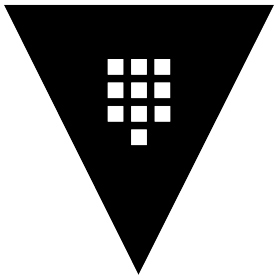
Mastering Vault HashiCorp

Complete Guide

HashiCorp Vault



MARYEM CHERIF

Network Security Engineer

Contents

Chapter 1

Introduction à HashiCorp Vault

1.1 Qu'est-ce que Vault?

HashiCorp Vault est une solution moderne de gestion des secrets et de protection des données sensibles. Dans les environnements DevOps et de cybersécurité, la gestion des secrets (mots de passe, clés API, certificats, tokens) est un défi critique. Vault apporte une approche centralisée, sécurisée et automatisée pour stocker, distribuer et contrôler l'accès aux informations sensibles.

1.1.1 Pourquoi utiliser Vault?

Sans un outil comme Vault, les organisations sont confrontées à de nombreux problèmes de sécurité et de gestion opérationnelle.

Problèmes fréquents sans Vault

- Secrets codés en dur dans le code source, ce qui augmente le risque de fuite
- Partage de mots de passe par email ou chat, non sécurisé et difficilement traçable
- Absence de rotation automatique des secrets, entraînant une exposition prolongée
- Pas de traçabilité des accès aux secrets, compliquant l'audit et la conformité
- Gestion manuelle sujette aux erreurs humaines

Vault résout ces problèmes en offrant des mécanismes robustes pour la **sécurité, la rotation et l'audit des secrets**.

Solutions apportées par Vault

- **Stockage centralisé et sécurisé** : Tous les secrets sont conservés dans un emplacement unique et chiffré.
- **Chiffrement automatique** : Les données sont chiffrées au repos et en transit, assurant leur confidentialité.
- **Rotation automatique** des credentials : Réduit le risque d'exploitation en cas de fuite.
- **Audit logging complet** : Toutes les requêtes et réponses sont enregistrées pour la traçabilité et la conformité.
- **Contrôle d'accès granulaire (ACL)** : Les utilisateurs et applications n'ont accès qu'aux secrets nécessaires.
- **Génération dynamique de secrets** : Création de secrets à durée de vie limitée pour minimiser l'exposition.

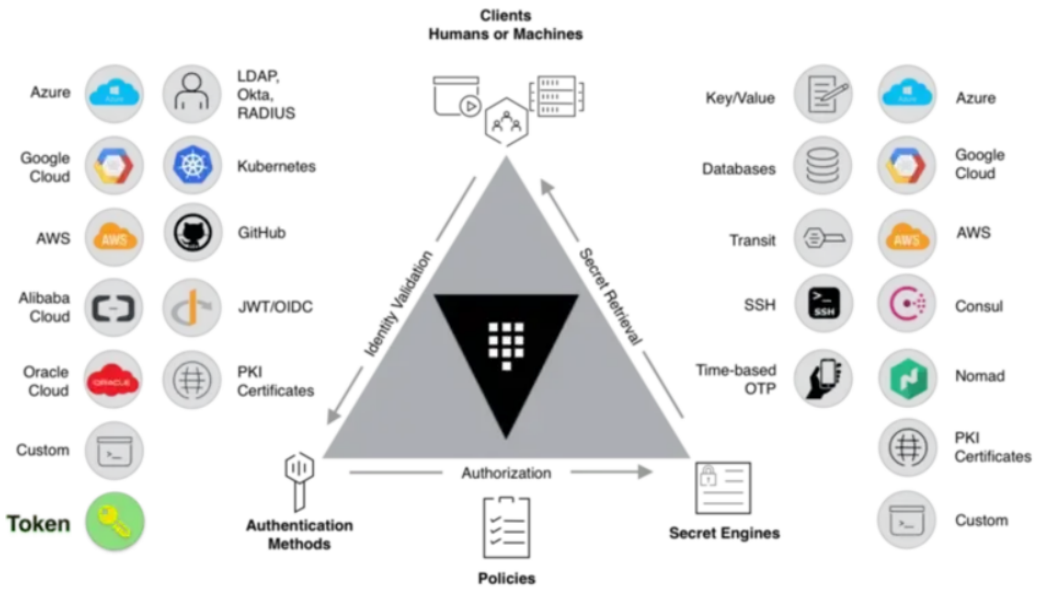


Figure 1.1: Vue d'ensemble de l'architecture de HashiCorp Vault

1.2 Architecture de Vault

Vault repose sur une architecture modulaire et sécurisée, permettant à la fois la haute disponibilité, la résilience et la protection des données.

1.2.1 Composants principaux

- **Storage Backend** : Fournit le stockage persistant des données chiffrées (ex. Consul, etcd, filesystem). Il assure la durabilité et la résilience des secrets.

- **Barrier** : Couche de chiffrement qui protège toutes les données et empêche l'accès tant que Vault est scellé.
- **Secrets Engines** : Modules spécialisés qui stockent, génèrent ou chiffrent différents types de secrets (KV, Database, AWS, Transit, etc.).
- **Auth Methods** : Méthodes d'authentification pour valider l'identité des utilisateurs ou applications (tokens, LDAP, AWS, Kubernetes, AppRole).
- **Audit Devices** : Enregistre tous les accès et opérations sur Vault pour garantir la traçabilité et faciliter les audits.
- **Policies** : Règles de contrôle d'accès définissant qui peut faire quoi et sur quel secret.

1.2.2 Concepts clés

Seal / Unseal : Vault démarre toujours dans un état "sealed" (scellé), où toutes les données sont chiffrées et inaccessibles. Pour le rendre opérationnel, il faut le "unseal" avec un nombre requis de clés de déchiffrement, garantissant ainsi la sécurité des données même en cas de compromission du serveur.

Root Token : Jeton initial possédant tous les privilèges, utilisé uniquement pour les opérations critiques et la configuration initiale. Il doit être révoqué après l'initialisation pour limiter les risques.

Lease : Durée de vie assignée à chaque secret. Après expiration, le secret est automatiquement révoqué, réduisant la fenêtre d'exposition et améliorant la sécurité globale.

Chapter 2

Installation et Configuration

2.1 Prérequis

- Système Linux/MacOS/Windows
- Docker (optionnel mais recommandé pour les tests)
- 1 GB RAM minimum
- Connaissances de base en ligne de commande

2.2 Installation de Vault

- **Documentation officelle installation sur :** https://developer.hashicorp.com/vault/install

2.2.1 Installation sur Linux

Listing 2.1: Installation de Vault sur Linux

```
# T l charger Vault
wget https://releases.hashicorp.com/vault/1.15.0/vault_1.15.0_linux_amd64.
    zip

# D compresser
unzip vault_1.15.0_linux_amd64.zip

# D placer dans le PATH
sudo mv vault /usr/local/bin/

# V rifier l'installation
vault version
```

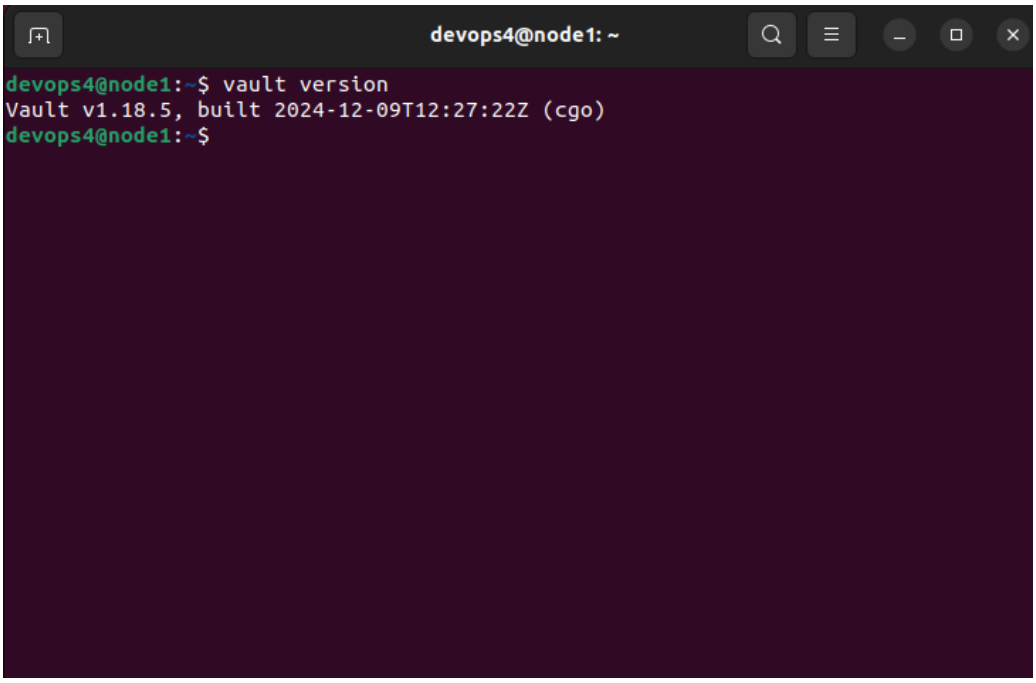


Figure 2.1: Sortie de la commande vault version

2.2.2 Installation avec Docker

Listing 2.2: Lancement de Vault avec Docker

```
# Mode d veloppement (NON PRODUCTION!)
docker run --cap-add=IPC_LOCK -d --name=vault-dev \
  -p 8200:8200 \
  -e 'VAULT_DEV_ROOT_TOKEN_ID=myroot' \
  hashicorp/vault:latest

# V rifier que le conteneur fonctionne
docker ps
```

2.3 Démarrage de Vault en mode Dev

Attention

Le mode développement est UNIQUEMENT pour les tests. Ne jamais utiliser en production!

- Le serveur démarre unsealed
- Stockage en mémoire (perte des données au redémarrage)
- Un seul unsealed key
- Communication HTTP (non chiffrée)

Listing 2.3: Démarrage de Vault en mode développement

```
# D marrer le serveur en mode dev
```

```
vault server -dev

# Dans un autre terminal, configurer les variables d'environnement
export VAULT_ADDR='http://127.0.0.1:8200'
export VAULT_TOKEN='root'  # Token affich  au d marrage

# Tester la connexion
vault status
```

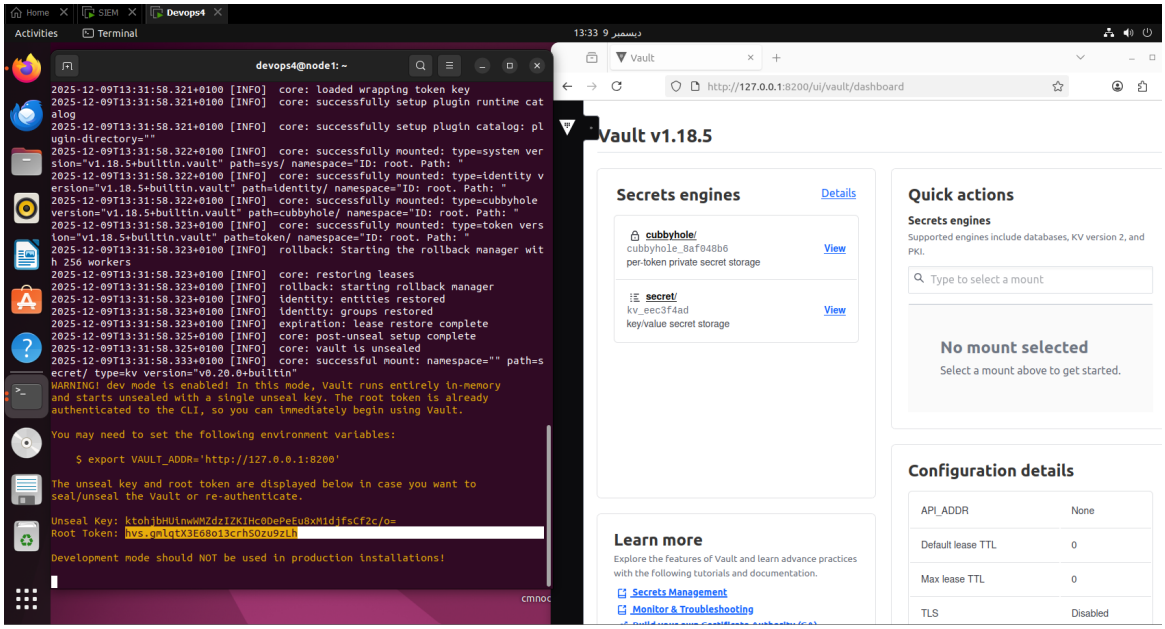


Figure 2.2: Démarrage du serveur Vault en mode développement

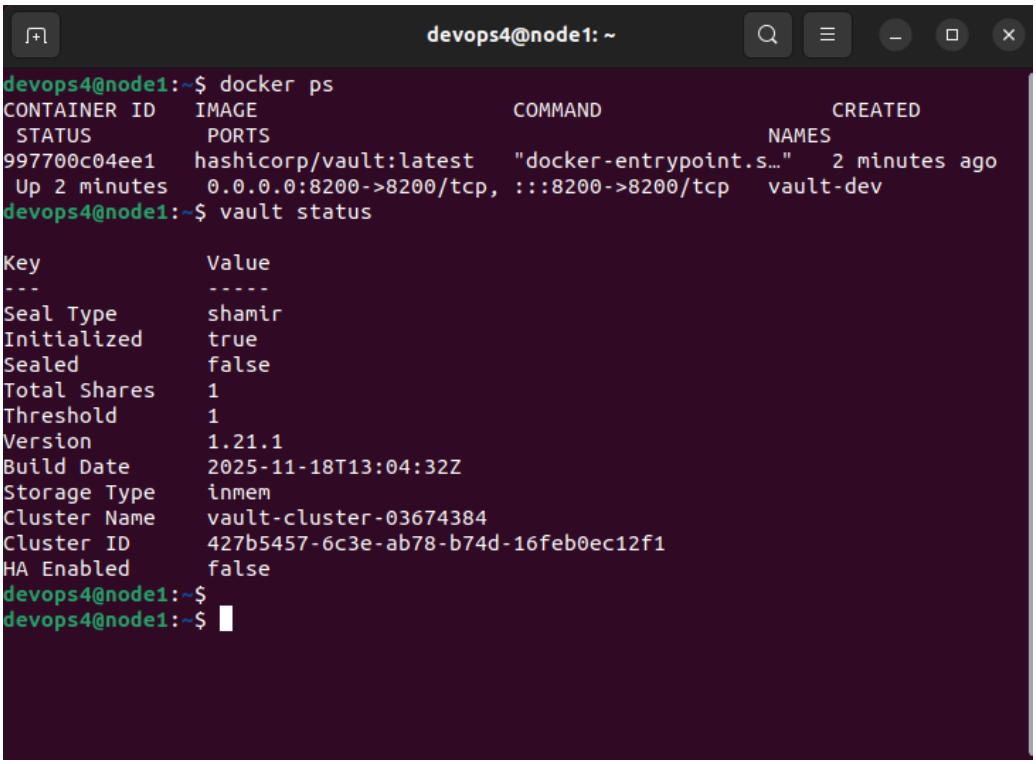


Figure 2.3: Sortie de la commande vault status

Chapter 3

Premiers Pas avec Vault

3.1 Utilisation de l'interface CLI

3.1.1 Écrire et lire des secrets

Listing 3.1: Opérations de base avec les secrets

```
#  crire  un secret
vault kv put secret/myapp/config \
    username=admin \
    password=supersecret123

# Lire un secret
vault kv get secret/myapp/config

# Lire un champ sp cifique
vault kv get -field=password secret/myapp/config

# Lister les secrets
vault kv list secret/myapp/
```

```
devops4@node1:~$ vault kv put secret/myapp/config username=admin password=supers
ecret123
====== Secret Path ======
secret/data/myapp/config

======= Metadata =======
Key                Value
---                -----
created_time       2025-12-09T13:36:59.332344797Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1
```

Figure 3.1: vault kv put output

```
devops4@node1:~$ vault kv get secret/myapp/config
====== Secret Path ======
secret/data/myapp/config

======= Metadata =======
Key                Value
---                -----
created_time       2025-12-09T13:36:59.332344797Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1

====== Data ======
Key         Value
---         -----
password    supersecret123
username    admin
```

Figure 3.2: Lecture d'un secret avec vault kv get

3.1.2 Mettre à jour et supprimer des secrets

Listing 3.2: Mise à jour et suppression de secrets

```
# Mettre    jour (ajouter/modifier des champs)
vault kv patch secret/myapp/config \
    api_key=abc123xyz

# Supprimer un secret
vault kv delete secret/myapp/config

# Voir l'historique des versions (KV v2)
vault kv metadata get secret/myapp/config

# Restaurer une version pr c dente
vault kv undelete -versions=1 secret/myapp/config
```

3.2 Interface Web UI

Vault dispose d'une interface web accessible par défaut sur `http://127.0.0.1:8200/ui`

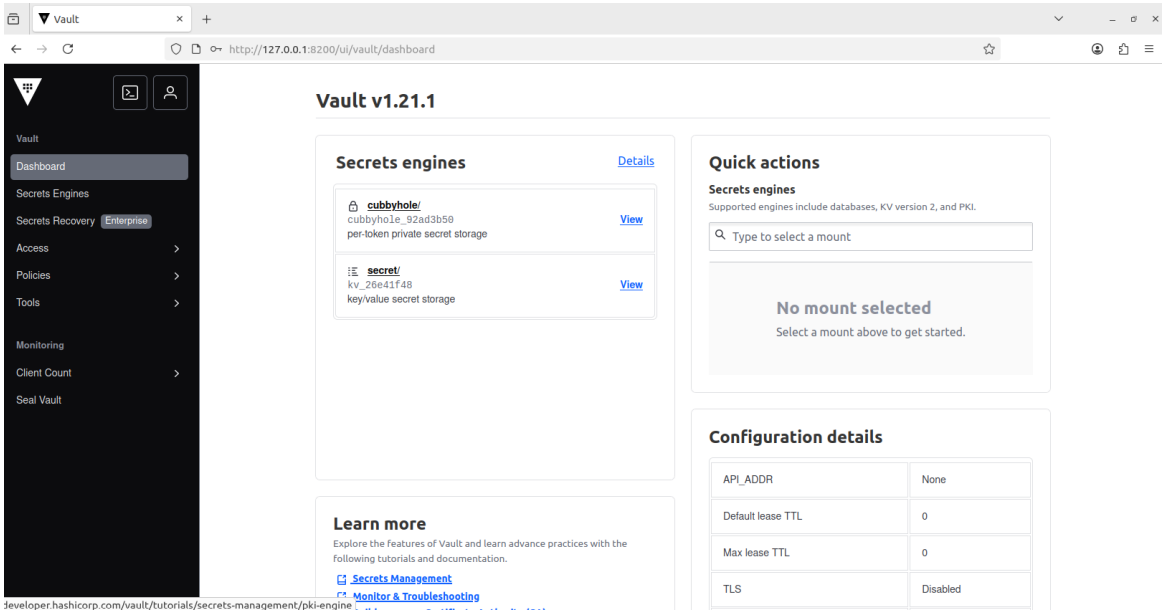


Figure 3.3: Dashboard principal de l'interface Web

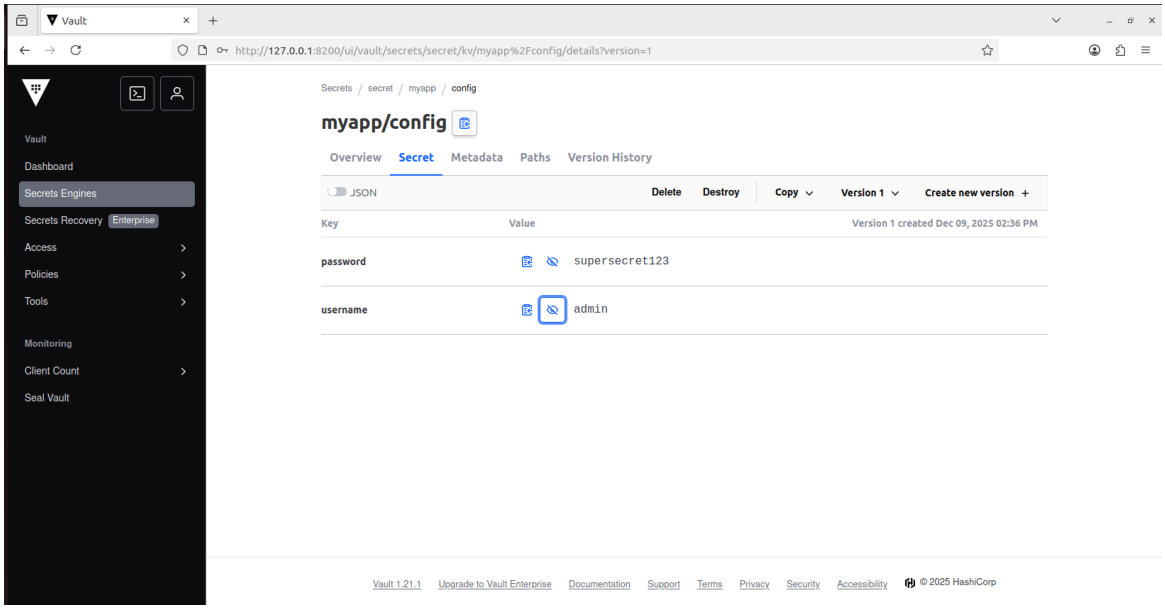


Figure 3.4: Gestion des secrets dans l'interface Web

Chapter 4

Secrets Engines

4.1 Introduction aux Secrets Engines

Les Secrets Engines sont des composants Vault qui stockent, génèrent ou chiffrent des données. Chaque engine est monté à un chemin spécifique.

4.1.1 Types de Secrets Engines

- **KV (Key-Value)**: Stockage simple de paires clé-valeur
- **Database**: Génération dynamique de credentials pour bases de données
- **AWS**: Génération dynamique de credentials AWS
- **PKI**: Génération et gestion de certificats X.509
- **Transit**: Chiffrement as-a-service
- **SSH**: Génération de certificats SSH

4.2 KV Secrets Engine

4.2.1 KV Version 1 vs Version 2

Fonctionnalité	**KV v1**	**KV v2**
Versioning (historique des secrets)	Non	Oui
Soft delete (suppression temporaire)	Non	Oui
TTL sur secrets	Non	Oui
Check-and-Set (verrouillage de version)	Non	Oui

Table 4.1: Comparaison KV v1 et KV v2

Listing 4.1: Activer KV v2

```
# Activer KV v2
vault secrets enable -path=secret-v2 kv-v2

```

```
# Ajouter un secret avec m tadonn es
vault kv put secret-v2/prod/database username=dbadmin password=
   SecurePass123 max_ttl=24h

# Lire le secret
vault kv get secret-v2/prod/database

# Voir l'historique des versions
vault kv metadata get secret-v2/prod/database
```

4.3 Database Secrets Engine

Le Database Secrets Engine permet de créer des credentials dynamiques pour des bases de données comme PostgreSQL.

Principe :

- Vault génère automatiquement des comptes avec un mot de passe temporaire.
- Chaque application/utilisateur a ses propres credentials.
- Rotation et révocation automatiques.

4.3.1 Configuration pour PostgreSQL

Listing 4.2: Configuration du Database Secrets Engine avec PostgreSQL

```
#!/bin/bash

# --- Variables ---
VAULT_ADDR='http://127.0.0.1:8200'
VAULT_TOKEN='myroot'
DB_USER='vaultadmin'
DB_PASS='vaultpassword'
DB_NAME='mydb'
ROLE_NAME='readonly'

# --- Export Vault env variables ---
export VAULT_ADDR
export VAULT_TOKEN

# --- Installer PostgreSQL si n cessaire ---
if ! command -v psql > /dev/null; then
    echo "PostgreSQL non trouv . Installation..."
    sudo apt update
    sudo apt install postgresql postgresql-contrib -y
fi

# --- Cr er utilisateur PostgreSQL et base ---
sudo -u postgres psql <<EOF
-- Cr er l'utilisateur Vault
CREATE ROLE $DB_USER LOGIN PASSWORD '$DB_PASS' CREATEROLE;

-- Cr er la base
CREATE DATABASE $DB_NAME;

```

```
-- Donner acc s    la base et aux tables
GRANT CONNECT ON DATABASE $DB_NAME TO $DB_USER;
\c $DB_NAME
GRANT USAGE ON SCHEMA public TO $DB_USER;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO $DB_USER;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO
   $DB_USER;
EOF

# --- Activer le Database Secrets Engine ---
vault secrets enable database

# --- Configurer la connexion PostgreSQL pour Vault ---
vault write database/config/postgresql \
    plugin_name=postgresql-database-plugin \
    allowed_roles="$ROLE_NAME" \
    connection_url="postgresql://{{username}}:{{password}}@127.0.0.1:5432/
       $DB_NAME" \
    username="$DB_USER" password="$DB_PASS"

# --- Cr er un r le Vault pour g n rer des credentials dynamiques ---
vault write database/roles/$ROLE_NAME \
    db_name=postgresql \
    creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{
       password}}' VALID UNTIL '{{expiration}}'; GRANT SELECT ON ALL
       TABLES IN SCHEMA public TO \"{{name}}\";" \
    default_ttl="1h" max_ttl="24h"

# --- G n rer un credential dynamique pour test ---
vault read database/creds/$ROLE_NAME
```

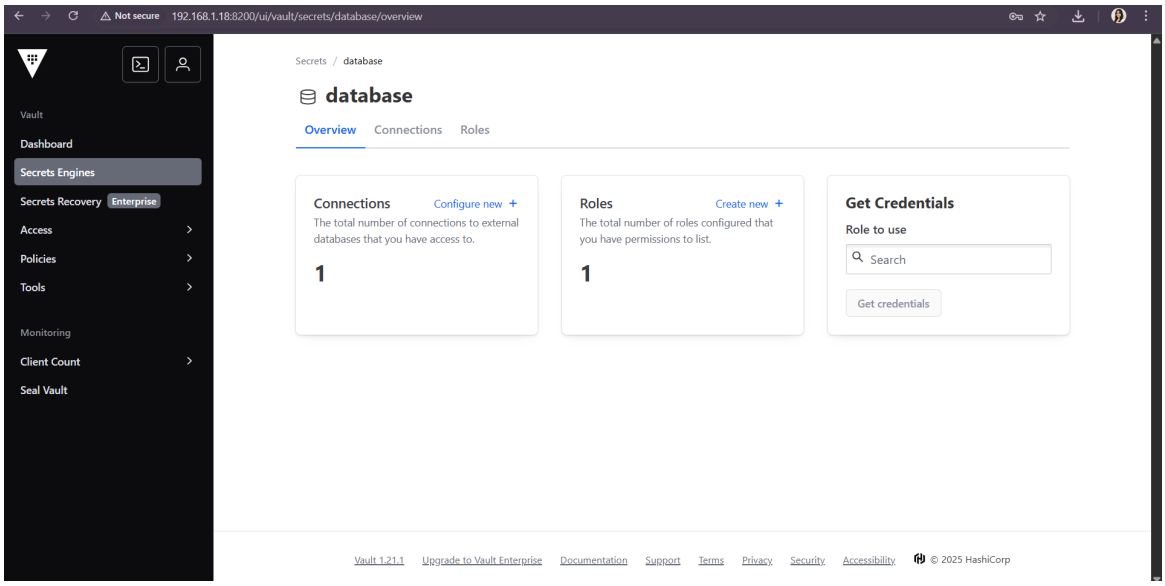


Figure 4.1: Génération de credentials dynamiques pour PostgreSQL

4.3.2 Avantages des credentials dynamiques

Bénéfices

- Credentials uniques par application/utilisateur
- Rotation automatique à l'expiration
- Révocation immédiate possible
- Traçabilité complète des accès
- Réduction de la surface d'attaque

Chapter 5

Méthodes d'Authentification

5.1 Introduction aux Auth Methods

Vault supporte de nombreuses méthodes d'authentification pour s'adapter à différents environnements.

5.1.1 Auth methods courantes

- **Token**: Authentification par token (méthode de base)
- **UserPass**: Nom d'utilisateur et mot de passe
- **LDAP/AD**: Intégration avec Active Directory
- **AppRole**: Pour les applications et automatisation
- **Kubernetes**: Pour les pods Kubernetes
- **AWS**: Authentification via IAM ou EC2
- **GitHub**: Via les teams et organisations GitHub

5.2 Token Authentication

Listing 5.1: Gestion des tokens

```
#Cr er un token avec une politique sp cifique
vault token create -policy=myapp-policy

# Cr er un token avec TTL limit  (Limiter sa dur e de vie)
vault token create -ttl=1h -policy=readonly

# R voquer ou renouveler un token.
vault token revoke s.abc123xyz

# Lister les tokens actifs (root requis)
vault list auth/token/accessors

# Renouveler un token
vault token renew
```

5.3 UserPass Authentication

Listing 5.2: Configuration UserPass

```
# Activer le userpass auth
vault auth enable userpass

# Cr er un utilisateur avec des policies
vault write auth/userpass/users/john \
    password=supersecret \
    policies=developer

# Se connecter avec l'utilisateur
vault login -method=userpass \
    username=john \
    password=supersecret

# Changer le mot de passe si n cessaire.
vault write auth/userpass/users/john/password \
    password=newsecretpassword
```

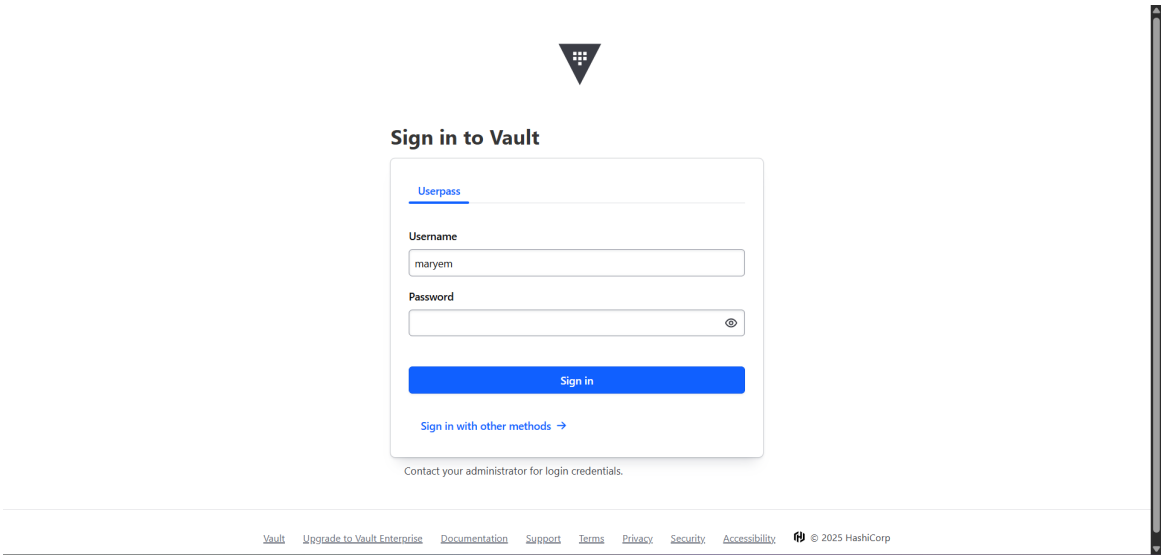


Figure 5.1: Authentification avec UserPass

5.4 AppRole Authentication

AppRole est la méthode recommandée pour les applications et l'automatisation.

Listing 5.3: Configuration AppRole Authentication dans Vault

```
#!/bin/bash
# ------------------------------
# Script explicatif pour AppRole
# ------------------------------

# --- Exporter les variables Vault ---
export VAULT_ADDR='http://127.0.0.1:8200'
export VAULT_TOKEN='myroot'  # token root pour config initiale

```

```
# --- Activer l'authentification AppRole ---
vault auth enable approle
# Cette commande active le backend AppRole pour que les applications puissent s'authentifier.

# --- Cr er un r le AppRole pour l'application ---
vault write auth/approle/role/my-application \
    secret_id_ttl=10m \          # dur e de vie du SecretID
    token_num_uses=10 \          # nombre de fois que le token peut  tre utilis
    token_ttl=20m \              # TTL du token g n r
    token_max_ttl=30m \          # TTL max du token
    secret_id_num_uses=40 \      # nombre de fois que le SecretID peut tre  utilis  pour login
    policies=myapp-policy        # attache la politique de permissions au token

# --- R cup rer le RoleID du r le cr  ---
vault read auth/approle/role/my-application/role-id
# Le RoleID est constant et servira avec le SecretID pour l'authentification.

# --- G n rer un SecretID temporaire ---
vault write -f auth/approle/role/my-application/secret-id
# Le SecretID est secret et combin  avec le RoleID pour login.

# --- Se connecter avec RoleID et SecretID ---
vault write auth/approle/login \
    role_id=xxxx-xxxx-xxxx \
    secret_id=yyyy-yyyy-yyyy
# Remplace xxxx et yyyy par les valeurs r cup r es.
# La sortie est un token d'acc s que l'application peut utiliser pour lire des secrets.

# --- Exemple d'utilisation du token pour lire un secret ---
export VAULT_TOKEN=s.abc123xyz
vault kv get secret/myapp/config
```

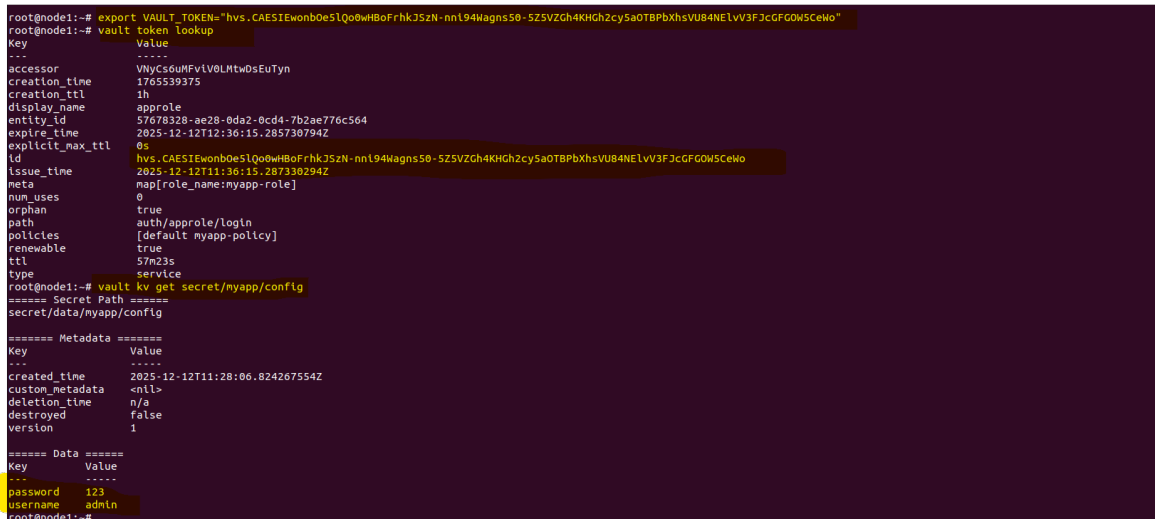


Figure 5.2: Workflow d'authentification AppRole

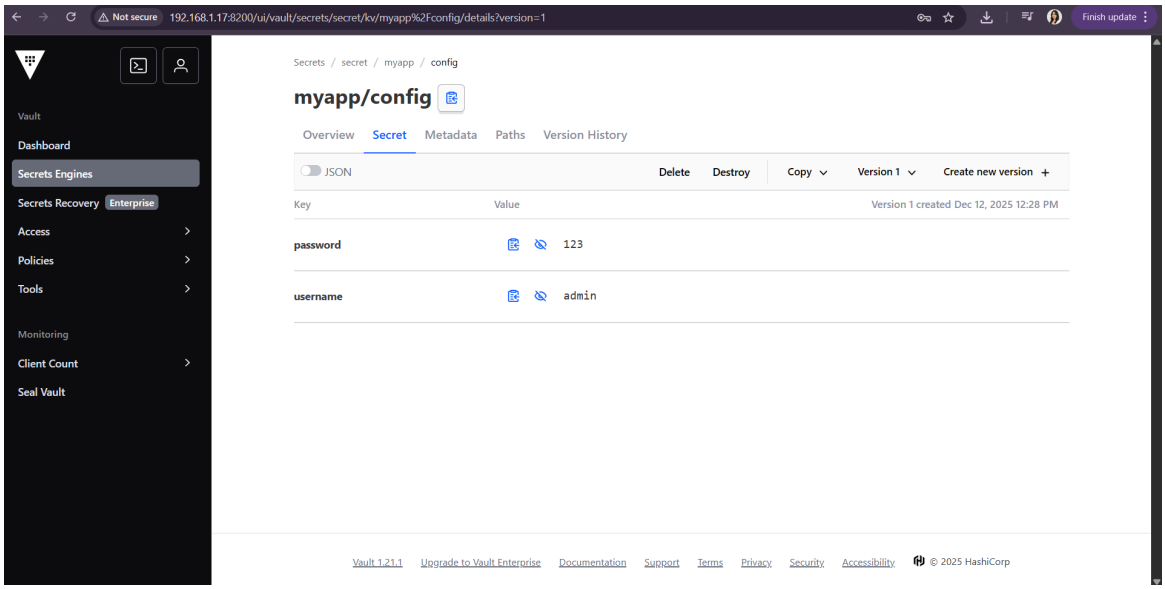


Figure 5.3: Workflow d'authentification AppRole 1

Chapter 6

Policies et Contrôle d'Accès

6.1 Comprendre les Policies

Les policies Vault utilisent le langage HCL (HashiCorp Configuration Language) pour définir les permissions.

6.1.1 Structure d'une policy

Listing 6.1: Exemple de policy Vault

```
# Policy pour une application de lecture seule
path "secret/data/myapp/*" {
  capabilities = ["read", "list"]
}

path "database/creds/readonly" {
  capabilities = ["read"]
}

# Acc s complet    un chemin sp cifique
path "secret/data/myapp/config" {
  capabilities = ["create", "read", "update", "delete", "list"]
}

# D nier explicitement l'acc s
path "secret/data/admin/*" {
  capabilities = ["deny"]
}
```

6.1.2 Capabilities disponibles

- **create**: Créer de nouvelles données
- **read**: Lire les données existantes
- **update**: Mettre à jour les données
- **delete**: Supprimer les données
- **list**: Lister les clés/chemins

- **deny**: Refuser explicitement l'accès
- **sudo**: Opérations privilégiées

6.2 Créer et appliquer des Policies

Listing 6.2: Gestion des policies

```
# Cr er une policy    partir d'un fichier
vault policy write myapp-policy myapp-policy.hcl

# Lire une policy
vault policy read myapp-policy

# Lister toutes les policies
vault policy list

# Supprimer une policy
vault policy delete myapp-policy

# Tester une policy (dry-run)
vault token create -policy=myapp-policy
```

6.2.1 Policy pour développeur

Listing 6.3: developer-policy.hcl

```
# Acc s complet aux secrets de d veloppement
path "secret/data/dev/*" {
  capabilities = ["create", "read", "update", "delete", "list"]
}

# Lecture seule pour les secrets de staging
path "secret/data/staging/*" {
  capabilities = ["read", "list"]
}

# Pas d'acc s    la production
path "secret/data/prod/*" {
  capabilities = ["deny"]
}

# G n ration de credentials DB de dev
path "database/creds/dev-readonly" {
  capabilities = ["read"]
}
```

Listing 6.4: Appliquer la policy développeur

```
# Cr er la policy
vault policy write developer developer-policy.hcl

# Cr er un utilisateur avec cette policy
vault write auth/userpass/users/alice \
    password=dev123 \
    policies=developer
```

```
8
9  # Tester avec le compte alice
10 vault login -method=userpass username=alice password=dev123
11 vault kv get secret/dev/app-config  # OK
12 vault kv get secret/prod/app-config # DENIED
```

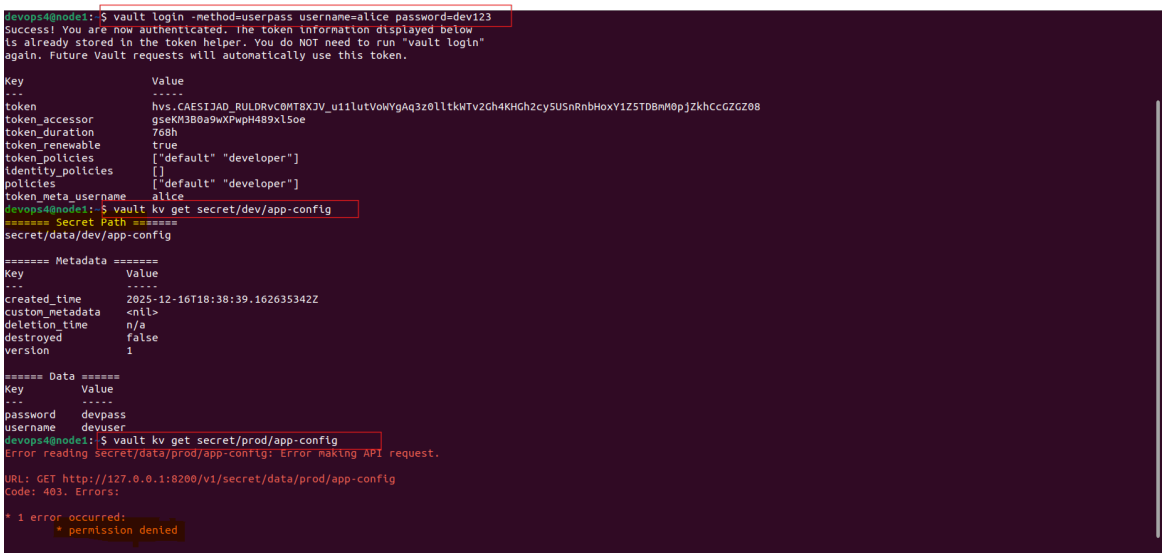


Figure 6.1: Test des permissions avec différentes policies

Chapter 7

Configuration Production

Cette partie montre comment Vault fonctionne en mode production, contrairement au mode *dev*. L'objectif de cette section est de :

- comprendre comment Vault stocke réellement les secrets,
- voir le processus d'initialisation,
- générer les *unseal keys* et le *root token*,
- apprendre à déverrouiller Vault manuellement.

En production :

- Vault ne démarre pas automatiquement en mode non scellé,
- Vault est scellé (*sealed*) au démarrage,
- l'accès dépend du déverrouillage manuel via les *unseal keys*.

7.1 Différences Dev vs Production

Aspect	**Dev**	**Production**
Stockage	In-memory	File / Consul / Cloud
TLS	Non obligatoire	Obligatoire
Déverrouillage	Auto	Manuel (Shamir / Auto-unseal)
HA (High Availability)	Non	Oui (cluster)
Audit	Non	Obligatoire

Table 7.1: Différences entre mode Dev et Production

7.2 Configuration du serveur

Voici la configuration utilisée pour le serveur Vault en production. Nous avons choisi un stockage local (*file*) et désactivé TLS pour nos tests internes.

Listing 7.1: config.hcl - Configuration production

```
storage "file" {
  path = "/home/devops4/vault_data"
}

listener "tcp" {
  address = "127.0.0.1:8200"
  tls_disable = 1
}

ui = true
log_level = "info"
```

> Remarque : le chemin de stockage a été modifié pour un répertoire dans le home utilisateur afin d'éviter les problèmes de permissions sur '/var/lib/vault'.

7.3 Initialisation et déverrouillage (Unsealing)

Listing 7.2: Initialiser et déverrouiller Vault

```
# D finir l'adresse de Vault
export VAULT_ADDR=http://127.0.0.1:8200

# D marrer le serveur Vault
vault server -config=/home/devops4/config.hcl &

# Initialiser Vault (g n re les 5 unseal keys et le root token)
vault operator init -key-shares=5 -key-threshold=3

# Sauvegarder PR CIEUSEMENT les unseal keys et le root token!
# Exemples:
# Unseal Key 1: ...
# Unseal Key 2: ...
# Initial Root Token: ...

# D verrouiller Vault avec 3 cl s
vault operator unseal <key1>
vault operator unseal <key2>
vault operator unseal <key3>

# V rifier que Vault est d verrouill
vault status

# Se connecter avec le root token
vault login <root_token>
```

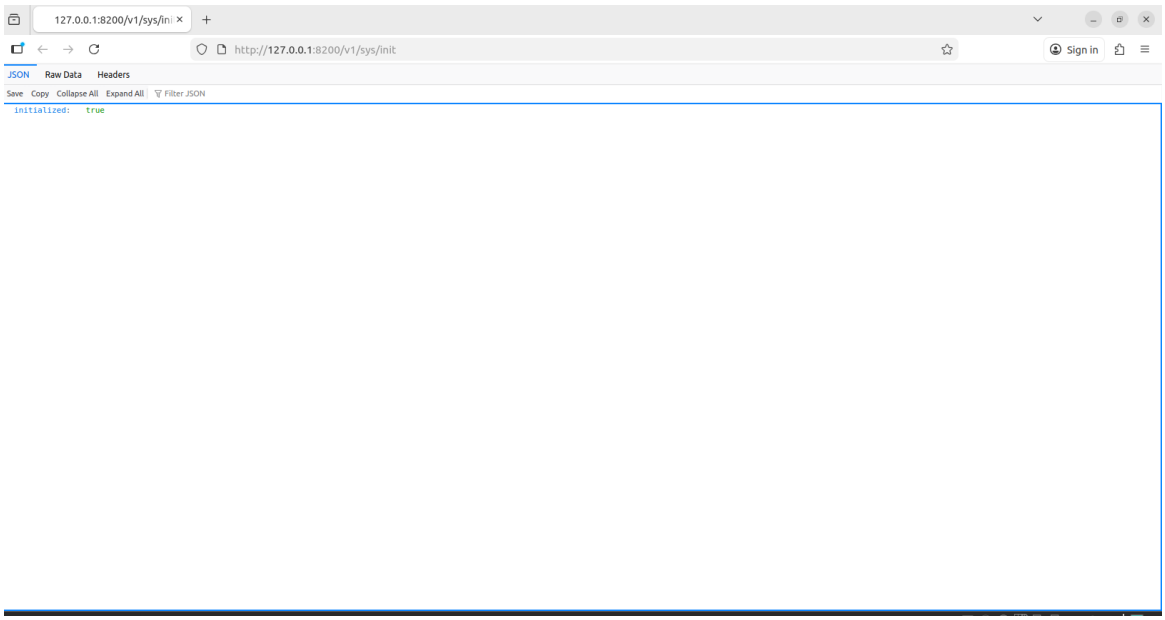


Figure 7.1: Exemple de sortie de `vault operator init` et du processus de déverrouillage

> Après cette étape, Vault est prêt pour créer des secrets, définir des policies et gérer les utilisateurs en production.

CRITIQUE

Les unseal keys et le root token DOIVENT être:

- Stockés de manière sécurisée (pas dans le code!)
- Distribués à différentes personnes
- Sauvegardés dans plusieurs endroits
- Protégés physiquement et logiquement

La perte de ces clés = perte définitive de l'accès aux données!

7.4 Audit Logging

Listing 7.3: Configuration de l'audit

```
# Activer l'audit vers un fichier
vault audit enable file file_path=/var/log/vault/audit.log

# Activer l'audit vers syslog
vault audit enable syslog

# Lister les devices d'audit
vault audit list

# D sactiver un device d'audit
vault audit disable file/
```

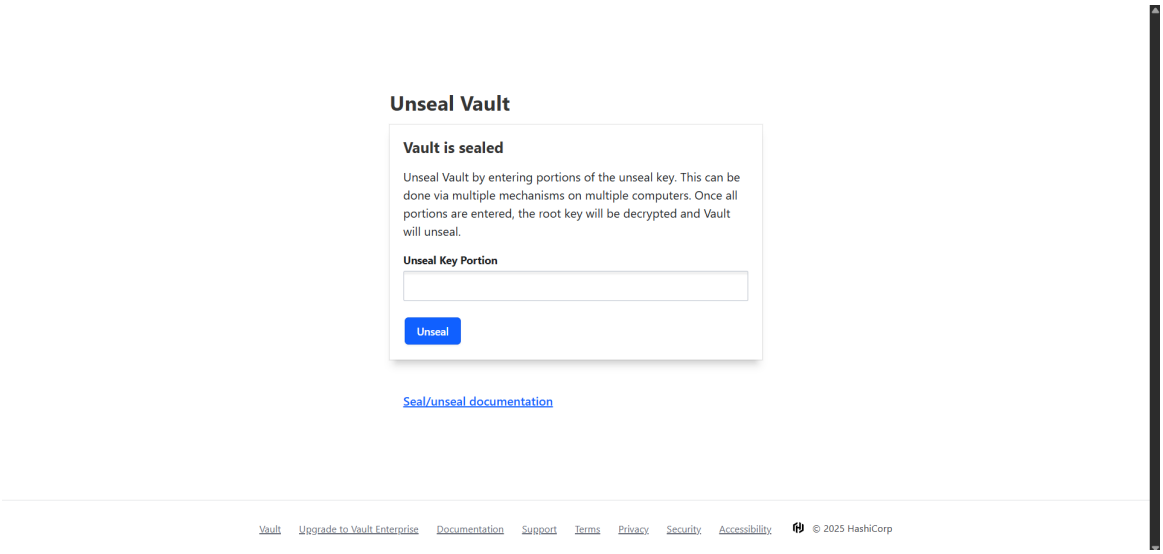


Figure 7.2: Accés vault avec clés d'unseal

Chapter 8

Projet Pratique: Application Web Sécurisée

8.1 Architecture du projet

Cette application Python démontre l'intégration sécurisée de HashiCorp Vault avec une application Flask, afin de :

- Éviter les mots de passe en dur dans le code
- Utiliser AppRole pour l'authentification machine-to-machine
- Utiliser Secrets dynamiques PostgreSQL
- Utiliser Transit Engine pour le chiffrement/déchiffrement
- Centraliser et auditer l'accès aux secrets

Pourquoi c'est important ? Dans un environnement DevOps / Cloud, les applications ne doivent jamais stocker des secrets dans le code ou les variables Git.

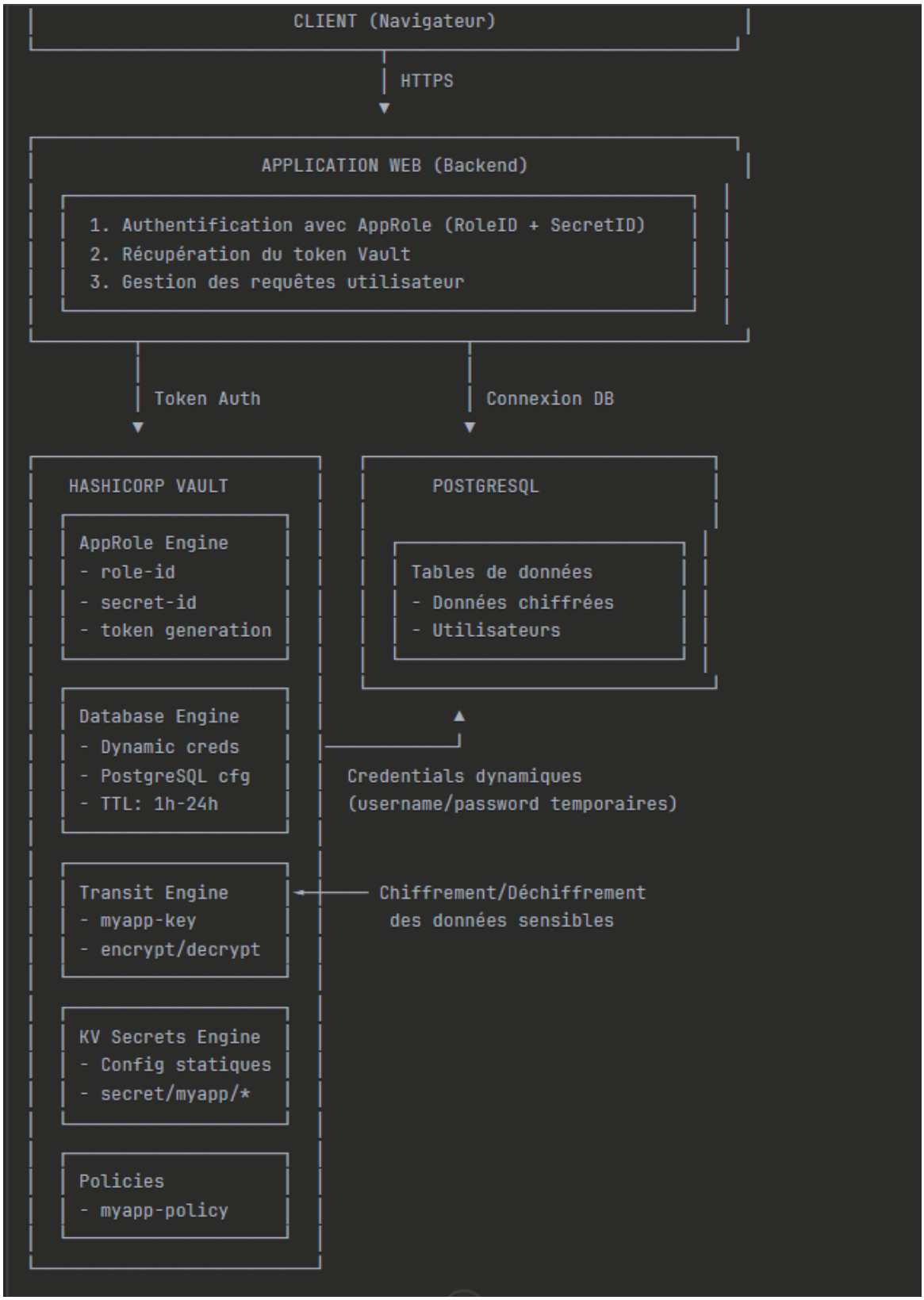


Figure 8.1: Architecture du projet pratique

8.2 Préparation de l'environnement

Listing 8.1: Setup Docker Compose

```yaml
# docker-compose.yml
version: '3.8'

services:
  vault:
    image: hashicorp/vault:latest
    ports:
      - "8200:8200"
    environment:
      VAULT_DEV_ROOT_TOKEN_ID: myroot
      VAULT_DEV_LISTEN_ADDRESS: 0.0.0.0:8200
    cap_add:
      - IPC_LOCK

  postgres:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: myapp
    ports:
      - "5432:5432"
```

Listing 8.2: Démarrer l'environnement

```bash
# Lancer les services
docker-compose up -d

# V rifier que tout fonctionne
docker-compose ps

# Configurer les variables d'environnement
export VAULT_ADDR='http://127.0.0.1:8200'
export VAULT_TOKEN='myroot'
```

Figure 8.2: Démarrage des services avec Docker Compose

8.3 Configuration de Vault

Listing 8.3: Configuration initiale de Vault

```
# 1. Activer le database engine
vault secrets enable database

# 2. Configurer PostgreSQL
vault write database
/config/postgresql \
    plugin_name=postgresql-database-plugin \
    allowed_roles="myapp-role" \
    connection_url="postgresql://{{username}}:{{password}}@postgres:5432/
       myapp?sslmode=disable" \
    username="postgres" \
    password="postgres"

# 3. Cr er un r le pour l'application
vault write database/roles/myapp-role \
    db_name=postgresql \
    creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{
       password}}' VALID UNTIL '{{expiration}}'; \
        GRANT SELECT, INSERT, UPDATE ON ALL TABLES IN SCHEMA public TO
           \"{{name}}\";" \
    default_ttl="1h" \
    max_ttl="24h"

# 4. Activer le Transit engine pour le chiffrement
vault secrets enable transit
vault write -f transit/keys/myapp-key

# 5. Activer AppRole pour l'authentification
vault auth enable approle

# 6. Cr er une policy pour l'application
cat > myapp-policy.hcl <<EOF
# Acc s aux credentials de base de donn es
path "database/creds/myapp-role" {
  capabilities = ["read"]
}

# Acc s au chiffrement/d chiffrement
path "transit/encrypt/myapp-key" {
  capabilities = ["update"]
}

path "transit/decrypt/myapp-key" {
  capabilities = ["update"]
}

# Acc s aux secrets statiques de l'app
path "secret/data/myapp/*" {
  capabilities = ["read", "list"]
}
EOF

vault policy write myapp-policy myapp-policy.hcl

# 7. Cr er le r le AppRole
vault write auth/approle/role/myapp \
    token_policies="myapp-policy" \
```

```
    token_ttl=1h \
    token_max_ttl=4h

# 8. R cup rer RoleID et SecretID
vault read auth/approle/role/myapp/role-id
vault write -f auth/approle/role/myapp/secret-id

# 9. Ajouter des secrets statiques
vault kv put secret/myapp/config \
    app_name="My Secure App" \
    env="development" \
    log_level="debug"
```

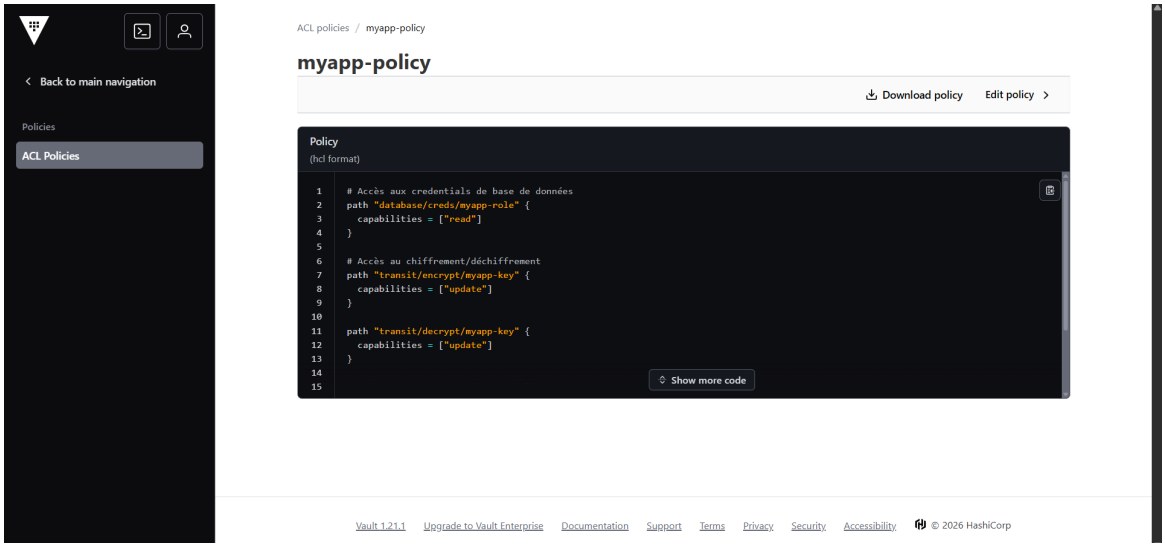


Figure 8.3: Configuration complète de Vault pour le projet

8.4 Application Python avec HashiCorp Vault

Cette section décrit une application Python utilisant Vault pour la gestion des secrets, l'authentification AppRole, le chiffrement/déchiffrement de données et la génération de credentials dynamiques pour une base de données PostgreSQL.

8.4.1 Structure du projet

Listing 8.4: Structure du projet Python

```
myapp/
            app.py              # Application principale Flask
            vault_client.py     # Client Python pour interagir avec Vault
            requirements.txt    # D pendances Python
            .env.example        # Exemple de variables d'environnement
```

8.4.2 Client Vault en Python

Le fichier `vault_client.py` contient une classe `VaultClient` qui gère :

- L'authentification avec AppRole

- La lecture des secrets KV
- Le chiffrement et déchiffrement avec le moteur Transit
- La génération de credentials dynamiques pour PostgreSQL
- Le renouvellement du token Vault si nécessaire

Listing 8.5: vault_client.py : Classe de gestion Vault

```python
import hvac
import base64
import os
from typing import Dict

class VaultClient:
    def __init__(self):
        # R cup ration des variables d'environnement
        self.vault_addr = os.getenv('VAULT_ADDR', 'http://localhost:8200')
        self.role_id = os.getenv('VAULT_ROLE_ID')
        self.secret_id = os.getenv('VAULT_SECRET_ID')
        self.client = None
        self._authenticate()

    def _authenticate(self):
        """Authentification aupr s de Vault via AppRole"""
        self.client = hvac.Client(url=self.vault_addr)
        response = self.client.auth.approle.login(
            role_id=self.role_id,
            secret_id=self.secret_id
        )
        print(f"    Authentifi  avec succ s")
        print(f"  Token TTL: {response['auth']['lease_duration']}s")

    # M thode pour r cup rer des credentials PostgreSQL dynamiques
    def get_db_credentials(self) -> Dict[str, str]:
        response = self.client.secrets.database.generate_credentials(
            name='myapp-role'
        )
        creds = {
            'username': response['data']['username'],
            'password': response['data']['password'],
            'lease_id': response['lease_id'],
            'lease_duration': response['lease_duration']
        }
        print(f"    Credentials DB g n r s : {creds['username']}")
        return creds

    # M thode pour chiffrer des donn es avec Transit
    def encrypt_data(self, plaintext: str) -> str:
        plaintext_b64 = base64.b64encode(plaintext.encode()).decode()
        response = self.client.secrets.transit.encrypt_data(
            name='myapp-key',
            plaintext=plaintext_b64
        )
        return response['data']['ciphertext']

    # M thode pour d chiffrer des donn es avec Transit
```

```
    def decrypt_data(self, ciphertext: str) -> str:
        response = self.client.secrets.transit.decrypt_data(
            name='myapp-key',
            ciphertext=ciphertext
        )
        return base64.b64decode(response['data']['plaintext']).decode()

    # M thode pour lire des secrets KV
    def get_config(self, path: str) -> Dict:
        response = self.client.secrets.kv.v2.read_secret_version(path=path
            )
        return response['data']['data']

    # Renouvellement du token Vault si besoin
    def renew_token(self):
        self.client.auth.token.renew_self()
        print("    Token renouvel ")
```

Figure 8.4: Authentification réussie du client Vault via AppRole

8.4.3 Application Flask principale

Le fichier `app.py` met en place :

- Un serveur Flask exposant plusieurs endpoints
- La gestion automatique des credentials PostgreSQL dynamiques
- Le chiffrement/déchiffrement de données via Vault Transit
- La lecture de secrets statiques depuis KV

Listing 8.6: app.py : Application principale

```
from flask import Flask, jsonify, request
import psycopg2
from vault_client import VaultClient
import time

app = Flask(__name__)
vault = VaultClient()

db_creds = None
db_creds_expiry = 0

# Fonction pour obtenir une connexion DB avec credentials dynamiques
def get_db_connection():
```

```
    global db_creds, db_creds_expiry
    if time.time() > db_creds_expiry:
        db_creds = vault.get_db_credentials()
        db_creds_expiry = time.time() + db_creds['lease_duration'] - 300
    conn = psycopg2.connect(
        host='localhost',
        database='myapp',
        user=db_creds['username'],
        password=db_creds['password']
    )
    return conn

# Endpoint de sant
@app.route('/health')
def health():
    config = vault.get_config('myapp/config')
    return jsonify({
        'status': 'healthy',
        'app_name': config['app_name'],
        'environment': config['env']
    })

# Endpoint pour chiffrer des donn es
@app.route('/encrypt', methods=['POST'])
def encrypt():
    data = request.json.get('data')
    if not data:
        return jsonify({'error': 'Missing data'}), 400
    ciphertext = vault.encrypt_data(data)
    return jsonify({'ciphertext': ciphertext})

# Endpoint pour d chiffrer des donn es
@app.route('/decrypt', methods=['POST'])
def decrypt():
    ciphertext = request.json.get('ciphertext')
    if not ciphertext:
        return jsonify({'error': 'Missing ciphertext'}), 400
    plaintext = vault.decrypt_data(ciphertext)
    return jsonify({'plaintext': plaintext})

# Endpoint pour r cup rer les 10 premiers utilisateurs depuis la DB
@app.route('/users', methods=['GET'])
def get_users():
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT id, email FROM users LIMIT 10")
    users = cursor.fetchall()
    cursor.close()
    conn.close()
    return jsonify({
        'users': [{'id': u[0], 'email': u[1]} for u in users],
        'db_user': db_creds['username']
    })

if __name__ == '__main__':
    db_creds = vault.get_db_credentials()
    db_creds_expiry = time.time() + db_creds['lease_duration'] - 300
    app.run(debug=True, host='0.0.0.0', port=5000)
```

8.4.4 Installation et démarrage

Listing 8.7: Installation des dépendances et lancement de l'application

```
# Installer les d pendances Python
pip install -r requirements.txt

# Configurer les variables d'environnement pour AppRole Vault
export VAULT_ADDR='http://localhost:8200'
export VAULT_ROLE_ID='<votre-role-id>'
export VAULT_SECRET_ID='<votre-secret-id>'

# D marrer l'application Flask
python app.py
```

8.4.5 Tests des endpoints

Listing 8.8: Tester les endpoints de l'application

```
# V rifier la sant  de l'application
curl http://localhost:5000/health

# Chiffrement de donn es
curl -X POST http://localhost:5000/encrypt \
  -H "Content-Type: application/json" \
  -d '{"data": "informations sensibles"}'

# D chiffrement de donn es
curl -X POST http://localhost:5000/decrypt \
  -H "Content-Type: application/json" \
  -d '{"ciphertext": "vault:v1:..."}'

# Lecture des utilisateurs depuis PostgreSQL avec credentials dynamiques
curl http://localhost:5000/users
```

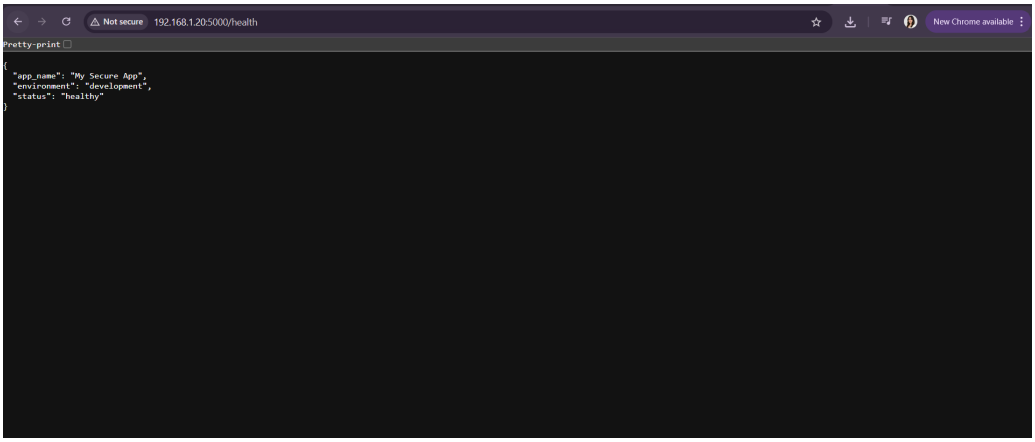


Figure 8.5: Réponse JSON de l'endpoint `/health`

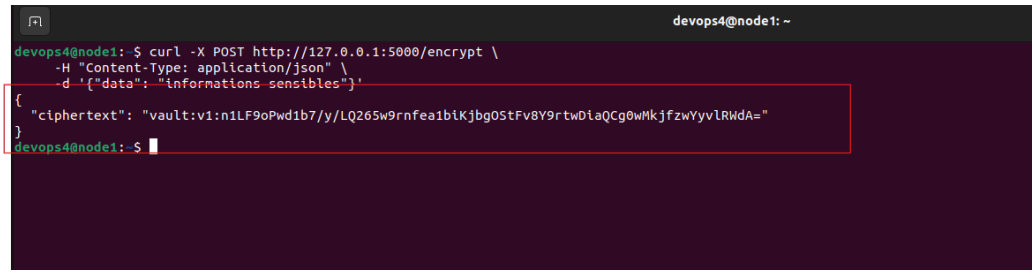


Figure 8.6: Chiffrement de données via l'API avec Vault Transit

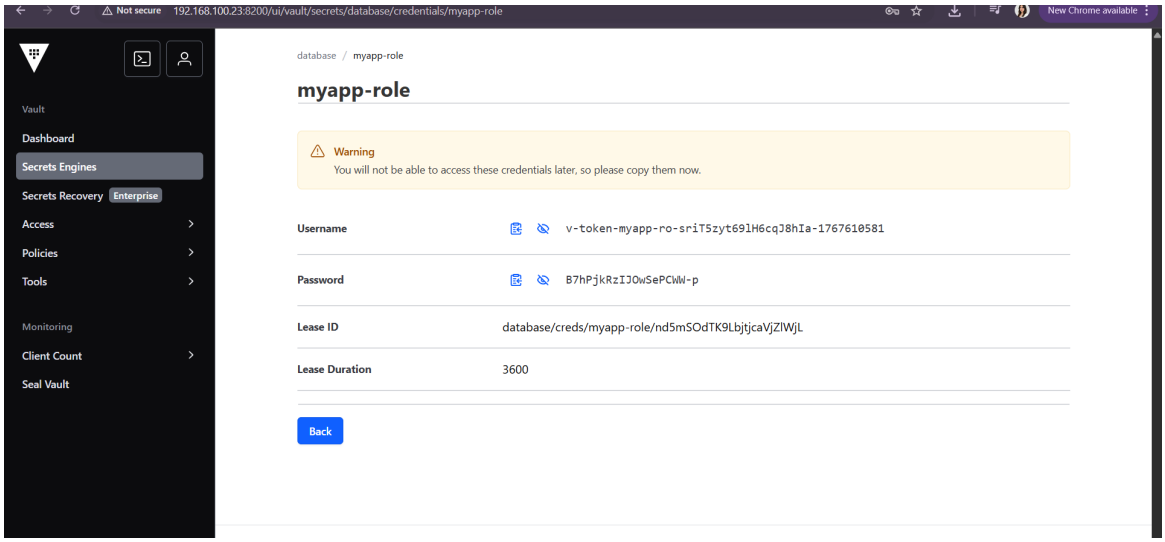


Figure 8.7: Configuration du rôle AppRole 'myapp' dans Vault UI

8.4.6 Explications pour débutants

- **VaultClient** : centralise toutes les interactions avec Vault.
- **AppRole** : méthode d'authentification sécurisée entre l'application et Vault.
- **KV v2** : stockage de secrets statiques, par exemple configuration.
- **Transit** : moteur de chiffrement/déchiffrement, utilisé pour protéger les données sensibles.
- **Database dynamic credentials** : Vault crée des utilisateurs temporaires pour PostgreSQL, ce qui améliore la sécurité.
- **Endpoints Flask** : fournissent des interfaces HTTP pour tester la lecture, le chiffrement et l'accès aux secrets.

8.4.7 Lien GitHub

Le code source complet de cette application est disponible sur GitHub. Les étudiants peuvent cloner le projet et l'exécuter localement pour suivre les exercices et tests.

- Dépôt GitHub : https://github.com/maryeeem/myapp-vault.git

- Pour cloner le projet :

```
git clone https://github.com/maryeeem/myapp-vault.git
cd myapp-vault
```

Chapter 9

Haute Disponibilité et Clustering

9.1 Introduction

HashiCorp Vault peut être déployé en mode **Haute Disponibilité (HA)** pour assurer la continuité de service et la tolérance aux pannes. Dans ce mode, plusieurs nœuds Vault collaborent pour traiter les requêtes tout en garantissant que les secrets restent accessibles même si un nœud tombe en panne.

9.2 Architecture HA

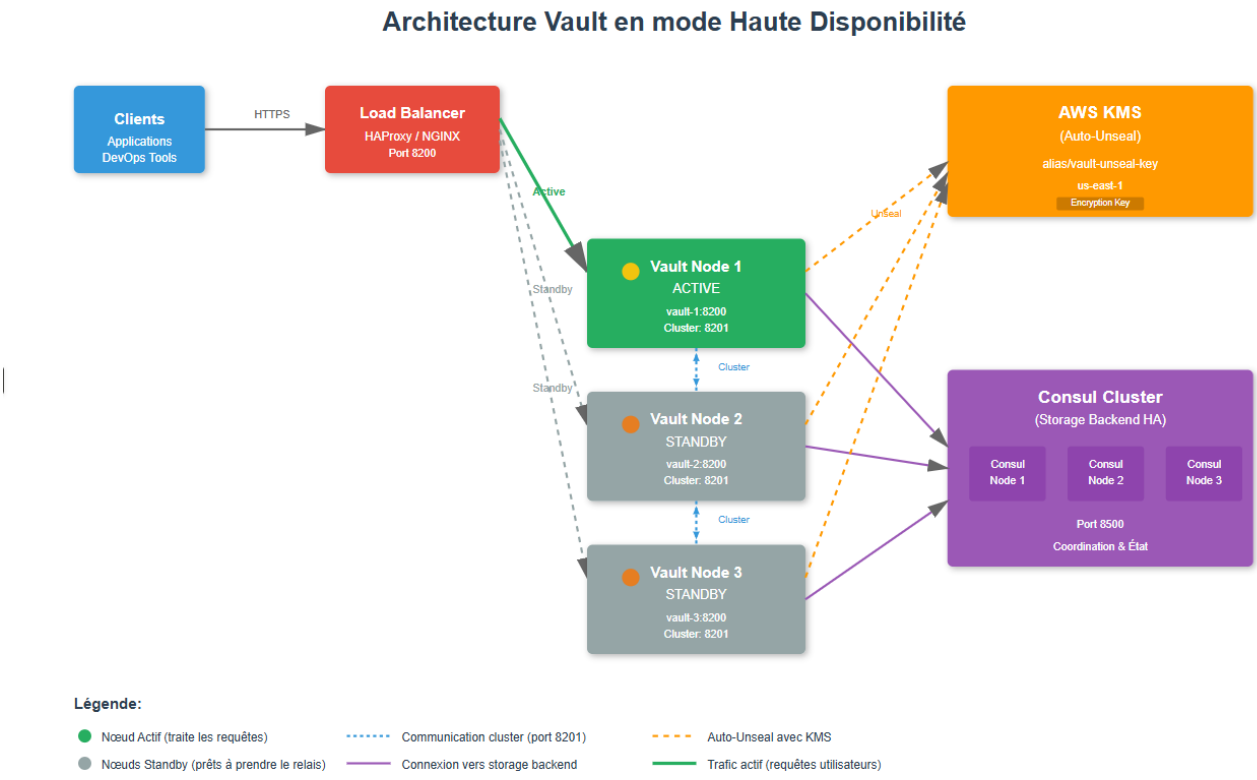


Figure 9.1: Architecture Vault en mode Haute Disponibilité

9.2.1 Explication des composants

- **Active node** : Un seul nœud actif qui gère toutes les requêtes d'écriture et de lecture.
- **Standby nodes** : Nœuds en attente, qui peuvent prendre le relais si le nœud actif tombe en panne.
- **Storage backend HA** : Backend de stockage compatible HA, comme Consul ou etcd, pour synchroniser l'état du cluster.
- **Load balancer** : Redirige les requêtes vers le nœud actif et répartit la charge.

Figure à fournir : Schéma montrant 1 nœud actif, plusieurs standby, un backend de stockage HA et un load balancer.

—

9.3 Configuration d'un cluster Vault avec Consul

9.3.1 Étape 1 : Configurer le stockage HA

Le fichier `vault-node1.hcl` définit le stockage backend et les adresses API et Cluster :

Listing 9.1: Exemple de configuration d'un nœud Vault avec Consul

```
storage "consul" {
  address = "consul:8500"
  path    = "vault/"
}

listener "tcp" {
  address       = "0.0.0.0:8200"
  tls_cert_file = "/vault/config/vault.crt"
  tls_key_file  = "/vault/config/vault.key"
}

api_addr = "https://vault-node1:8200"
cluster_addr = "https://vault-node1:8201"

ui = true

seal "awskms" {
  region     = "us-east-1"
  kms_key_id = "alias/vault-key"
}
```

Explication :

- `storage "consul"` : définit Consul comme backend HA pour synchroniser l'état du cluster.
- `listener "tcp"` : expose Vault via TLS.
- `seal "awskms"` : configure Auto-Unseal (ici AWS KMS).

9.3.2 Étape 2 : Déploiement avec Docker Compose

Le fichier `docker-compose.yml` permet de lancer un cluster de 3 nœuds Vault avec Consul :

Listing 9.2: Docker Compose pour cluster Vault HA

```
version: '3.8'

services:
  consul:
    image: consul:latest
    command: agent -server -bootstrap-expect=1 -ui -client=0.0.0.0
    ports:
      - "8500:8500"

  vault-1:
    image: hashicorp/vault:latest
    ports:
      - "8200:8200"
    volumes:
      - ./config/vault-node1.hcl:/vault/config/vault.hcl
    environment:
      VAULT_API_ADDR: https://vault-1:8200
    depends_on:
      - consul
    cap_add:
      - IPC_LOCK

  vault-2:
    image: hashicorp/vault:latest
    ports:
      - "8201:8200"
    volumes:
      - ./config/vault-node2.hcl:/vault/config/vault.hcl
    environment:
      VAULT_API_ADDR: https://vault-2:8200
    depends_on:
      - consul
    cap_add:
      - IPC_LOCK

  vault-3:
    image: hashicorp/vault:latest
    ports:
      - "8202:8200"
    volumes:
      - ./config/vault-node3.hcl:/vault/config/vault.hcl
    environment:
      VAULT_API_ADDR: https://vault-3:8200
    depends_on:
      - consul
    cap_add:
      - IPC_LOCK
```

Explication :

- Chaque service Vault est un nœud du cluster.
- `depends_on` assure que Consul démarre avant Vault.

- `IPC_LOCK` empêche Vault de swapper la mémoire sensible.

9.3.3 Étape 3 : Vérifier le cluster

- Vérifier le statut de chaque nœud avec la commande :

```
vault status
```

- Faire une capture d'écran pour montrer les nœuds Active et Standby.

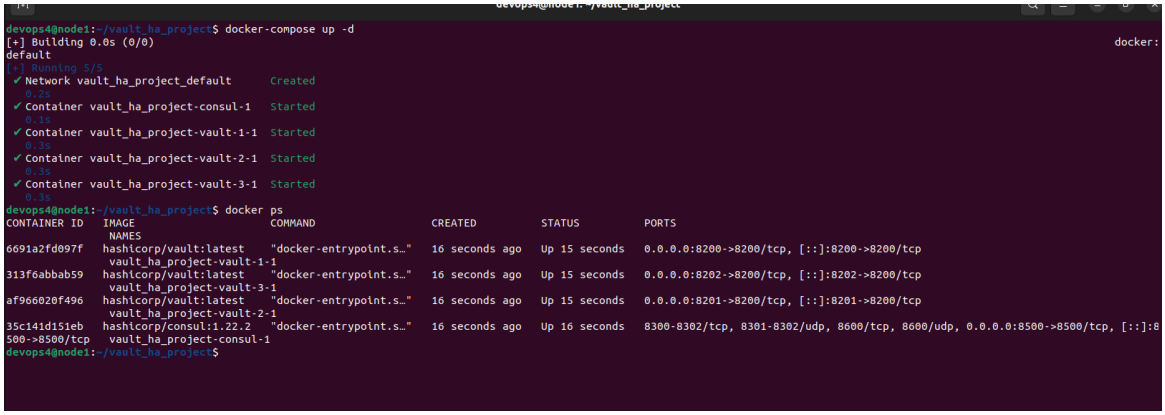


Figure 9.2: Statut du cluster Vault avec 3 nœuds

9.4 Configuration Auto-Unseal avec AWS KMS

Vault peut se déverrouiller automatiquement grâce à AWS KMS, ce qui évite de saisir manuellement les clés de déverrouillage.

9.4.1 Étape 1 : Créer la clé KMS

```
aws kms create-key --description "Vault Unseal Key"
aws kms create-alias --alias-name alias/vault-unseal-key --target-key-id <key-id>
```

9.4.2 Étape 2 : Configurer Vault

Ajouter dans le fichier `vault.hcl` :

```
seal "awskms" {
  region     = "us-east-1"
  kms_key_id = "alias/vault-unseal-key"
  endpoint   = "https://kms.us-east-1.amazonaws.com"
}
```

9.4.3 Étape 3 : Initialiser Vault

```
vault operator init -recovery-shares=5 -recovery-threshold=3
```

- Capturer la sortie de l'initialisation montrant les clés de récupération et que Vault est Auto-Unseal.

Explication :

- Vault peut se déverrouiller seul grâce à KMS.
- Les `recovery-shares` et `threshold` permettent de récupérer Vault si la KMS n'est pas accessible.

Chapter 10

Intégration de Vault avec Kubernetes

Ce chapitre présente l'intégration de HashiCorp Vault avec Kubernetes afin de sécuriser la gestion des secrets dans des applications conteneurisées. L'objectif est de permettre aux pods Kubernetes d'accéder aux secrets de manière dynamique, sécurisée et sans stocker d'informations sensibles dans le code ou les fichiers de configuration.

10.1 Déploiement de Vault sur un cluster Kubernetes mono-nœud

Dans un environnement de test ou de lab avec un seul nœud Kubernetes, il est recommandé de déployer **un seul pod Vault** pour éviter les problèmes de volumes hostPath et de planification.

10.1.1 Installation avec Helm

Les étapes suivantes permettent d'installer Vault avec un pod unique et l'interface graphique activée.

Listing 10.1: Installation de Vault sur Kubernetes mono-nœud avec Helm

```
# Ajouter le d p t Helm officiel de HashiCorp
helm repo add hashicorp https://helm.releases.hashicorp.com
helm repo update

# Cr er le namespace Vault
kubectl create namespace vault

# Installer Vault en mono-pod (HA d sactiv e)
helm install vault hashicorp/vault \
  --namespace vault \
  --set server.ha.enabled=false \
  --set ui.enabled=true \
  --set ui.serviceType=LoadBalancer \
  --set server.dataStorage.enabled=true \
  --set server.dataStorage.storageClass=standard \
  --set server.dataStorage.size=1Gi

# V rifier l tat du pod
kubectl get pods -n vault
```

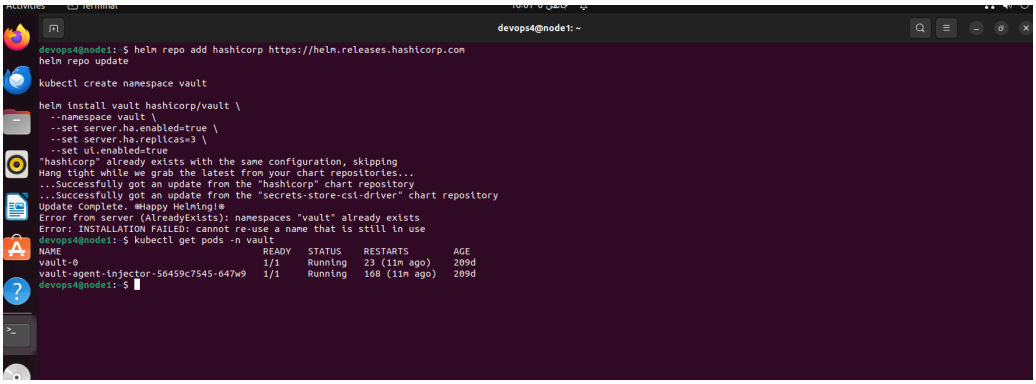


Figure 10.1: Pod Vault déployé sur un cluster Kubernetes mono-nœud

Remarque : Pour un cluster mono-nœud, l'option 'hostPath' ou une PV locale fonctionne correctement car le pod est toujours planifié sur le même nœud.

10.2 Création des Persistent Volumes (PV) et Persistent Volume Claims (PVC)

Pour permettre à Vault de stocker les secrets de manière persistante :

10.2.1 Création des volumes sur le nœud

Listing 10.2: Préparer les dossiers sur le nœud

```
sudo mkdir -p /mnt/data/vault
sudo chown -R 1000:1000 /mnt/data/vault
sudo chmod -R 700 /mnt/data/vault
```

10.2.2 Définition du PV et PVC

Listing 10.3: Persistent Volume et Persistent Volume Claim pour Vault

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: vault-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  persistentVolumeReclaimPolicy: Delete
  hostPath:
    path: "/mnt/data/vault"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: vault-data
```

```
    namespace: vault
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
  volumeName: vault-pv
```

Appliquer ces fichiers :

Listing 10.4: Appliquer PV et PVC

```
kubectl apply -f vault-pv.yaml
kubectl apply -f vault-pvc.yaml
```

Vérifier que le PVC est bien **Bound** :

```
kubectl get pvc -n vault
```

10.3 Initialisation et Unseal du Pod Vault

Une fois le pod Vault lancé, il doit être initialisé et déverrouillé pour fonctionner.

10.3.1 Initialisation

Listing 10.5: Initialisation de Vault

```
kubectl exec -n vault vault-0 -- vault operator init \
  -key-shares=5 \
  -key-threshold=3 \
  -format=json > cluster-keys.json
```

10.3.2 Déverrouillage (Unseal)

Listing 10.6: Déverrouillage de Vault

```
kubectl exec -n vault vault-0 -- vault operator unseal <UNSEAL_KEY_1>
kubectl exec -n vault vault-0 -- vault operator unseal <UNSEAL_KEY_2>
kubectl exec -n vault vault-0 -- vault operator unseal <UNSEAL_KEY_3>
```

Vérifier le statut :

```
kubectl exec -n vault vault-0 -- vault status
```

10.4 Authentification Kubernetes

Vault peut vérifier l'identité des pods via leur ServiceAccount, ce qui permet un accès sécurisé aux secrets.

10.4.1 Activation et configuration

Listing 10.7: Activation de l'auth Kubernetes

```
kubectl exec -n vault vault-0 -- vault auth enable kubernetes

kubectl exec -n vault vault-0 -- vault write auth/kubernetes/config \
  kubernetes_host="https://kubernetes.default.svc:443" \
  kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      \
  token_reviewer_jwt=@/var/run/secrets/kubernetes.io/serviceaccount/token
```

10.4.2 Création d'une policy et d'un rôle

Listing 10.8: Policy Vault pour l'application

```
path "secret/data/k8s-app/*" {
  capabilities = ["read"]
}
```

Listing 10.9: Association du rôle à un ServiceAccount

```
kubectl exec -n vault vault-0 -- vault policy write k8s-app k8s-app-policy
    .hcl

kubectl exec -n vault vault-0 -- vault write auth/kubernetes/role/myapp \
  bound_service_account_names=myapp \
  bound_service_account_namespaces=default \
  policies=k8s-app \
  ttl=24h
```

10.5 Injection des secrets avec Vault Agent Injector

10.5.1 Exemple de pod utilisant Vault Agent Injector

Listing 10.10: Pod Kubernetes utilisant Vault Agent Injector

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  annotations:
    vault.hashicorp.com/agent-inject: "true"
    vault.hashicorp.com/role: "myapp"
    vault.hashicorp.com/agent-inject-secret-database: "secret/data/k8s-app
       /database"
    vault.hashicorp.com/agent-inject-template-database: |
      {{- with secret "secret/data/k8s-app/database" -}}
      export DB_USERNAME="{{ .Data.data.username }}"
      export DB_PASSWORD="{{ .Data.data.password }}"
      {{- end }}
spec:
  serviceAccountName: myapp
  containers:
  - name: app
    image: myapp:latest
```

```
19      command: ["/bin/sh"]
20      args: ["-c", "source /vault/secrets/database && ./app"]
```

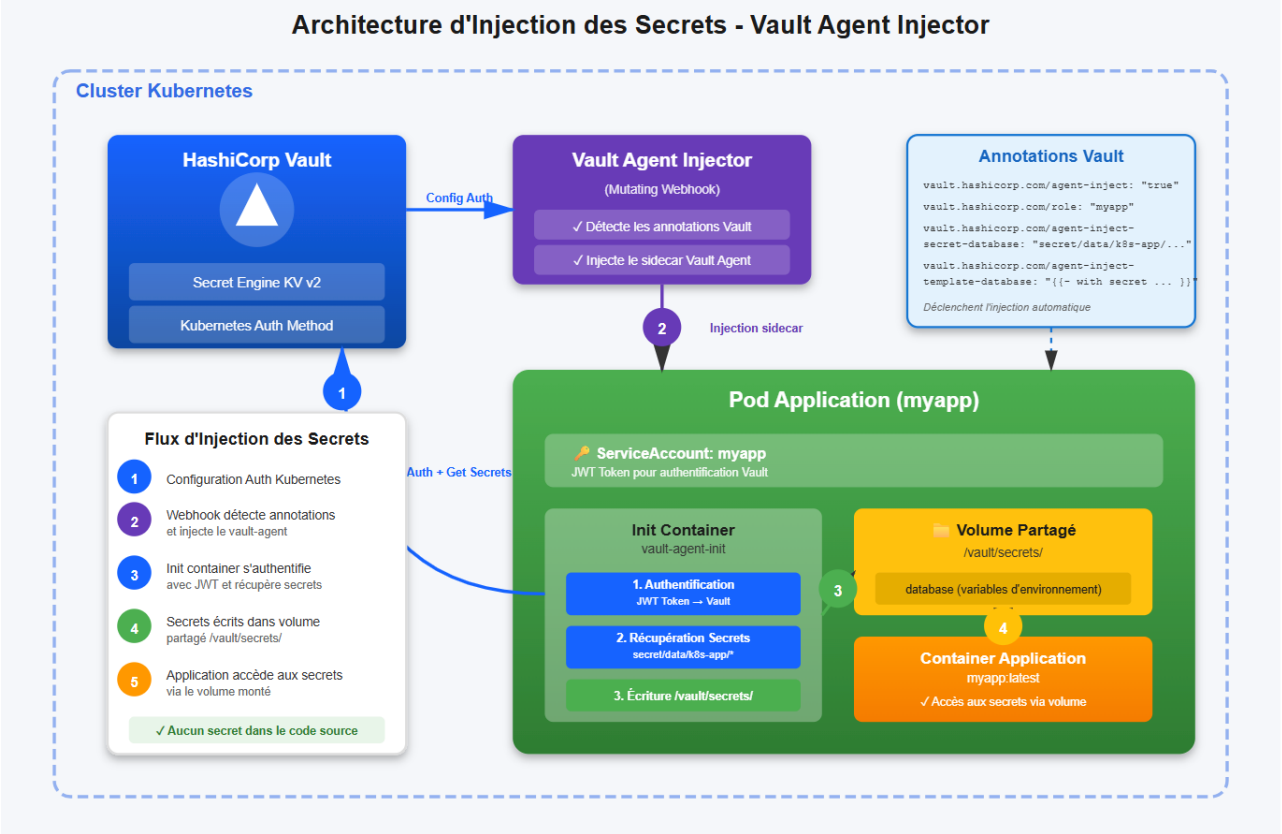


Figure 10.2: Injection automatique des secrets via Vault Agent Injector

Chapter 11

Monitoring et Troubleshooting de Vault

Vault est un composant critique pour la gestion des secrets et des tokens. Un bon monitoring permet de détecter rapidement les problèmes et de garantir la disponibilité et la sécurité. Dans ce chapitre, nous détaillons comment collecter les métriques avec Prometheus, les visualiser avec Grafana, et résoudre les problèmes courants.

11.1 Collecte des métriques Vault

Vault expose des métriques détaillées sur son fonctionnement interne. Ces métriques sont essentielles pour surveiller :

- L'état de déverrouillage (sealed/unsealed)
- Les nœuds actifs et standby dans un cluster HA
- L'utilisation mémoire et CPU
- L'activité des tokens et des leases
- Les performances générales

11.1.1 Configuration de la telemetry pour Prometheus

Pour que Vault expose ses métriques au format Prometheus, il faut configurer le bloc 'telemetry' dans le fichier de configuration de Vault (`vault.hcl`) :

Listing 11.1: Configuration telemetry dans vault.hcl

```
1 telemetry {
2   prometheus_retention_time = "30s"  # Intervalle de rafra chissement des
        m triques
3   disable_hostname = true            # Supprime  l ajout  automatique du
       hostname
4 }
```

> Cette configuration permet à Prometheus de scraper régulièrement Vault et d'obtenir des métriques à jour.

11.1.2 Scraper les métriques depuis Vault

Après avoir démarré Vault et défini le token root, on peut récupérer les métriques Prometheus via l'API :

Listing 11.2: Récupération des métriques Prometheus

```
export VAULT_ADDR=http://192.168.100.23:8200
export VAULT_TOKEN=hvs.Owl4mAk7WK2OnkeCQGFhiqOJ

curl -H "X-Vault-Token: $VAULT_TOKEN" \
     http://192.168.100.23:8200/v1/sys/metrics?format=prometheus
```

Métriques importantes à surveiller :

- `vault core unsealed` : indique si Vault est sealed ou déverrouillé
- `vault core active` : indique si le nœud est actif ou standby
- `vault runtime alloc bytes` : mémoire utilisée par Vault
- `vault token creation` : nombre de tokens créés
- `vault expire num leases` : nombre de leases actifs

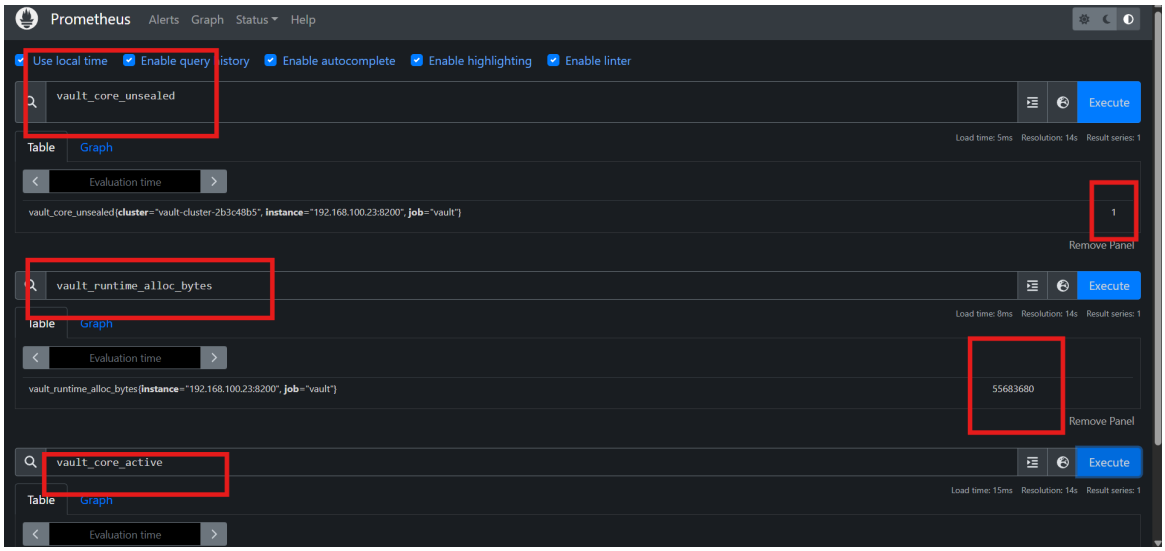


Figure 11.1: Métriques Vault accessibles via Prometheus

11.1.3 Visualisation avec Grafana

Une fois les métriques disponibles via Prometheus, Grafana permet de créer des dashboards graphiques :

- Installer Grafana et connecter Prometheus comme datasource
- Créer un dashboard avec des panels pour chaque métrique clé
- Visualiser par exemple : utilisation mémoire, nombre de tokens, état des nœuds, performance des leases

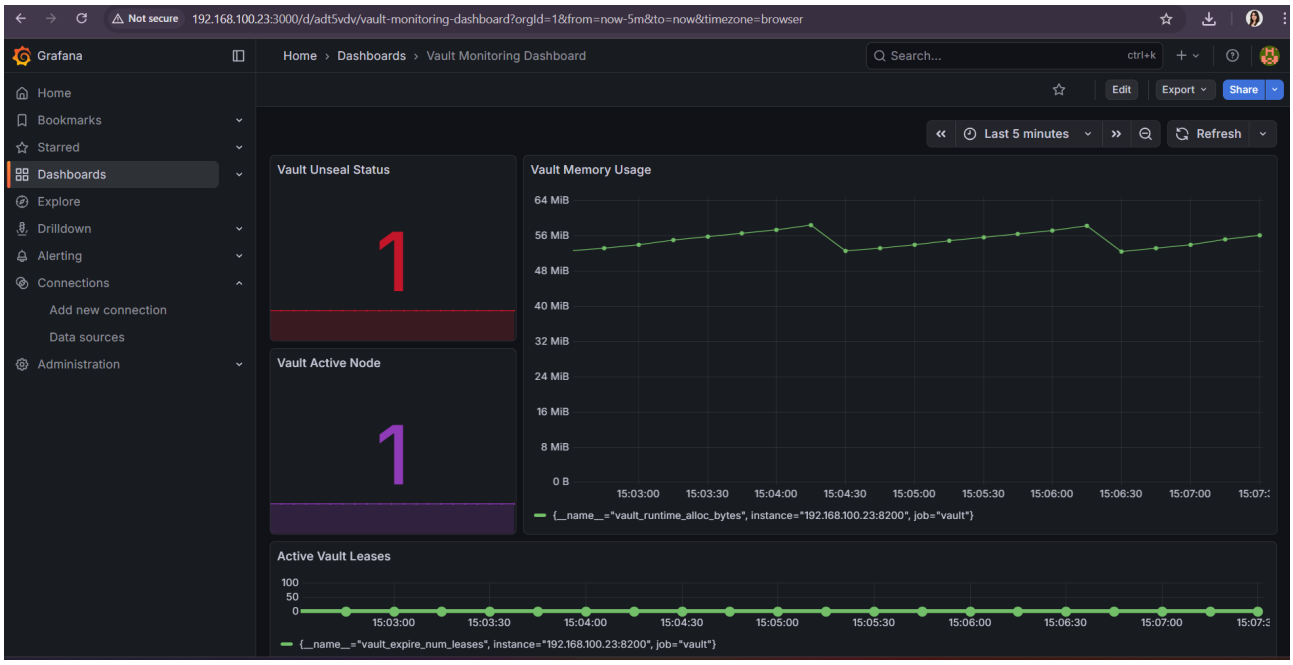


Figure 11.2: Exemple de dashboard Grafana pour Vault

11.2 Troubleshooting courant

11.2.1 Vault sealed après redémarrage

Si Vault apparaît comme ‘sealed‘ après un redémarrage, il faut vérifier l’état et déverrouiller manuellement si nécessaire :

Listing 11.3: Diagnostiquer un problème de seal

```
# V rifier l' tat  de Vault
vault status

# Consulter les logs r cents
journalctl -u vault -n 100

# D verrouiller Vault
vault operator unseal

# V rifier l'auto-unseal si configur
vault read sys/seal-status
```

11.2.2 Performance dégradée

Pour détecter et diagnostiquer les problèmes de performance :

Listing 11.4: Analyse des performances Vault

```
# Consulter les leases actifs
vault list sys/leases/lookup/

# V rifier les tokens actifs
vault token lookup -accessor <accessor-id>
```

```

# Analyser les logs d'audit
tail -f /var/log/vault/audit.log | jq

# Profiling pour analyser le CPU/m moire
vault debug -duration=30s -output=vault-debug.tar.gz
```

11.2.3 Problèmes de connexion

Pour diagnostiquer les problèmes réseau ou TLS :

Listing 11.5: Diagnostics de connexion

```
# V rifier la sant  de Vault
curl -k https://vault.example.com:8200/v1/sys/health

# Tester TLS
openssl s_client -connect vault.example.com:8200

# V rifier les certificats PKI
vault read sys/mounts/pki/cert/ca

# Activer les logs d taill s
export VAULT_LOG_LEVEL=trace
vault status
```

Chapter 12

Sécurité Avancée

Ce chapitre présente les mécanismes de sécurité avancée offerts par HashiCorp Vault. Ces fonctionnalités permettent de renforcer la protection des secrets sensibles, de réduire les risques liés à la compromission des accès et d'assurer une gestion sécurisée du cycle de vie des secrets.

12.1 Rotation des secrets

La rotation des secrets est un principe fondamental de la sécurité moderne. Elle consiste à renouveler périodiquement les clés, mots de passe ou identifiants afin de limiter leur durée d'exposition en cas de fuite ou de compromission.

Vault automatise cette rotation pour plusieurs types de secrets, réduisant ainsi les erreurs humaines et améliorant la conformité aux bonnes pratiques de sécurité.

12.1.1 Rotation automatique des clés

Vault permet de configurer une rotation automatique pour les clés cryptographiques et les identifiants dynamiques, notamment via les moteurs *Transit* et *Database*.

Listing 12.1: Configuration de la rotation automatique des secrets

```
# Activer la rotation automatique pour une cl  Transit
vault write transit/keys/myapp-key/config \
  auto_rotate_period=24h

# D finir la dur e de vie maximale des connexions    la base de donn es
vault write database/config/postgresql \
  max_connection_lifetime=1h

# Forcer une rotation manuelle imm diate
vault write -f transit/keys/myapp-key/rotate
```

La rotation automatique garantit que :

- les clés cryptographiques sont régulièrement renouvelées ;
- les identifiants de bases de données expirent automatiquement ;
- les applications utilisent toujours des secrets valides et à jour.

12.1.2 Révocation de secrets

La révocation permet d'invalider immédiatement un secret ou un accès compromis. Vault offre plusieurs niveaux de révocation : par lease, par chemin ou par utilisateur.

Listing 12.2: Révocation des secrets dans Vault

```
# R voquer un lease sp cifique
vault lease revoke database/creds/myapp/abc123

# R voquer tous les leases associ s    un chemin
vault lease revoke -prefix database/creds/myapp/

# R voquer tous les tokens associ s    un utilisateur
vault token revoke -mode=path auth/userpass/login/alice
```

La révocation est particulièrement utile dans les cas suivants :

- compromission d'un identifiant ;
- départ d'un employé ;
- incident de sécurité nécessitant une coupure immédiate des accès.

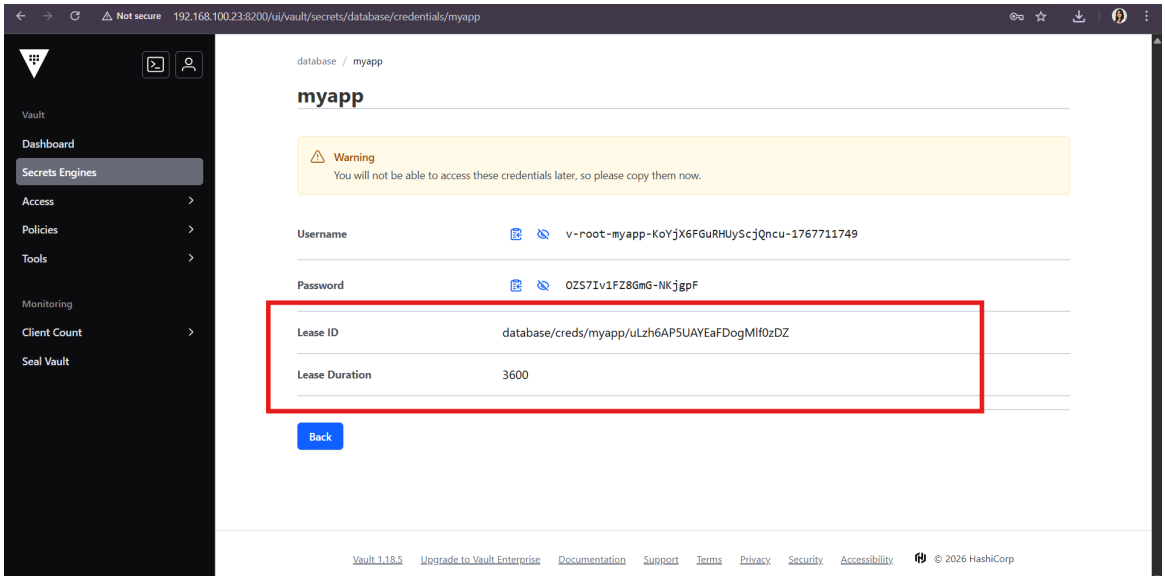


Figure 12.1: Processus de révocation des leases dans Vault

12.2 Response Wrapping

Le *Response Wrapping* est un mécanisme de sécurité avancé permettant de transmettre des secrets de manière temporaire et sécurisée. Vault encapsule le secret dans un token à usage unique, avec une durée de vie limitée.

12.2.1 Principe de fonctionnement

- Le secret réel n'est jamais transmis directement.
- Un token de wrapping est généré avec un TTL défini.

- Le token ne peut être utilisé qu'une seule fois.

Listing 12.3: Utilisation du Response Wrapping

```
# G n rer un secret wrapp  avec une dur e de vie de 60 secondes
vault kv get -wrap-ttl=60s secret/myapp/config

# Le destinataire r cup re le secret (une seule fois)
vault unwrap <wrapping-token>

# G n rer un Secret ID AppRole wrapp
vault write -wrap-ttl=120s -f \
  auth/approle/role/myapp/secret-id
```

Ce mécanisme est particulièrement adapté aux scénarios suivants :

- transmission sécurisée de secrets entre équipes ;
- intégration CI/CD ;
- déploiement automatisé d'applications.

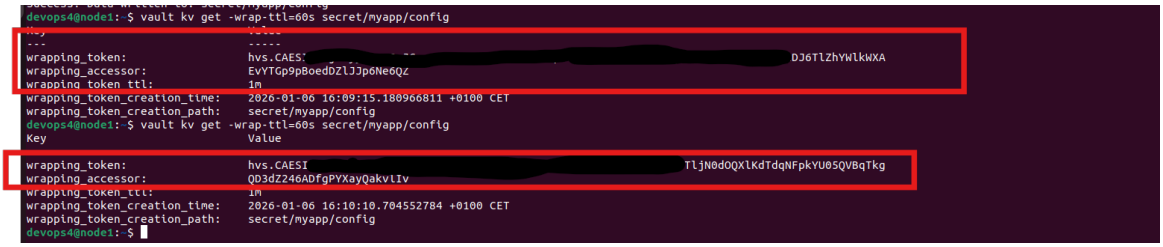


Figure 12.2: Mécanisme de Response Wrapping dans Vault

12.3 Sentinel Policies (Vault Enterprise)

Sentinel est un langage de politiques utilisé dans la version Enterprise de Vault. Il permet d'appliquer des règles de gouvernance et de conformité avant l'exécution des opérations sensibles.

12.3.1 Contrôle d'accès avancé

Les politiques Sentinel permettent par exemple :

- d'imposer l'authentification multifacteur (MFA) ;
- de restreindre les opérations à des plages horaires spécifiques ;
- de bloquer certaines actions en dehors du cadre défini.

Listing 12.4: Exemple de politique Sentinel

```
import "time"

# Autoriser les op rations sensibles uniquement
# pendant les heures ouvrables
```

```
main = rule {
  time.now.weekday in ["Monday", "Tuesday", "Wednesday", "Thursday", "
     Friday"] and
  time.now.hour >= 9 and
  time.now.hour <= 18 and
  request.operation in ["create", "update", "delete"]
}
```

Grâce à Sentinel, Vault devient un outil central de gouvernance de la sécurité, garantissant que les accès respectent les politiques internes et les exigences de conformité.

Chapter 13

Bonnes Pratiques

Dans ce chapitre, nous présentons les recommandations essentielles pour sécuriser et administrer HashiCorp Vault dans un environnement de production. L'objectif est de garantir la confidentialité, l'intégrité et la disponibilité des secrets, tout en facilitant la maintenance et la conformité.

13.1 Gestion des secrets

La gestion des secrets est au cœur de Vault. Une configuration correcte et des pratiques rigoureuses permettent de limiter les risques de fuite ou de compromission des informations sensibles.

À FAIRE

- **Utiliser des credentials dynamiques :** Générer des secrets temporaires pour limiter l'exposition.
- **Définir des TTL courts :** Réduire la durée de vie des secrets selon le principe du moindre privilège.
- **Activer l'audit logging dès le début :** Suivre toutes les actions pour détecter les anomalies et garantir la traçabilité.
- **Utiliser des policies granulaires :** Accorder uniquement les permissions nécessaires pour chaque utilisateur ou application.
- **Implémenter la rotation automatique des secrets :** Éviter la réutilisation prolongée des credentials sensibles.
- **Monitorer les métriques critiques :** Surveiller les performances et les alertes pour anticiper les incidents.
- **Tester régulièrement les backups :** S'assurer que les sauvegardes sont fiables et restaurables.
- **Utiliser TLS partout :** Chiffrer toutes les communications pour prévenir les interceptions.

À ÉVITER

- **Utiliser le mode dev en production :** Ce mode n'est pas sécurisé et ne doit jamais être exposé.
- **Stocker le root token :** Le root token doit être révoqué après initialisation pour éviter les compromissions.
- **Partager les tokens entre applications :** Chaque application doit avoir son propre token pour limiter l'impact d'un incident.
- **Désactiver l'audit logging :** Empêche de tracer les actions et complique la détection des intrusions.
- **Utiliser des TTL trop longs :** Augmente la fenêtre d'exposition en cas de fuite.
- **Négliger la rotation des clés :** Les secrets statiques sont plus vulnérables aux attaques.
- **Ignorer les logs d'audit :** Les logs sont essentiels pour les analyses post-incidents.
- **Communication HTTP non chiffrée :** Expose les secrets aux interceptions réseau.

13.2 Architecture recommandée

Pour un déploiement sécurisé et résilient, il est conseillé de suivre l'architecture de production illustrée ci-dessous. Cette configuration inclut le high-availability, la réplication des données et l'auto-unseal via KMS pour un fonctionnement fiable.

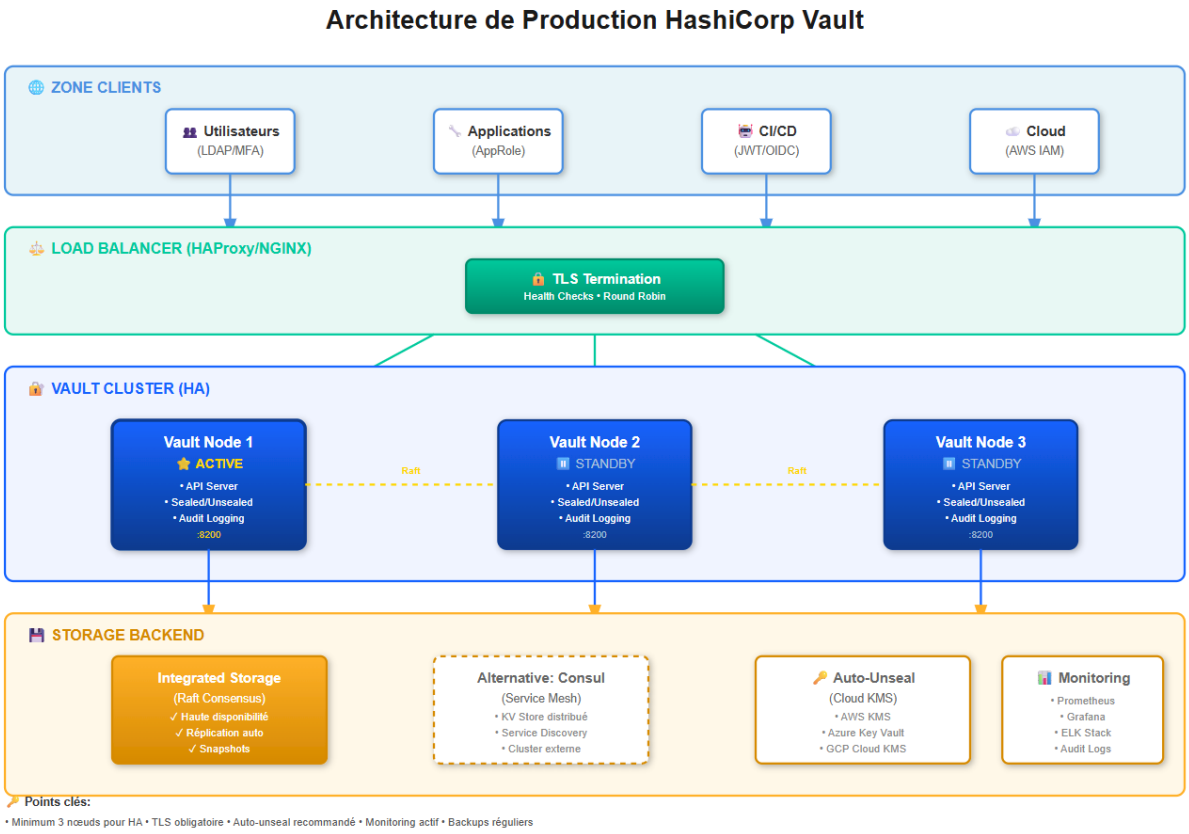


Figure 13.1: Architecture de production recommandée pour Vault

13.3 Checklist de sécurité

La checklist suivante regroupe les actions essentielles pour sécuriser un déploiement Vault. Elle peut servir de référence lors de l'audit ou de la mise en production.

1. **Initialisation**

   Générer un nombre suffisant de clés unseal (5 ou plus) pour garantir la résilience.

   Distribuer les clés à différentes personnes de confiance pour éviter un point de défaillance unique.

   Sauvegarder les clés dans plusieurs coffres-forts sécurisés.

   Révoquer le root token initial après configuration.

2. **Configuration**

   Activer TLS avec des certificats valides pour chiffrer les communications.

   Configurer l'auto-unseal via un KMS pour simplifier la mise en service après un reboot.

   Activer l'audit logging sur plusieurs destinations pour garantir la disponibilité des logs.

   Mettre en place le mode HA avec au moins 3 nœuds pour la haute disponibilité.

3. **Accès**

   Utiliser AppRole pour les applications automatisées.

   Implémenter une authentification multi-facteurs (MFA) pour les utilisateurs humains.

   Définir des policies suivant le principe du moindre privilège.

   Planifier et automatiser la rotation régulière des secrets.

4. **Monitoring**

   Mettre en place des alertes sur les métriques critiques pour détecter les anomalies.

   Examiner régulièrement les logs d'audit pour identifier tout comportement suspect.

   Effectuer des tests de disaster recovery pour s'assurer de la résilience.

   Maintenir une documentation complète et à jour pour toutes les configurations et procédures.

Chapter 14

Cas d'Usage Avancés

HashiCorp Vault ne se limite pas au stockage et à la rotation de secrets simples. Il offre des moteurs avancés qui permettent de gérer des infrastructures cryptographiques (PKI), de signer des certificats SSH ou de générer des OTP pour l'authentification à deux facteurs. Ce chapitre détaille ces cas d'usage avancés.

14.1 PKI (Public Key Infrastructure)

Le moteur PKI de Vault permet de créer et gérer des certificats X.509 de manière centralisée, en remplaçant ou en complétant une autorité de certification traditionnelle.

14.1.1 Étapes de configuration

1. **Activer le moteur PKI :** Il faut activer le moteur PKI sur un chemin spécifique dans Vault.

```
vault secrets enable pki
```

2. **Générer une CA racine interne :** La CA (Certificate Authority) interne émettra les certificats pour vos services.

```
vault write pki/root/generate/internal \
    common_name="My Company Root CA" \
    ttl=87600h
```

3. **Configurer les URLs :** Ces URLs seront utilisées pour publier les certificats et la CRL (Certificate Revocation List).

```
vault write pki/config/urls \
    issuing_certificates="https://vault.example.com:8200/v1/pki/ca" \
    crl_distribution_points="https://vault.example.com:8200/v1/pki/
       crl"
```

4. **Créer un rôle pour les certificats :** Un rôle définit les domaines autorisés, les sous-domaines et la durée maximale de vie des certificats.

```
vault write pki/roles/example-dot-com \
    allowed_domains="example.com" \
    allow_subdomains=true \
    max_ttl="720h"
```

5. **Émettre un certificat :** On peut générer un certificat pour un service ou une application.

```
vault write pki/issue/example-dot-com \
    common_name="app.example.com" \
    ttl="24h"
```

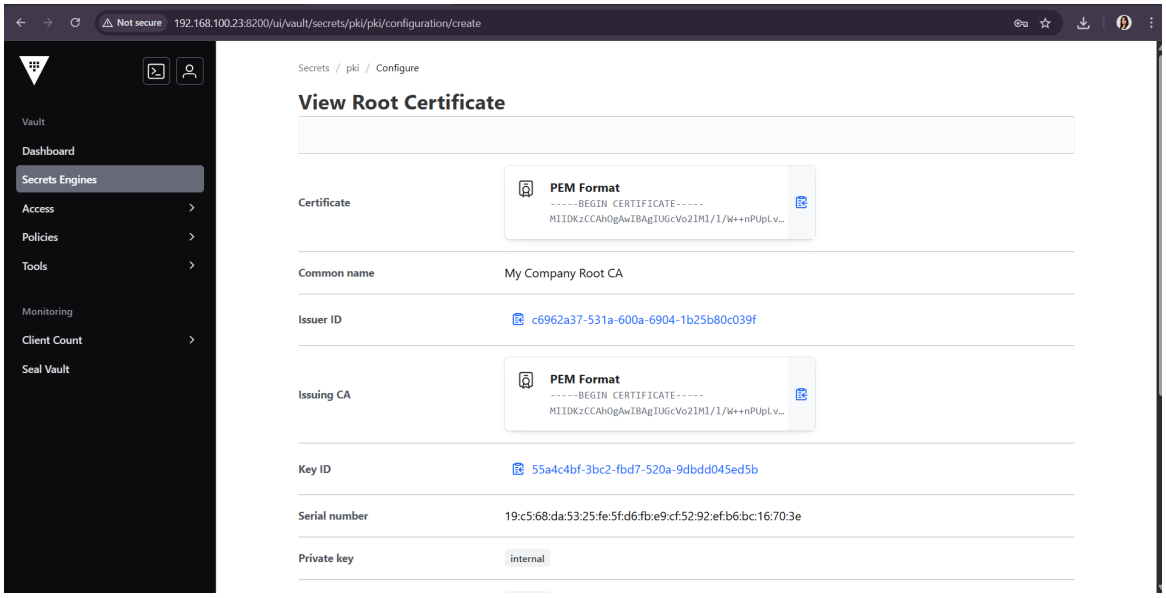


Figure 14.1: Génération d'un certificat avec le moteur PKI de Vault

14.2 SSH Secrets Engine

Vault peut gérer des certificats SSH signés, permettant d'authentifier des utilisateurs ou des machines sans gérer de clés statiques.

14.2.1 Configuration des certificats SSH

1. **Activer le moteur SSH :**

```
vault secrets enable ssh
```

2. **Configurer une CA SSH :** Vault génère une clé de signature qui servira à signer les clés publiques des utilisateurs.

```
vault write ssh/config/ca generate_signing_key=true
```

3. **Créer un rôle SSH :** Le rôle définit qui peut obtenir des certificats, leur durée et les extensions autorisées.

```
vault write ssh/roles/admin \
    key_type=ca \
    ttl=8h \
    max_ttl=24h \
```

```
    allow_user_certificates=true \
    allowed_users="*" \
    allowed_extensions="permit-pty,permit-port-forwarding" \
    default_extensions="permit-pty=,permit-port-forwarding="
```

4. **Signer une clé publique :** L'utilisateur soumet sa clé publique et Vault retourne un certificat signé.

```
vault write ssh/sign/admin public_key=@$HOME/.ssh/id_rsa.pub
```

5. **Connexion SSH avec certificat :** Le certificat signé peut être utilisé pour se connecter à la machine distante.

```
ssh -i ~/.ssh/id_rsa -i ~/.ssh/id_rsa-cert.pub user@server
```

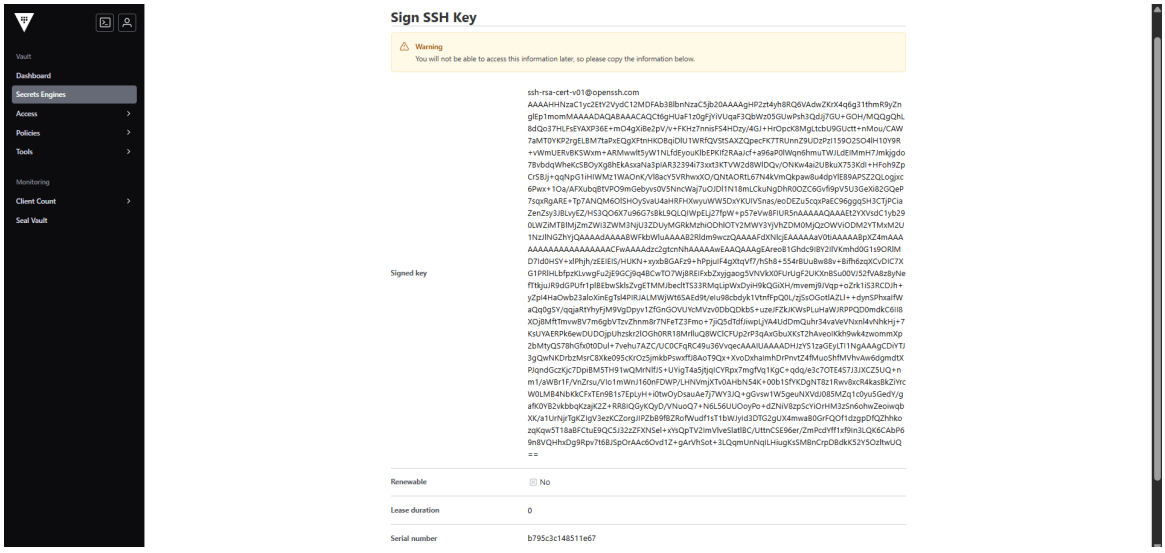


Figure 14.2: Signature d'une clé SSH via Vault

14.3 TOTP (Time-based One-Time Password)

Vault peut également générer des OTP (One-Time Password) pour l'authentification multifacteur.

14.3.1 Configuration TOTP

1. **Activer le moteur TOTP :**

```
vault secrets enable totp
```

2. **Créer une clé TOTP pour un utilisateur/service :**

```
vault write totp/keys/aws-mfa \
    issuer=AWS \
    account_name=maryem@example.com \
    generate=true
```

3. **Obtenir le QR Code :** L'utilisateur peut scanner le code dans une application Authenticator (Google Authenticator, Authy...)

```
vault read -field=barcode totp/keys/aws-mfa
```

4. **Générer un code TOTP :**

```
vault read totp/code/aws-mfa
```

Chapter 15

Conformité et Audit

La conformité et l'audit constituent des piliers essentiels de la sécurité des systèmes d'information. Dans un environnement manipulant des secrets sensibles (clés API, mots de passe, certificats), il est indispensable de disposer d'une traçabilité complète des accès et des opérations effectuées.

HashiCorp Vault intègre un mécanisme d'audit natif permettant d'enregistrer toutes les requêtes traitées par le serveur, qu'elles soient autorisées ou refusées. Ces journaux d'audit sont indispensables pour répondre aux exigences de normes telles que ISO 27001, SOC 2 ou PCI-DSS.

15.1 Configuration de l'audit

15.1.1 Principe des audit devices

Un *audit device* dans Vault est une destination vers laquelle toutes les requêtes et réponses du serveur sont enregistrées sous forme de logs structurés au format JSON. Vault permet de configurer plusieurs audit devices simultanément afin de répondre aux besoins de sécurité, de redondance et d'intégration avec des outils tiers (SIEM).

15.1.2 Multiples destinations d'audit

Dans cette configuration, trois types de destinations d'audit sont activés :

- un fichier local pour l'archivage,
- un service syslog pour la centralisation système,
- un socket réseau pour l'intégration avec un SIEM.

Listing 15.1: Configurer plusieurs audit devices

```
# Audit vers un fichier local
vault audit enable file file_path=/var/log/vault/audit.log

# Audit vers syslog
vault audit enable syslog tag="vault" facility="LOCAL7"

# Audit via socket (int gration SIEM)
vault audit enable socket \
  address=siem.example.com:514 \
  socket_type=tcp
```

```
11
12 # V rifier les audit devices actifs
13 vault audit list -detailed
```

Cette approche multi-destination garantit que les événements critiques sont conservés même en cas de défaillance d'un des mécanismes de journalisation.

=

La figure **??** illustre les audit devices actifs, confirmant que Vault enregistre correctement les événements sur plusieurs supports.

15.2 Analyse des logs d'audit

Les logs d'audit générés par Vault sont au format JSON, ce qui facilite leur analyse à l'aide d'outils tels que `jq`. Cette analyse permet d'identifier les comportements anormaux, les accès non autorisés et les tentatives d'attaque.

15.2.1 Exemples d'analyses avec jq

Listing 15.2: Analyser les logs d'audit avec jq

```
1  # Afficher toutes les op rations d'un utilisateur donn
2  cat audit.log | jq 'select(.auth.display_name=="john")'
3
4  # Filtrer les op rations sur un chemin sp cifique
5  cat audit.log | jq 'select(.request.path | contains("secret/prod"))'
6
7  # Identifier les op rations ayant   chou
8  cat audit.log | jq 'select(.error != "")'
9
10 # Top 10 des utilisateurs les plus actifs
11 cat audit.log | jq -r '.auth.display_name' | sort | uniq -c | sort -rn |
       head -10
12
13 # D tecter des acc s hors heures de bureau
14 cat audit.log | jq 'select(.time | fromdateiso8601 | strftime("%H") |
       tonumber < 8 or tonumber > 18)'
```

Ces requêtes permettent de détecter rapidement des comportements suspects tels que des accès inhabituels ou des tentatives répétées échouées.

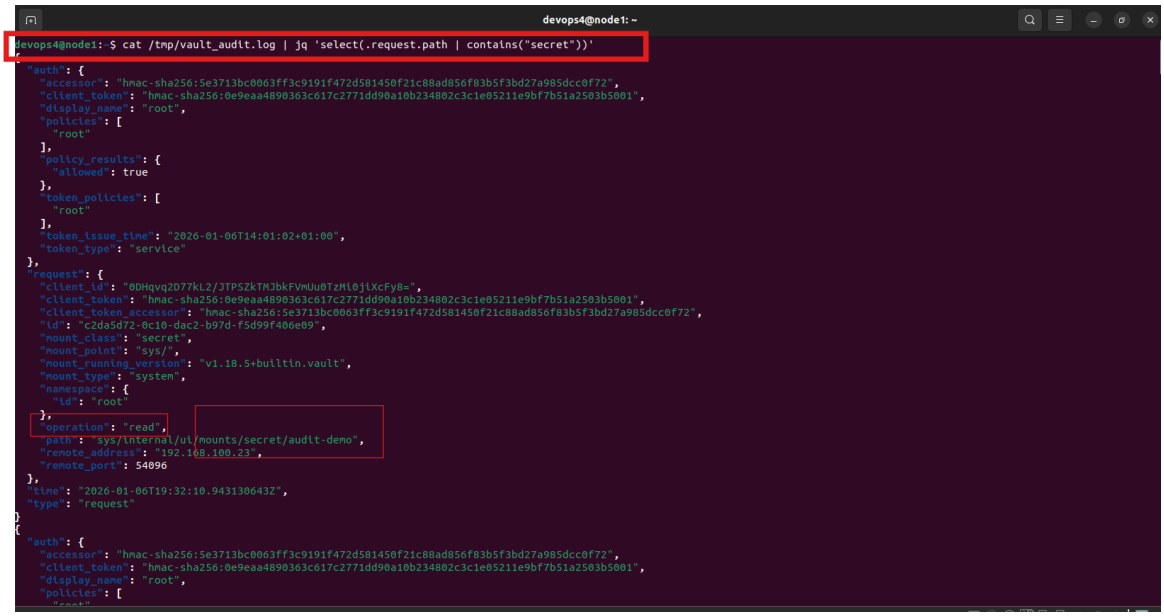


Figure 15.1: Analyse des logs d'audit Vault à l'aide de jq

15.3 Rapports de conformité

Afin de faciliter les audits périodiques, il est utile d'automatiser la génération de rapports de conformité à partir des logs d'audit. Le script Python suivant analyse les journaux Vault et produit un rapport synthétique au format HTML.

15.3.1 Génération automatique de rapports

Listing 15.3: Script de génération de rapport de conformité

```python
#!/usr/bin/env python3
import json
import sys
from datetime import datetime
from collections import Counter

def analyze_audit_log(log_file):
    """Analyse le log d'audit Vault"""

    stats = {
        'total_operations': 0,
        'failed_operations': 0,
        'users': Counter(),
        'paths': Counter(),
        'operations': Counter(),
        'errors': []
    }

    with open(log_file, 'r') as f:
        for line in f:
            try:
                entry = json.loads(line)
                stats['total_operations'] += 1
```

```
24
25                  user = entry.get('auth', {}).get('display_name', 'unknown'
                        )
26                  stats['users'][user] += 1
27
28                  path = entry.get('request', {}).get('path', 'unknown')
29                  stats['paths'][path] += 1
30
31                  op = entry.get('request', {}).get('operation', 'unknown')
32                  stats['operations'][op] += 1
33
34                  if entry.get('error'):
35                      stats['failed_operations'] += 1
36                      stats['errors'].append({
37                          'time': entry.get('time'),
38                          'user': user,
39                          'path': path,
40                          'error': entry.get('error')
41                      })
42              except json.JSONDecodeError:
43                  continue
44
45      return stats
46
47  def generate_report(stats):
48      """G  n  re  un rapport HTML de conformit """
49
50      html = f"""
51      <html>
52      <head>
53          <title>Rapport de Conformit  Vault</title>
54          <style>
55              body {{ font-family: Arial, sans-serif; margin: 20px; }}
56              table {{ border-collapse: collapse; width: 100%; }}
57              th, td {{ border: 1px solid #ddd; padding: 10px; }}
58              th {{ background-color: #3498db; color: white; }}
59          </style>
60      </head>
61      <body>
62          <h1>Rapport de Conformit  Vault</h1>
63          <p>G  n  r   le : {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}</
               p>
64
65          <h2>Statistiques  G n rales </h2>
66          <ul>
67              <li>Total d'op rations : {stats['total_operations']}</li>
68              <li>Op rations    choues    : {stats['failed_operations']}</li>
69          </ul>
70
71          <h2>Utilisateurs les plus actifs</h2>
72          <table>
73              <tr><th>Utilisateur</th><th>Op rations </th></tr>
74      """
75
76      for user, count in stats['users'].most_common(10):
77          html += f"<tr><td>{user}</td><td>{count}</td></tr>"
78
79      html += "</table></body></html>"
```

```
    return html

if __name__ == '__main__':
    stats = analyze_audit_log(sys.argv[1])
    report = generate_report(stats)
    with open('compliance_report.html', 'w') as f:
        f.write(report)
```

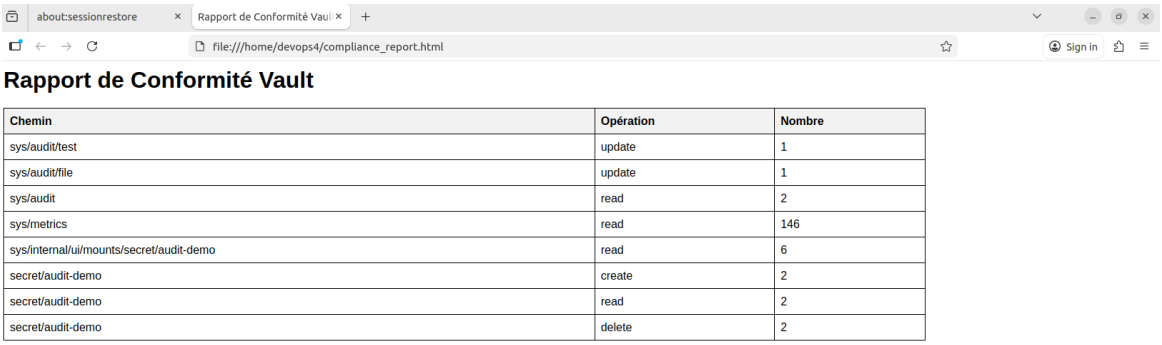


Figure 15.2: Rapport de conformité HTML généré automatiquement

La figure 15.2 montre un exemple de rapport de conformité généré, facilitant les audits de sécurité et la démonstration de conformité réglementaire.

Chapter 16

Commandes Utiles

Ce chapitre regroupe les principales commandes de HashiCorp Vault utilisées au quotidien par les administrateurs systèmes et les ingénieurs DevSecOps. Ces commandes permettent de superviser l'état du serveur, de gérer les secrets, les politiques de sécurité, l'authentification ainsi que les mécanismes d'audit.

16.1 Commandes CLI essentielles

Les commandes suivantes constituent la base de l'administration de Vault via l'interface en ligne de commande (CLI). Elles couvrent les opérations les plus courantes liées à la supervision, à la gestion des accès et à la sécurité.

Listing 16.1: Commandes Vault essentielles

```
# V rification de l' tat  et de la sant  du serveur
vault status
vault read sys/health

# Gestion des secrets (KV engine)
vault kv put secret/path key=value       # Cr ation ou mise    jour
    d un  secret
vault kv get secret/path                 # Lecture  d un  secret
vault kv delete secret/path              # Suppression  d un  secret
vault kv list secret/                    # Liste des secrets disponibles

# Authentification et gestion des tokens
vault login                              # Authentification    Vault
vault token lookup                       # Informations sur le token
   courant
vault token renew                        # Renouvellement du token

# Gestion des politiques de s curit  (ACL)
vault policy list                        # Liste des policies existantes
vault policy read policy-name            # Lecture  d une  policy
vault policy write policy-name policy.hcl# Cr ation ou mise    jour
    d une  policy

# Configuration et gestion de  l audit
vault audit list                         # Liste des audit devices actifs
vault audit enable file file_path=/var/log/vault/audit.log

# Gestion des moteurs de secrets
```

```
vault secrets list                          # Liste des secrets engines activ s
vault secrets enable -path=custom database
vault secrets disable custom/

# Gestion des m thodes   d  authentification
vault auth list                             # M thodes   d  authentification disponibles
vault auth enable userpass
vault auth disable userpass/
```

Chapter 17

Guide de Dépannage

Ce chapitre présente les problèmes les plus fréquemment rencontrés lors de l'utilisation de HashiCorp Vault, ainsi que les solutions recommandées. Il constitue une référence rapide pour le diagnostic et la résolution des incidents opérationnels.

17.1 Problèmes courants et solutions

Problème	**Solution recommandée**
Vault scellé (Sealed)	Exécuter la commande `vault operator unseal` et fournir le nombre requis de clés de déchiffrement afin de rendre Vault opérationnel.
Permission denied	Vérifier les capacités du token à l'aide de la commande `vault token capabilities <token> <path>` et ajuster la policy si nécessaire.
Connexion refusée	Vérifier que le serveur Vault est en cours d'exécution et que la variable d'environnement `VAULT_ADDR` pointe vers la bonne adresse.
Erreur de certificat TLS	Contrôler la validité des certificats TLS. En environnement de test, il est possible de désactiver temporairement la vérification avec `VAULT_SKIP_VERIFY=true`.
Token expiré	Renouveler le token avec `vault token renew` ou effectuer une nouvelle authentification.

Table 17.1: Guide de dépannage des problèmes courants rencontrés avec Vault

Chapter 18

Glossaire

Cette section définit les principaux termes techniques utilisés dans l'écosystème HashiCorp Vault afin de faciliter la compréhension des concepts abordés dans ce document.

Seal / Unseal État de protection de Vault. Lorsqu'il est scellé, les données chiffrées sont inaccessibles. Le déverrouillage (Unseal) permet de charger la clé maître en mémoire.

Token Jeton d'authentification utilisé pour accéder à Vault, associé à des politiques de sécurité définissant les permissions.

Lease Durée de validité d'un secret dynamique, après laquelle celui-ci expire automatiquement.

Policy Ensemble de règles de contrôle d'accès (ACL) définissant les actions autorisées sur des chemins Vault spécifiques.

Secret Engine Composant responsable de la génération, du stockage et de la gestion des secrets (KV, Database, AWS, etc.).

Root Token Token initial disposant de privilèges complets, généré lors de l'initialisation de Vault.

AppRole Méthode d'authentification destinée aux applications automatisées et aux workloads non humains.

Transit Moteur de chiffrement fournissant des services de cryptographie sans stocker les données sensibles.

PKI Infrastructure de gestion des clés publiques permettant la génération et la gestion de certificats TLS.

Namespace (Vault Enterprise) Mécanisme d'isolation logique permettant de séparer plusieurs environnements au sein d'un même cluster.

Replication (Vault Enterprise) Fonctionnalité permettant la réplication des données entre plusieurs clusters Vault.

Fin du voyage, début de l'aventure

Cette documentation est le fruit d'une exploration passionnée dans l'univers fascinant de la cybersécurité.
Chaque page reflète un engagement envers la rigueur, l'innovation et la soif d'apprendre.

— Maryem Cherif —

« La sécurité n'est pas une destination, c'est un voyage continu. »