Architektur: Seal & Unseal

Seal / Unseal

Vault startet sealed: Die Daten sind mit einem Encryption Key (Keyring) verschluesselt, der Keyring mit dem Root Key, der Root Key mit dem Unseal Key. Erst nach dem Unseal kann Vault Requests bedienen.
`vault operator init` teilt den Unseal Key per Shamir's Secret Sharing in Shares auf – Default 5 Shares, Threshold 3. Shares einzeln, in beliebiger Reihenfolge eingeben.

Shamir vs. Auto-Unseal

Shamir (Default): Menschen halten Key-Shares, jeder Neustart braucht das Quorum – operativ teuer, HA-feindlich.
Auto-Unseal: delegiert den Unseal Key an einen vertrauten Dienst (Cloud-KMS, HSM oder Transit-Engine eines zweiten Vault). Vault entsiegelt sich beim Start selbst; fuer Quorum-Operationen (z.B. **`generate-root`**) treten Recovery Keys an die Stelle der Unseal Keys.

Integrated Storage (Raft)

Eingebauter HA-Datastore: repliziert per Raft-Konsens auf alle Server, kein externes Backend noetig. Quorum = Mehrheit der Voting-Nodes, deshalb ungerade Server-Zahl (3 oder 5).
Join ueber **`retry_join`**-Stanza in der Config oder **`vault operator raft join`** – Nodes einzeln joinen und Health abwarten.

```
vault operator init     # 5 Shares, Threshold 3
vault operator unseal   # 3x, je ein Share
vault status
vault operator raft list-peers
vault operator raft snapshot save backup.snap
```

Secrets Engines

KV v2: versionierte Secrets

Jeder Key traegt Versionen (**`max_versions`** pro Mount/Key). Soft Delete macht eine Version unzugaenglich, aber wiederherstellbar (**`undelete`**); destroy loescht Versionsdaten endgueltig. **`cas`** (Check-and-Set) verhindert Lost Updates.
API-Pfade je Mount: **`data/`**, **`metadata/`**, **`delete/`**, **`destroy/`** – wichtig fuer Policies!

```
vault secrets enable -version=2 kv
vault kv put    -mount=secret app/db user=svc pass=s3
vault kv get    -mount=secret -version=2 app/db
vault kv patch  -mount=secret app/db pass=neu
vault kv undelete -mount=secret -versions=2 app/db
vault kv destroy  -mount=secret -versions=1 app/db
```

Database: dynamische Credentials

Vault erzeugt DB-Accounts on demand: Jede App-Instanz bekommt eigene, kurzlebige Credentials mit Lease/TTL; nach Ablauf revoked Vault den Account. Role mappt auf **`creation_statements`** mit **`{{username}}`**/**`{{password}}`**-Platzhaltern.
Static Roles rotieren bestehende Accounts nach Zeitplan. Nach dem Setup rotate-root: das Bootstrap-Passwort kennt danach nur noch Vault.

```
vault read database/creds/readonly  # user+pass+lease
vault write -force database/rotate-root/pg
```

PKI: Zertifikate on demand

X.509-Zertifikate ohne manuellen CSR-Prozess – pro Workload-Instanz ein eigenes, kurzlebiges Zertifikat. Kurze TTLs statt Revocation: haelt CRLs klein.
Aufbau: Root-CA und Intermediate-CA getrennt aufsetzen, Issuance laeuft ueber die Intermediate. Roles begrenzen **`allowed_domains`**. ACME wird als Protokoll unterstuetzt.

Transit: Encryption as a Service

Ver-/Entschluesselung als API – Vault speichert die Daten nicht. Ciphertext traegt die Key-Version als Prefix (**`vault:v1:...`**).
rotate erzeugt eine neue Key-Version, rewrap hebt Ciphertext ohne Plaintext-Zugriff auf die neue Version, **`min_decryption_version`** sperrt Alt-Versionen. Dazu: sign/verify, HMAC, Datakeys fuer Envelope Encryption.

```
vault write transit/encrypt/app \
  plaintext=$(base64 <<< "geheim")
vault write -f transit/keys/app/rotate
vault write transit/rewrap/app ciphertext=vault:v1:...
```

Auth-Methoden

kubernetes

Pod authentisiert sich mit seinem ServiceAccount-JWT; Vault validiert es gegen die TokenReview-API des Clusters. Config: **`kubernetes_host`**, **`kubernetes_ca_cert`**, **`token_reviewer_jwt`**. Die Role bindet **`bound_service_account_names`** + **`..._namespaces`** an Policies.

```
vault auth enable kubernetes
vault write auth/kubernetes/config \
  kubernetes_host=https://10.0.0.1:443 \
  kubernetes_ca_cert=@ca.crt
vault write auth/kubernetes/role/app \
  bound_service_account_names=myapp \
  bound_service_account_namespaces=prod \
  token_policies=app-ro token_ttl=1h
```

AppRole

Fuer Maschinen/CI: RoleID (quasi Username) + SecretID (geheim, Pull-Mode empfohlen). SecretID mit **`secret_id_ttl`** und **`secret_id_num_uses`** begrenzen; Zustellung an die App ueber Response Wrapping (kurzlebiges Wrapping-Token) statt Klartext. **`token_type=batch`** empfohlen.

OIDC / JWT

Zwei Modi einer Methode: oidc = interaktiver Redirect-Flow (Authorization Code + PKCE, **`oidc_discovery_url`** + Client-ID/-Secret) fuer Menschen; jwt = Bearer-JWT gegen JWKS/statische Keys fuer Workloads/CI.
`bound_audiences` muss exakt einem **`aud`**-Claim entsprechen; **`bound_claims`** prueft weitere Claims; **`user_claim`** bestimmt den Entity-Alias.

Policies (HCL)

Deny by default

Policies (HCL oder JSON) beschreiben erlaubte Pfade – leere Policy = kein Zugriff. Capabilities: **`create`**, **`read`**, **`update`**, **`patch`**, **`delete`**, **`list`**, **`sudo`** (root-protected Paths), **`deny`** (gewinnt immer).
`default`-Policy haengt an jedem Token; **`root`**-Policy kann alles – nie an Alltags-Tokens.

```
# app-ro.hcl -- KV v2 braucht data/-Prefix!
path "secret/data/app/*" {
  capabilities = ["read"]
}
path "secret/metadata/app/*" {
  capabilities = ["list"]
}
```

Globs

`*` nur am Pfadende = Prefix-Match (**`secret/data/bar/*`**).
`+` matcht genau ein Segment: **`secret/data/+/team`** trifft **`secret/data/foo/team`**.
`vault policy write app-ro app-ro.hcl` laedt die Policy.

Token-Lifecycle

Service vs. Batch

Service-Tokens (**`hvs.`**): renewbar, revozierbar, Accessor, koennen Child-Tokens erzeugen – Zustand im Storage.
Batch-Tokens (**`hvb.`**): verschluesselter Blob ohne Storage-Eintrag, nicht renewbar, kein Child – leichtgewichtig fuer hohe Last.

TTL, Renewal, Orphan

Jedes Nicht-Root-Token hat eine TTL; Renewal verlaengert bis zur Max-TTL (System-Default 32 Tage, konfigurierbar). Periodic Tokens leben beliebig lang, solange sie pro Periode renewed werden; explicit max TTL setzt eine harte Grenze.
Revoke eines Tokens revoked seine Child-Tokens mit – Orphan-Tokens haben keinen Parent und ueberleben.

```
vault token create -policy=app-ro -ttl=1h
vault token renew  <token>
vault token lookup            # eigenes Token
vault token revoke -accessor <accessor>
```

Kubernetes-Integration

Agent Injector (Annotations)

Mutating Webhook injiziert einen Vault-Agent-Sidecar; Secrets landen als Dateien unter **`/vault/secrets/`**. Templates rendern beliebige Formate. **`agent-pre-populate-only`** fuer Jobs/CronJobs (nur Init-Container, Pod terminiert sauber).

```
metadata:
  annotations:
    vault.hashicorp.com/agent-inject: "true"
    vault.hashicorp.com/role: "app"
    vault.hashicorp.com/agent-inject-secret-db:
      "secret/data/app/db"
    vault.hashicorp.com/agent-inject-template-db: |
      {{- with secret "secret/data/app/db" -}}
      pg://{{ .Data.data.user }}:{{ .Data.data.pass }}@db
      {{- end -}}
```

Vault CSI Provider

Secrets-Store-CSI-Driver mountet Secrets als Volume (**`SecretProviderClass`**), Auth ueber den Pod-ServiceAccount (kubernetes/JWT). Optionaler Sync in K8s-Secrets fuer Env-Vars. Alle Secrets Engines nutzbar.

VSO: Vault Secrets Operator

Operator synct Vault-Secrets in native K8s-Secrets: CRDs **`VaultConnection`**, **`VaultAuth`**, **`VaultStaticSecret`**, **`VaultDynamicSecret`**, **`VaultPKISecret`**. Remediert Drift am Ziel-Secret, rotiert und kann Deployments per **`rolloutRestartTargets`** neu ausrollen.

```
apiVersion: secrets.hashicorp.com/v1beta1
kind: VaultStaticSecret
metadata: {name: app-db, namespace: prod}
spec:
  vaultAuthRef: app-auth
  mount: secret
  type: kv-v2
  path: app/db
  refreshAfter: 60s
  destination: {name: app-db, create: true}
```

Audit Devices

Pflicht fuer Prod

Typen: file, syslog, socket – loggen jeden Request/Response. Sensible String-Werte stehen default nur als HMAC-SHA256-Hash im Log.

Blocking-Verhalten: Kann Vault auf kein aktiviertes Audit Device schreiben, beantwortet es keine Requests mehr – deshalb mindestens zwei Devices auf getrennten Pfaden.

```
vault audit enable file \
  file_path=/var/log/vault_audit.log
vault audit list
```

Anti-Patterns

Was du nicht tun solltest

Root-Token im Alltag: nur fuer Setup/Break-Glass; danach revoken (**`vault token revoke`**) und bei Bedarf per **`generate-root`** + Quorum neu erzeugen.

KV als Konfig-Store: Vault verwaltet Secrets, kein allgemeines App-Config-Management – Nicht-Geheimes gehoert in ConfigMaps/Git, sonst waechst Policy- und Audit-Flaeche ohne Gewinn.

Policies auf **`secret/app`** statt **`secret/data/app`**: KV-v2-Pfad-Falle, die Policy greift nie.

Leases ignorieren: dynamische DB-Credentials cachen und nach Lease-Ablauf weiterverwenden bricht in Prod.

Kein/ein Audit Device: blind im Incident bzw. Blocking-Risiko – immer zwei.

Shamir-Shares bei einer Person: hebelt das Threshold-Konzept aus; Shares auf Rollen verteilen oder Auto-Unseal nutzen.

Sealed Vault als Ausnahme behandeln: Restart heisst sealed – ohne Auto-Unseal steht HA-Failover, bis das Quorum tippt.

vault-cheatsheet.de

Vault Cheatsheet von OMNI52

Weiterfuehren, nicht selbst neu erfinden

Vault-Rollout & Secrets-Architektur

OMNI52™ hilft Plattform-Teams, Vault fuer regulierte Enterprise-Umgebungen sauber aufzustellen.

HA-Design mit Integrated Storage (Raft) und Auto-Unseal, Auth-Anbindung an Kubernetes und OIDC, Policy-Design nach Least-Privilege, Audit-Pipeline mit zwei Devices, K8s-Integration via Vault Secrets Operator oder Agent Injector. Output landet als GitOps-Repo bei euch im Cluster: Helm-Charts, Policies, Runbooks. Euer Team operiert weiter, nicht wir.

OMNI52 GmbH, Ebern · istio-consulting.de · kostenloses 30-min Initial-Assessment

Aus der OMNI52 Cheatsheet-Fabrik

Verwandte Sheets: kubernetes-cheatsheet.de (Kubernetes Core) · istio-cheatsheet.de (Service-Mesh-Layer) · rancher-cheatsheet.de (Rancher-Management) · rke2-cheatsheet.de (RKE2-Distribution).