things happen after that, and neither is in the documentation.

The first is that the installation stops being a project and becomes a dependency. Nobody thanks you for it. It appears in other teams' incident reviews as a line that reads “waiting on Vault”, and the people writing those lines have never read its configuration.

The second is that the failure modes change. Day 1 failures are configuration errors: a wrong `VAULT_ADDR`, a policy that grants too much, a certificate without the right SAN. They are annoying, they are immediate, and the error message names the cause. Day 2 failures are different. They accumulate. Nothing is misconfigured, every command succeeds, and eleven months later the storage backend is four hundred gigabytes and the cluster will not start.

This book is about the second kind.

The three questions Day 2 asks

Everything in this book is a variation on one of three questions, and it is worth naming them now because they are the ones nobody asks during the installation project.

What stops when Vault stops? Not “is Vault highly available” — that is a property of Vault. This is a property of everything else. Chapter 13 makes you measure it, and the number is usually worse than the estimate.

What accumulates? Leases, tokens, identity entities, audit log, storage. Each grows for a good reason, none of them shrinks on its own, and the first symptom of any of them is the same: something that used to take milliseconds takes seconds.

What can you prove? Six months after the fact, to somebody who does not trust you, with the evidence you happened to be keeping at the time. This is the question that turns an audit log from a file into a control, and it is answerable only if somebody made a decision before it was asked.

Do it first. The numbers, the trade-offs and the reasons are after the lab, where the thing you just watched gives them weight.

Lab 1 — Measure what depends on you

The rest of this book breaks things deliberately. This chapter measures, because you cannot judge a blast radius you have never counted.

The cluster from the repository, three nodes, freshly initialised:

```
$ make reset
$ make up && make init && make unseal
$ . ./scripts/vault-env.sh
```

Every lab in this book starts with these lines. `make reset` first means a lab never inherits whatever the previous one left behind; on a clean checkout it has nothing to do.

A fresh cluster gives small numbers, which is fine: the point is the habit of taking them. The numbers get interesting against a Vault that has lived — the lab from the first book in the state its final chapter left it, or any Vault you are allowed to be honest about. The whole lab as one line per run, to append to a file you keep: `./chapters/ch01/measure.sh`.

Step 1 — Count the authentication paths

```
$ vault auth list -detailed
```

Each mount is a way in, and each one is a way in that can break on its own. A Kubernetes auth mount stops working when the cluster's API server certificate rotates. An OIDC mount

stops working when the identity provider changes an endpoint. Neither failure looks like a Vault failure from the outside.

Write the number down. Then, for each mount, write the name of the system whose availability it depends on. That second column is the one that matters and it is usually empty in people's heads.

Step 2 — Count what has a lease

```
$ vault list sys/leases/lookup/  
$ vault list sys/leases/lookup/database/creds/
```

Leases are the load-bearing structure of Day 2. They are also the thing that grows silently. What you want here is not the number but the *trajectory*, so record it with a timestamp and do it again next week.

Step 3 — Ask the storage how big it is

```
$ vault status | grep -i storage  
$ sudo du -sh cluster/data*
```

The `sudo` is not optional. Vault creates its Raft directory inside the container as UID 100 with mode 0700, and a plain `du` as your own user reports “Permission denied” and counts almost nothing.

For a lab this is a few megabytes. Record it anyway. The value of this measurement is entirely in the second and third reading.

Step 4 — Find the consumers Vault does not know about

This is the step with no command.

Vault knows about tokens and mounts. It does not know that the payments service reads one secret at start-up and then holds it for six weeks, or that a scheduled job on somebody's laptop refreshes a certificate every Monday. Those consumers are invisible to `vault list` and they are the ones that break in the interesting way.

The audit log is the only honest source:

```
$ jq -r '.request.path' logs/audit.log \
  | sort | uniq -c | sort -rn | head -20
```

Every path anybody actually used, ranked. Compare it with the list of consumers you believe you have. The difference is your Day 2 backlog.

If that command produced nothing, the audit device is not enabled, and that is the first finding of this book: **you cannot answer any of the three questions without it**. On a fresh lab it is not, so turn it on and run the command again after a few requests:

```
$ vault audit enable file file_path=/vault/logs/audit.log
```

Then clean up:

```
$ make reset
```

What this book assumes

That you have read the first one, or know its contents from elsewhere. Specifically: that you can initialise and unseal a Vault, write a policy that is not `secret/*`, explain the difference between a token and an entity, and say what a lease is. If any of those is shaky, the earlier book covers it and this one will not.

Nothing from that book is repeated here. Where a mechanism matters and was already explained, you will find a pointer rather than a recap. That is a deliberate constraint: a Day 2 book that re-teaches Day 1 is a Day 1 book with a longer title.

What has changed is the altitude. The earlier book asked *how do I switch this on*. This one asks *what does it do to me in the second year*.

What replaced the exam boxes

The first book opened every chapter with the certification objectives it covered and closed with questions in the exam's format. There is no certification for this material, so both are gone.

In their place, each chapter ends with a **runbook fragment**: a page you can lift into your own operational documentation, with the commands, the decision points and the places to write your

own names and numbers. They are deliberately incomplete. A runbook that arrives finished is a runbook nobody reads, because nobody had to think about it.

The shape of the operational cost

Here is the thing that surprises people who have run Vault for less than a year.

The cost of operating Vault is not proportional to the number of secrets in it. It is proportional to **the number of systems that depend on it** and **the rate at which credentials change**. A Vault holding fifty thousand static key/value secrets for one team is close to free. A Vault holding four hundred secrets, issuing dynamic database credentials to eleven services across three clusters, is a system with an on-call rotation.

That has a practical consequence for how you grow. Adding secrets is cheap; adding *consumers* is not. Every new consumer adds a dependency edge, an authentication path that can break independently, a set of leases that can pile up, and a team that will page you. The question to ask before onboarding one is not “can Vault hold this” — it always can — but “who answers when this consumer cannot reach Vault at three in the morning”.

Break it on purpose

Cause each of these, read the real message, then fix it. These are the failures you will meet again, and having produced one deliberately is what makes it recognisable in a second rather than in half an hour.

`vault list sys/leases/lookup/` returns a permission error

Cause it: run it with a token whose policy grants `read` and not `list` on that path.

The path is root-protected. A token needs `list` and `sudo` on `sys/leases/lookup/*`. This is deliberate: the lease inventory reveals the shape of every dynamic integration you have.

The audit log is enormous and `jq` is slow

Cause it: enable a file audit device and leave it a week, then run a `jq` filter over the whole file.

Expected. Shipping, retention and searchable storage for audit logs are their own project, and a large one. For now, work on a slice:

```
$ tail -100000 logs/audit.log \  
  | jq -r '.request.path' | sort | uniq -c
```

du reports a size far larger than the number of secrets suggests

Cause it: write the same key a thousand times and compare `du` against the secret count.

One secret, and the disk grew. KV v2 keeps old versions of the key (ten by default, `max_versions`), and every write also passes through the Raft log and leaves free pages in BoltDB that are reused but never returned to the filesystem. In production the same symptom has other usual suspects: leases, the audit log inside the same volume, or an identity store that grew an entity per login rather than per person. Chapter 16 takes leases and free pages apart, Chapter 15 the audit disk, Chapter 19 the identity store.

Do this without looking

1.1 Count the leases on your own cluster and write the number down with today's date. You cannot measure growth you never started measuring.

1.2 Find one path where a token can read and not list, and say what that costs whoever is on call.

1.3 Measure the audit log's growth for one hour and extrapolate to a year.

1.4 Name the three things you would look at first if Vault stopped answering right now. Then check that you can reach all three.

What you should now be able to do

You can now:

- State the three questions Day 2 asks, and say which chapter answers each
- Count your own authentication paths and name what each one depends on
- Take the three measurements whose value is in their trajectory
- Explain why consumer count, not secret count, drives operational cost

- Say why none of this is answerable without an audit device

Worked solutions

1.1 The number matters less than the fact that it now exists. Day two begins the moment somebody can say *it was 4,000 last month*.

1.2 Usually a KV path under an application's policy. The cost is a diagnosis that takes an hour because nobody can enumerate what is there.

1.3 Extrapolate honestly: an hour of quiet afternoon is not a Monday morning. Record which hour you measured.

1.4 If any of the three requires a credential that lives in Vault, you have found Chapter 15 early.

What a Vault operator knows without looking it up

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

Question	Answer
What day two begins with	the first number you wrote down
Command that counts leases	<code>vault list sys/leases/lookup/</code>
Capability that command needs	<code>list</code> and <code>sudo</code>
What grows without anybody deciding	the audit log
What <code>du</code> counts that secrets do not	old versions and free pages
First field to check when Vault is odd	<code>Sealed</code>
What you cannot measure later	growth you never started measuring
Three things to reach in an incident	status, logs, and the runbook
What must not depend on Vault	the way you reach those three
Difference between day one and day two	nothing new was built

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

Runbook fragment — Monthly measurement

Copy into your own documentation. Fill in the names.

```
Owner: _____
Deputy: _____
Measured monthly, first working day.

vault auth list -detailed      -> mounts: ____
vault list sys/leases/lookup/  -> leases: ____
du -sh <storage path>         -> size:  ____

Compare with last month. Investigate any of:
- lease count up more than 20 % with no new consumer
- storage up with no new consumer
- an auth mount nobody can name the owner of

Escalate to: _____
```

Next: Chapter 2 is the one nobody writes: how you get a Vault into production in the first place — a development instance, a staging one, and the list of things that must be finished before the first real secret goes in.

Chapter 2 — Before the First Real Secret

Why this chapter exists

Every guide starts a Vault. None of them says when you are allowed to put a real secret in it. That gap is where the damage happens. A development instance gets a hostname, somebody stores a database password “just to try it”, a second team hears there is a Vault now, and eleven months later that container is load-bearing and has never been backed up. Nobody decided this. It accumulated.

This chapter is the decision that prevents it: what a development instance is for, what a staging instance proves, and the list that has to be finished before the first secret that matters goes in.

Three instances, three different jobs

Development exists to answer *how does this work*. It is thrown away weekly. It holds nothing anybody would miss.

Staging exists to answer *does our configuration work* — the policies, the auth methods, the applications that fetch from it. It is a rehearsal of production, and its value comes entirely from being configured the same way.

Production exists to answer *is the secret safe and available*. It is the only one of the three where the answer to “can we lose this” is no.

The failure is not having three. It is having one and calling it different names in different conversations.

Do it first. The numbers, the trade-offs and the reasons are after the lab, where the thing you just watched gives them weight.

Lab 2 — Fail your own acceptance check

The cluster from the repository, three nodes, freshly initialised. Chapter 4 explains what the three are for; here they only have to run:

```
$ make reset
$ make up && make init && make unseal
$ . ./scripts/vault-env.sh
```

Step 1 — Write the check

```
$ cat > accept.sh <<'EOF'
pass=0; fail=0
chk(){ printf "  %-46s " "$1"
        if eval "$2" >/dev/null 2>&1
        then echo "PASS"; pass=$((pass+1))
        else echo "FAIL"; fail=$((fail+1)); fi; }

chk "storage is not in-memory" \
  '[ "$(vault status -format=json | jq -r .storage_type)" \
    != "inmem" ]'
chk "at least one audit device" \
  '[ "$(vault audit list -format=json | jq "keys|length")" -ge 1 ]'
chk "initial root token revoked" \
  '! vault token lookup -format=json \
    | jq -e ".data.policies|index(\"root\")"'
echo "  $pass passed, $fail open"
EOF
```

The full version, with all seven, is in the repository as `chapters/ch02/accept.sh`.

Step 2 — Run it against the lab

```
$ ./chapters/ch02/accept.sh
```

```
storage is not in-memory          PASS
more than one node                PASS
TLS in use                       PASS
at least one audit device         FAIL
initial root token revoked        FAIL
a snapshot can be taken          PASS
failure tolerance at least 1      PASS
---
5 passed, 2 open
```

The lab in this book fails its own acceptance check, and it is supposed to. This lab, like every lab in this book, uses the root token out of `init.json` and writes nothing to an audit log until a chapter asks for one. That is fine for a lab and disqualifying for anything else, and now you have the two lines that say so.

Step 3 — Close the first gap

```
$ vault audit enable file file_path=/vault/logs/audit.log
Success! Enabled the file audit device at: file/
```

One command. It is on the list because it is one command that nobody runs, not because it is hard.

Step 4 — Close the second, in the right order

The remaining failure is the root token, and this is the step where people lock themselves out.

Build the replacement first:

```
$ vault policy write admin - <<'POL'
path "*" {
  capabilities = ["create","read","update","delete","list","sudo"]
}
POL
$ vault auth enable userpass
$ vault write auth/userpass/users/opsadmin \
  password=lab-only policies=admin
```

Then prove it works, while you still hold the root token:

```
$ ADMIN=$(vault login -token-only -method=userpass \  
    username=opsadmin password=lab-only)  
$ VAULT_TOKEN=$ADMIN vault token lookup | grep policies  
policies          [admin default]
```

A plain `vault login` would not do here. `vault-env.sh` exported `VAULT_TOKEN`, and the variable beats the token that `vault login` stores — the CLI even says so:

```
WARNING! The VAULT_TOKEN environment variable is set! The value of  
this variable will take precedence; ...
```

So the shell would still be holding root, and the next command would revoke the only token you think you are keeping. Only now, with root still in `VAULT_TOKEN`, revoke it, and switch:

```
$ vault token revoke -self  
$ export VAULT_TOKEN=$ADMIN
```

Run the check again, with the new token:

```
$ ./chapters/ch02/accept.sh  
...  
7 passed, 0 open
```

Note the login syntax. `username=` and `password=` are parameters, not flags — `vault login -method=userpass -username=opsadmin` fails with flag provided but not defined: `-username`, and it fails in the middle of the one procedure where you are about to give up your only way in.

Step 5 — Clean up

Keep first. Before you reset anything, write down the output of `accept.sh` — the line that says how many of the seven passed, and which two did not.

That page is the point of the exercise. The cluster is disposable; what you learned about your own installation is not, and Appendix G is the review that asks for all of them at once.

```
$ make reset
```

What a development server actually is

Run one and ask it:

```
$ vault server -dev
$ vault status -format=json
```

```
{"sealed":false,"storage_type":"inmem",
  "initialized":true,"n":1,"t":1}
```

Six facts in one line, and every one of them disqualifies it:

storage_type: inmem. Memory. Write a secret, restart the container, and:

```
$ vault kv get secret/probe
No value found at secret/data/probe
```

sealed: false with n:1, t:1. One key share, threshold one, and it unsealed itself. There is no ceremony because there is nothing to protect.

No TLS. The listener is plain HTTP

A fixed root token with no expiry. Usually `root`, because somebody put it in an environment variable to save typing.

No audit device. `vault audit list` returns nothing. Whatever happens here leaves no trace.

Mounts you did not create. `cubbyhole/`, `identity/`, `secret/`, `sys/` are there on first boot — which is why examples that write to `secret/` work in dev and fail everywhere else.

A development server is not a small production server. It is a different program wearing the same name, and the distance between them is this chapter.

The list

Before the first real secret, seven things are true or you are not finished. They are checkable, which means they belong in a script rather than a meeting:

Check	Why it is on the list
Storage is not in-memory	Chapter 4
More than one node	Chapter 4
TLS in use	Nothing else on this list matters without it
At least one audit device	Chapter 8 shows what it records
Initial root token revoked	Chapter 8 shows how to get another
A snapshot can be taken	Chapter 9
Failure tolerance at least 1	Chapter 5

All but the TLS row point at a later chapter, which is deliberate: this list is the table of contents for the rest of the book, expressed as things that must be true rather than things to read.

Break it on purpose

Cause each of these, read the real message, then fix it. These are the failures you will meet again, and having produced one deliberately is what makes it recognisable in a second rather than in half an hour.

flag provided but not defined: -username

Cause it: pass the username as a flag, `-username=opsadmin`.

The auth method's parameters are `key=value` arguments after `-method=userpass`, not flags. Get this wrong immediately before revoking the root token and you have a genuine incident; Chapter 8 is then the chapter you need.

The check passes but the audit log is empty

Cause it: enable the file device on a path inside the container that is not mounted from the host. `vault audit list` shows the device is enabled, and nothing appears on the host. The path is inside the container. If the directory is not mounted from the host, the log lives and dies with the container — and Chapter 23 of the first book is where the fail-closed consequence bites.

Staging drifted from production and nobody noticed

Cause it: change one setting in production by hand, then diff the two configurations six months later.

Staging is only worth its cost while its configuration matches. If the two are configured by hand they diverge within a month. The remedy is to stop configuring them by hand — policies, auth methods and mounts as reviewed code, applied to both. Until that exists, the honest position is that staging proves less than you think.

The development instance became production

Cause it: point a second team at the dev instance and wait.

The usual path: a hostname, a second team, a year. There is no technical remedy at that point — the remedy is this chapter’s list, applied to the instance you now have, and the answer will be that most of it fails.

Do this without looking

2.1 Enable an audit device on your own instance and prove it is writing, without trusting the command’s exit code.

2.2 Diff your staging configuration against production and count the differences. Explain each one or remove it.

2.3 Find out who can log in to your development instance, and whether anything in production depends on it.

2.4 Write down the version you are running and where that number came from.

What you should now be able to do

You can now:

- Say what each of the three instances is for, and what it proves
- Name six properties of a development server that disqualify it, and show that none of its data survives a restart
- State the seven conditions that must hold before the first real secret

- Run those conditions as a script rather than discuss them
- Explain why the lab in this book fails two of them on purpose
- Build a non-root administrative path and prove it before giving up the root token

Worked solutions

2.1 Read the file. A device that is enabled and writing nowhere is the failure mode of Chapter 15.

2.2 Zero differences is unusual and worth being suspicious of. The useful outcome is a short list you can defend.

2.3 The uncomfortable answer is common: something in production reads from it, and nobody planned that.

2.4 If the answer is “the wiki”, check the binary.

What a Vault operator knows without looking it up

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

Question	Answer
What proves an audit device works	reading the file
What an exit code proves	that the command returned
What drift is	staging and production disagreeing
How drift starts	one change by hand
What a development instance becomes	production, quietly
Where the version number should come from	the binary
What the wiki knows about your version	what somebody last typed
First thing to write down	who is allowed to change configuration
What a configuration diff is worth	one short list you can defend
Zero differences between environments	is unusual, and worth checking

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

Runbook fragment — Go-live

Copy into your own documentation. Fill in the blanks.

```
Installation: _____
Go-live date: _____
Signed off by: _____

THE SEVEN
[ ] storage is not in-memory      backend: _____
[ ] more than one node           count: _____
[ ] TLS in use                   cert expires: _____
[ ] audit device enabled         path: _____
    and the log is outside the container: yes / no
[ ] initial root token revoked   replacement path: _____
    replacement proven before revocation: yes / no
[ ] snapshot taken and inspected stored at: _____
[ ] failure tolerance >= 1      measured: _____

BEFORE THE FIRST REAL SECRET
[ ] key holders named and reachable (Chapter 8)
[ ] restore rehearsed at least once (Chapter 10)
[ ] who is on call for this: _____
[ ] what breaks if it is down: _____

Result of accept.sh on go-live day: ____ passed, ____ open
If any are open, name them and who accepted the risk:
_____
```

Next: Chapter 3 asks the question this list quietly assumed — that there are enough people. Three key holders, an operator who is not one of them, and somebody who answers at two in the morning is already four, and most teams discover the number by counting who is unreachable.