Projet : Document d'Architecture Technique (DAT) Portail d'Acccès Zero Trust Access d'une Entreprise

PKI- IAM - Cryptographie

Réalisé par :

Joseline YOUEGO

Table des matières

1 Description du projet

1.1 Scénario fonctionnel du projet SecureCorp Zero Trust

1.1.1 Objectif général

L'objectif de ce projet est de construire une preuve de concept (*Proof of Concept*, POC) démontrant la maîtrise d'une architecture de sécurité moderne dite Zero Trust. Il ne s'agit pas uniquement d'installer des composants techniques (Vault, Keycloak, Nginx), mais de montrer leur intégration cohérente dans une chaîne de confiance complète, automatisée et documentée.

Le projet se déroule dans un contexte fictif : l'entreprise **SecureCorp**. Cette entreprise dispose d'une application interne sensible (un portail employé), utilisée aussi bien au bureau qu'en télétravail.

Problématique de départ. Historiquement, l'application était accessible uniquement sur le réseau interne, avec une gestion locale des comptes utilisateurs et des mots de passe, sans chiffrement systématique des communications. Cette situation n'est plus acceptable au regard des exigences de sécurité actuelles.

En tant qu'ingénieur sécurité de SecureCorp, la mission est la suivante :

- permettre aux employés d'accéder au portail depuis l'extérieur (télétravail, mobilité);
- supprimer les logins locaux dans l'application au profit d'une authentification centralisée;
- chiffrer l'ensemble des communications;
- appliquer le principe du moindre privilège pour la gestion des secrets (notamment les accès à la base de données).

1.1.2 Principes de sécurité appliqués

Trois règles d'or structurent l'architecture cible :

1. **Nul ne passe sans identité (IAM).** L'application métier ne gère plus directement les mots de passe. Toute authentification passe par un fournisseur d'identité

centralisé (Keycloak), via un protocole standard (OIDC / OAuth2). L'application délègue la vérification de l'identité et ne reçoit que des jetons.

2. **Chiffrement partout (PKI).** Les communications entre les différents composants sont systématiquement chiffrées. Une PKI interne basée sur Vault émet les certificats utilisés pour :
   - l'exposition HTTPS du portail via la passerelle (Nginx) ;
   - les échanges internes entre services, à terme en *mutual TLS* (mTLS).
3. **Moindre privilège et secrets dynamiques.** Les secrets sensibles (en particulier les identifiants de base de données de Keycloak) ne sont pas stockés en clair dans des fichiers de configuration. Ils sont générés à la demande par Vault (moteur `database`), avec un TTL limité. Keycloak ne connaît donc ses identifiants que lors du démarrage, et ceux-ci expirent automatiquement.

1.1.3 Rôle des machines virtuelles dans le scénario

L'architecture réseau décrite à la section 1.7.1 est complétée par un scénario fonctionnel réparti sur trois machines virtuelles principales :

1. `srv-sec-core` – le coffre-fort de SecureCorp (192.168.56.10).

- **HashiCorp Vault** :
  - joue le rôle d'Autorité de Certification (Root CA et Intermediate CA) pour l'émission de certificats internes ;
  - fournit des secrets dynamiques pour PostgreSQL via le moteur `database` (création d'utilisateurs temporaires pour Keycloak).
- **Keycloak** :
  - centralise la gestion des identités et des rôles ;
  - authentifie les utilisateurs du portail et fournit des jetons OIDC à l'application.

2. `srv-gateway` – la forteresse exposée (192.168.56.30). Il Héberge un reverse proxy **Nginx** qui constitue l'unique point d'entrée depuis l'extérieur; il termine également les connexions `HTTPS` (certificats internes émis par la PKI Vault) et enfin redirige les requêtes vers les services internes :

- le portail applicatif sur `srv-app-prod` ;
- l'interface Keycloak sur `srv-sec-core`.

3. `srv-app-prod` – **l'application métier (192.168.56.20).**

   Il héberge le **portail employé** (application Python (flask) / Gunicorn) et ne gère pas directement l'authentification :

   – lorsqu'un utilisateur souhaite se connecter, l'application le redirige vers Keycloak ;

   – après authentification réussie, l'application reçoit un code d'autorisation puis un jeton, qu'elle valide avant d'ouvrir une session.

1.1.4 Parcours utilisateur : du navigateur à l'application

Le scénario utilisateur complet peut être résumé ainsi :

1. **Accès initial au portail.** Depuis son poste, un employé saisit l'URL du portail interne dans son navigateur :

   `https://portal.securecorp.local`

   La requête atteint la passerelle `srv-gateway`, qui présente un certificat interne émis par la PKI. Le navigateur affiche éventuellement un avertissement de certificat autosigné, normal dans le cadre du laboratoire.

2. **Demande de connexion.** Sur la page d'accueil du portail, l'utilisateur clique sur un bouton Se connecter via SSO. L'application ne lui demande pas de mot de passe local ; elle déclenche un flux OIDC.

3. **Redirection vers l'Identity Provider.** Le reverse proxy redirige l'utilisateur vers Keycloak sur `srv-sec-core`. Le navigateur affiche alors la page de login fournie par Keycloak (realm SecureCorp).

4. **Authentification centralisée.** L'utilisateur saisit ses identifiants sur l'interface Keycloak. Keycloak vérifie ces informations, éventuellement avec un second facteur (MFA) dans une extension future du projet.

5. **Retour vers le portail avec un code.** En cas de succès, Keycloak redirige le navigateur vers le portail avec un *code d'autorisation* (flux OAuth2 / OIDC de type authorization code).

6. **Échange de code contre un jeton.** De manière transparente pour l'utilisateur, le portail contacte ensuite Keycloak en arrière-plan pour échanger ce code contre :

   - un *ID Token* (identité de l'utilisateur) ;
   - un *Access Token* éventuellement utilisé pour des appels API.

   L'application vérifie la signature et les informations du jeton (audience, date d'expiration, rôles).

7. **Ouverture de session applicative.** Si le jeton est valide, l'application considère l'utilisateur comme authentifié et ouvre une session. Les pages protégées du portail sont alors accessibles, avec un affichage conditionnel en fonction des rôles (par exemple `ROLE_USER` vs `ROLE_ADMIN`).

1.1.5 Lien avec la méthodologie et l'état d'avancement

La section *Méthodologie utilisée* décrit comment cette infrastructure a été mise en place de manière reproductible à l'aide de Vagrant et d'Ansible (Infrastructure as Code). Ce scénario fonctionnel sert de fil conducteur aux différentes phases techniques décrites dans le mémoire :

- Phase 1 : mise en place du coeur de sécurité (`srv-sec-core`) avec Vault en tant que PKI et gestionnaire de secrets dynamiques ;
- Phase 2 : installation et intégration de Keycloak avec Vault et PostgreSQL ;
- Phases suivantes : configuration de la passerelle `srv-gateway`, sécurisation des flux en HTTPS et intégration OIDC dans l'application sur `srv-app-prod`.

Certaines étapes, comme l'échange final du code d'autorisation contre un jeton (et les erreurs 502 éventuellement rencontrées dans les premières intégrations), sont précisément le reflet de la complexité réelle de ce type d'architecture. Elles sont documentées dans les chapitres techniques et, le cas échéant, dans un *post-mortem* détaillé, afin de montrer la démarche de diagnostic et de correction adoptée.

1.2 Déploiement de l'infrastructure virtuelle

1.2.1 Lancement des machines avec `vagrant up`

Après avoir placé le fichier `Vagrantfile` à la racine du projet, nous lançons l'infrastructure avec la commande suivante :

```
vagrant up
```

La figure 1.1 montre la sortie de cette commande, avec le téléchargement de l'image Ubuntu et la création des trois machines virtuelles.

Figure 1.1: Exécution de la commande `vagrant up` et création des VMs.

1.2.2 Vérification de l'état des machines avec `vagrant status`

Une fois le déploiement terminé, nous vérifions l'état des machines avec :

```
vagrant status
```

La figure 1.2 confirme que les trois machines `sec-core`, `app-prod` et `gateway` sont en état `running`.

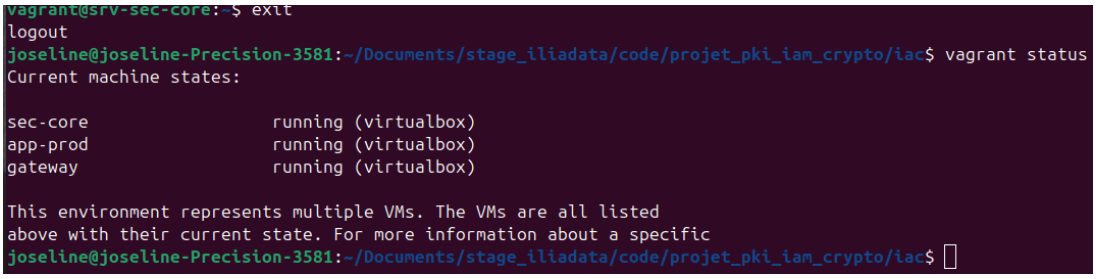

Figure 1.2: Vérification de l'état des VMs avec `vagrant status`.

1.2.3 Connexion SSH à la machine `sec-core`

Pour vérifier la connectivité et la configuration réseau, nous nous connectons à la machine `sec-core` :

```
vagrant ssh sec-core
```

Puis nous testons la résolution de noms avec un ping vers la gateway :

```
ping srv-gateway
```

La figure 1.3 illustre la session SSH et le succès du ping vers `srv-gateway`, preuve que le DNS local (`/etc/hosts`) est correctement configuré.

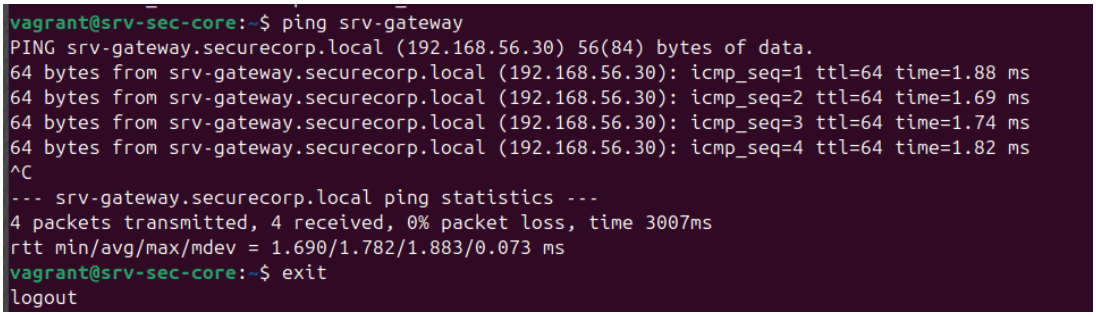

Figure 1.3: Connexion SSH à `sec-core` et test de connectivité vers `srv-gateway`.

1.3 Phase 1 : Mise en place du coeur de sécurité sur `srv-sec-core`

L'objectif de cette première phase est de transformer la machine virtuelle `srv-sec-core` en véritable coeur de sécurité du laboratoire SecureCorp. Cette VM héberge à la fois le gestionnaire de secrets HashiCorp Vault et l'infrastructure de gestion des certificats (PKI) à deux niveaux (Autorité Racine et Autorité Intermédiaire).

1.3.1 Installation de HashiCorp Vault

Après connexion à la VM via Vagrant :

```
vagrant ssh sec-core
```

nous procédons à l'installation de Vault en utilisant le dépôt officiel HashiCorp. L'utilisation de ce dépôt, plutôt que des paquets Ubuntu par défaut, permet de disposer d'une version à jour, conforme aux environnements de production.

Les étapes sont les suivantes :

- Installation des prérequis et de la clé GPG HashiCorp ;
- Ajout du dépôt `apt.releases.hashicorp.com` au système ;
- Installation du paquet `vault` via `apt-get`.

La figure 1.4 illustre l'installation réussie de Vault sur la machine `srv-sec-core`.

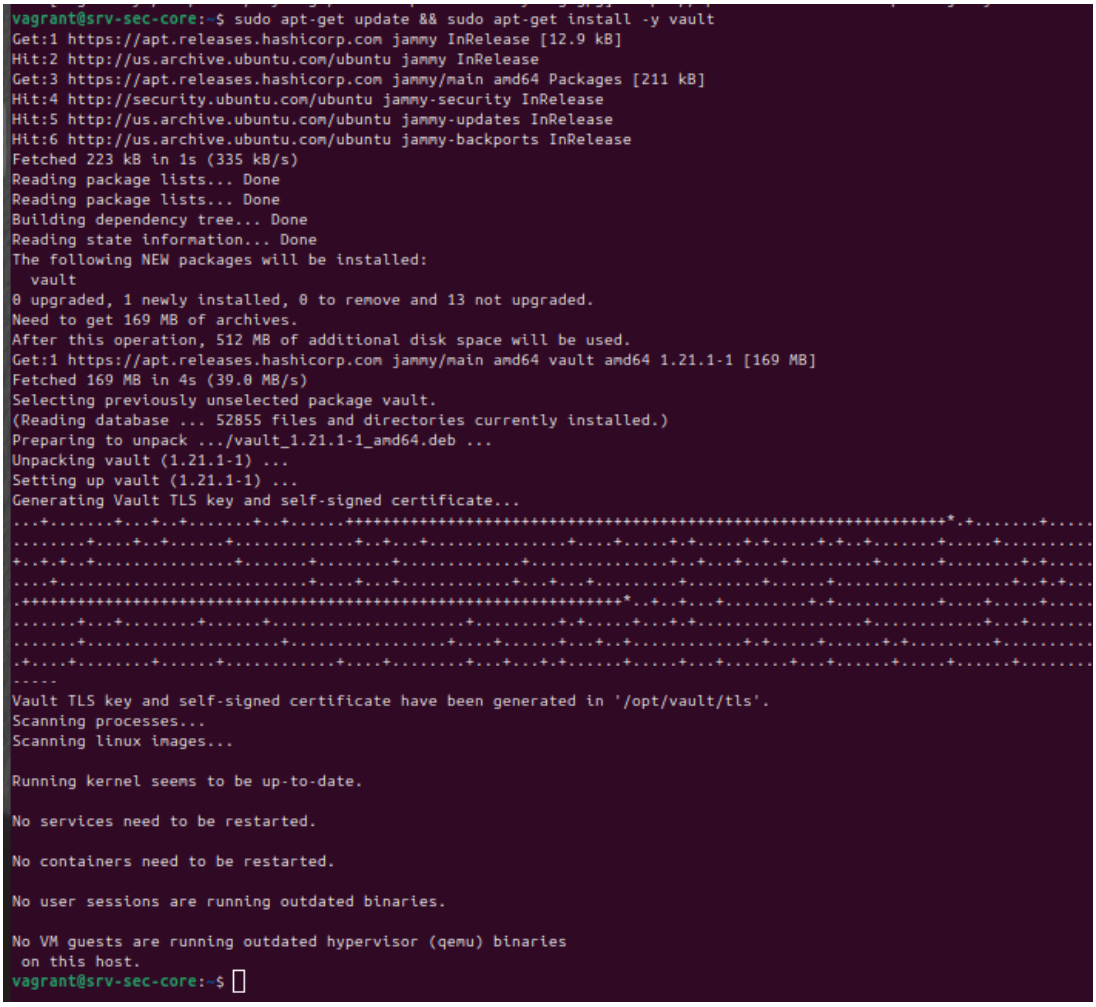

Figure 1.4: Installation de HashiCorp Vault depuis le dépôt officiel HashiCorp.

1.3.2 Configuration de Vault en service `systemd`

Afin de se rapprocher des pratiques de gestion de services en entreprise, Vault n'est pas lancé manuellement en ligne de commande, mais configuré comme service `systemd`. Un fichier d'unité `/etc/systemd/system/vault.service` est créé avec, notamment, la directive `ExecStart` suivante :

```
/usr/bin/vault server -dev \
  -dev-listen-address="0.0.0.0:8200" \
  -dev-root-token-id="root"
```

Dans le cadre de ce laboratoire, Vault est démarré en mode *dev* et écoute sur toutes les interfaces réseau (`0.0.0.0:8200`) afin de permettre les échanges avec les autres VMs. Un token racine fixe (`root`) est utilisé pour simplifier les manipulations ; il s'agit évidemment d'un compromis pédagogique et non d'une configuration acceptable en production.

Le service est ensuite activé et démarré via :

```
sudo systemctl enable vault
sudo systemctl start vault
```

```
sudo systemctl status vault
```

Lors de l'exécution de `systemctl status`, la sortie est affichée à travers un *pager* (généralement `less`), ce qui est indiqué par la mention lines 1–12 (END) en bas de l'écran. Pour revenir au shell, il suffit d'appuyer sur la touche `q`. La figure 1.5 montre l'état active (running) du service Vault, confirmant son bon fonctionnement.

```
vagrant@srv-sec-core:~$ vault secrets enable database
Success! Enabled the database secrets engine at: database/
vagrant@srv-sec-core:~$ vault write database/config/postgresql \
    plugin_name=postgresql-database-plugin \
    allowed_roles="keycloak-role" \
    connection_url="postgresql://{{username}}:{{password}}@localhost:5432/keycloak" \
    username="vault" \
    password="vault-root-password"
Success! Data written to: database/config/postgresql
vagrant@srv-sec-core:~$ vault write database/roles/keycloak-role \
    db_name=postgresql \
    creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}'; GRANT ALL PRIVI
LEGES ON DATABASE keycloak TO \"{{name}}\"; GRANT ALL ON SCHEMA public TO \"{{name}}\";" \
    default_ttl="1h" \
    max_ttl="24h"
Success! Data written to: database/roles/keycloak-role
vagrant@srv-sec-core:~$ vault read database/creds/keycloak-role
Key                Value
---                -----
lease_id           database/creds/keycloak-role/mo6dK3ZJnU3kEaFYdyPhJJ7a
lease_duration     1h
lease_renewable    true
password           jbOd--8Jgo1wWYm6ZKis
username           v-token-keycloak-vQDmhBe5MPOG1lNsfkM2-1763773701
vagrant@srv-sec-core:~$
```

Figure 1.5: Service `vault` géré par `systemd` et en état active (running).

1.3.3 Préparation de l'environnement et installation de `jq`

Pour pouvoir interagir confortablement avec Vault, deux variables d'environnement sont définies de manière persistante dans le fichier `~/.bashrc` :

- `VAULT_ADDR` : URL du serveur Vault (`http://127.0.0.1:8200`) ;
- `VAULT_TOKEN` : token d'authentification utilisé (`root` dans ce laboratoire).

Après mise à jour du fichier de configuration et rechargement avec `source ~/.bashrc`, la commande suivante permet de vérifier l'état du serveur :

```
vault status
```

Par ailleurs, la construction de la PKI repose sur l'utilisation de l'outil `jq`, qui permet d'extraire proprement les champs renvoyés au format JSON par Vault (par exemple les CSR et certificats). Cet outil n'étant pas installé par défaut, il est ajouté via :

```
sudo apt-get install -y jq
```

La figure 1.6 présente la sortie de la commande `vault status`, indiquant un serveur non scellé (Sealed: false) et prêt à émettre des certificats.

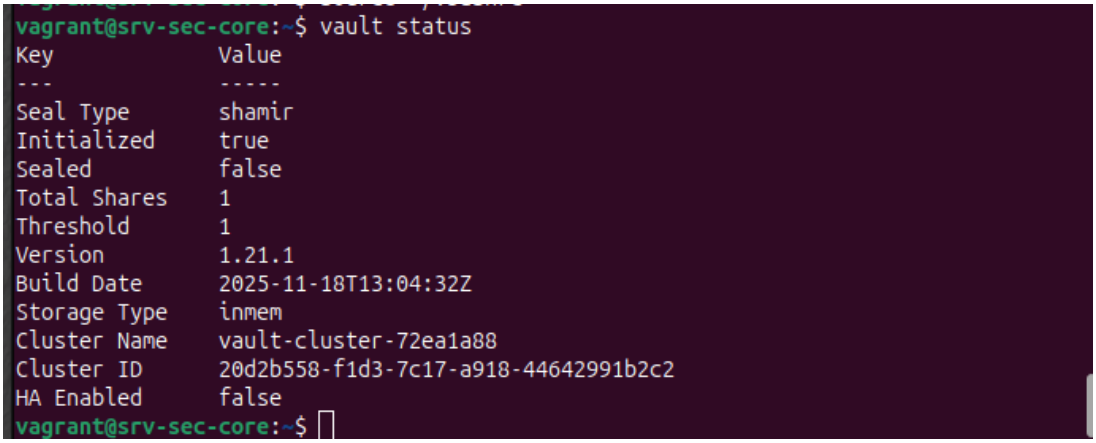

Figure 1.6: Vérification de l'état de Vault avec la commande `vault status`.

1.3.4 Mise en place de la PKI à deux niveaux

Une fois Vault opérationnel, nous configurons une infrastructure de gestion de certificats (PKI) à deux niveaux, composée d'une Autorité Racine (Root CA) et d'une Autorité Intermédiaire (Intermediate CA). Cette architecture est conforme aux bonnes pratiques de sécurité : la racine est très peu utilisée et peut rester hors ligne, tandis que l'intermédiaire signe les certificats utilisés au quotidien et peut être révoquée si nécessaire.

1.3.4.1 Création de l'Autorité Racine

Nous activons tout d'abord un moteur de secrets de type PKI monté sur le chemin `pki`, puis nous ajustons le TTL maximal à 10 ans :

```
vault secrets enable pki
vault secrets tune -max-lease-ttl=87600h pki
```

Un certificat racine interne est ensuite généré avec le nom commun SecureCorp Root CA et exporté dans un fichier `root_ca.crt`. Enfin, les URLs de publication des certificats et des listes de révocation (CRL) sont configurées pour permettre aux clients d'aller vérifier la validité des certificats émis.

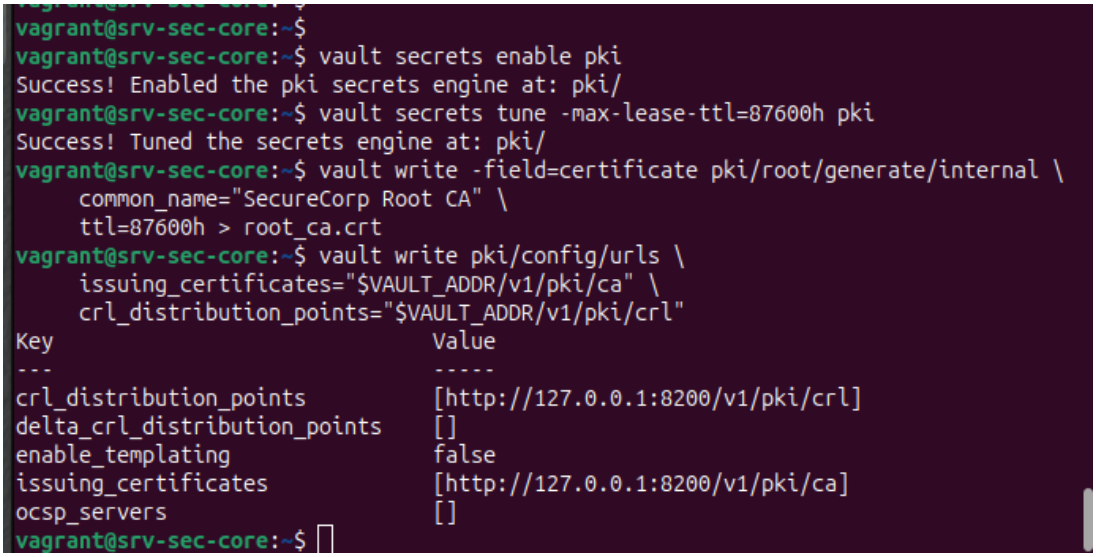

Figure 1.7: Création de la Root CA dans Vault (`pki`) .

1.3.4.2 Création de l'Autorité Intermédiaire

Une seconde instance du moteur PKI est activée sur le chemin `pki_int` pour héberger l'Autorité Intermédiaire. Après avoir ajusté son TTL maximal à un an, une requête de signature de certificat (CSR) est générée :

```
vault write -format=json pki_int/intermediate/generate/internal \
     common_name="SecureCorp Intermediate CA" | \
     jq -r '.data.csr' > pki_intermediate.csr
```

Cette CSR est signée par l'Autorité Racine via la commande `pki/root/sign-intermediate`, puis le certificat intermédiaire obtenu est importé dans le moteur `pki_int`. La figure 1.8 illustre les principales étapes de création et d'enregistrement des autorités racine et intermédiaire dans Vault.

Figure 1.8: Création de l'Intermediate CA dans Vault (`pki_int`).

1.3.4.3 Définition des rôles et émission de certificats serveur

Pour structurer l'émission des certificats, un rôle dédié aux serveurs de domaine `securecorp.local` est créé :

```
vault write pki_int/roles/securecorp-dot-local \
     allowed_domains="securecorp.local" \
     allow_subdomains=true \
     max_ttl="72h"
```

Ce rôle autorise la génération de certificats pour le domaine racine `securecorp.local` ainsi que pour tous ses sous-domaines (par exemple `portal.securecorp.local` ou `idp.securecorp.local`), avec une durée de vie maximale de 72 heures.

Un premier test d'émission de certificat est réalisé avec :

```
vault write pki_int/issue/securecorp-dot-local \
     common_name="test.securecorp.local" ttl="24h"
```

La réponse renvoyée contient le certificat X.509, la clé privée associée et la chaîne de certification complète. Ce test valide la bonne configuration de la PKI basée sur Vault. La figure 1.9 montre le succès de cette opération d'émission de certificat pour le sous-domaine `test.securecorp.local`.

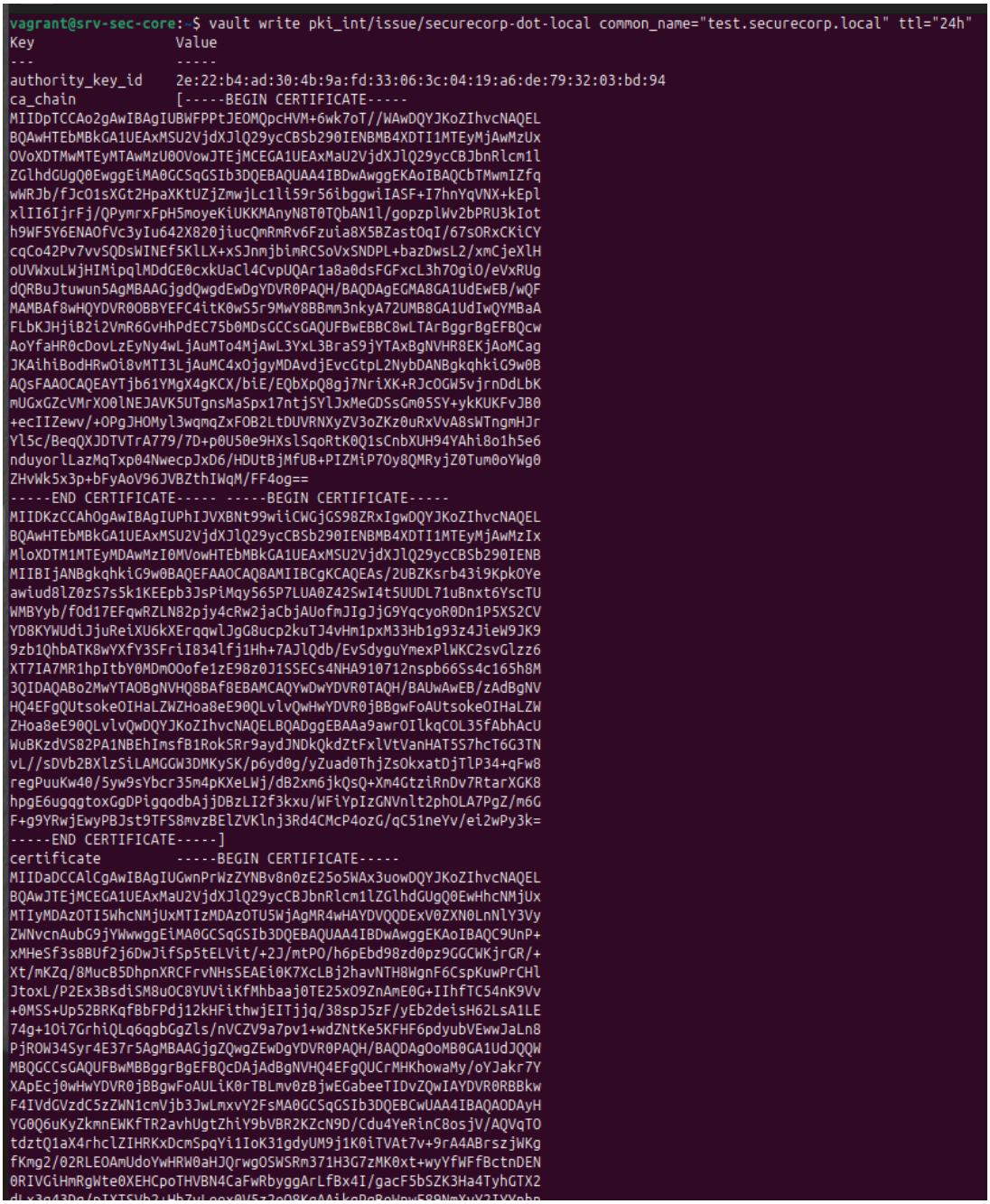

Figure 1.9: Émission d'un certificat pour `test.securecorp.local` via l'Autorité Intermédiaire.

1.3.5 Bilan de la phase 1

À l'issue de cette phase, la VM `srv-sec-core` joue pleinement son rôle de coeur de sécurité :

- Vault est déployé comme service `systemd` et accessible sur le réseau interne du laboratoire.
- Une hiérarchie PKI à deux niveaux (Root CA + Intermediate CA) est opéra-

tionnelle.

- L'émission de certificats pour le domaine `securecorp.local` est fonctionnelle et pilotée par des rôles dédiés.

Cette base sera utilisée dans les phases suivantes pour sécuriser les communications (HTTPS et mTLS) entre la passerelle, le serveur d'application et, plus tard, l'Identity Provider.

1.4 Phase 2 : Mise en place de l'Identity Provider (Keycloak) intégré à Vault

Après avoir mis en place le coeur PKI avec Vault, l'étape suivante consiste à déployer un fournisseur d'identité (Identity Provider, IdP) moderne : Keycloak. L'objectif n'est pas seulement de lancer Keycloak, mais de l'intégrer dans une approche Zero Trust en déléguant à Vault la gestion des identifiants de base de données. Ainsi, Keycloak n'utilise pas un mot de passe statique, mais des identifiants éphémères générés à la demande par Vault.

1.4.1 Préparation de l'environnement : Java 17 et PostgreSQL

Keycloak repose sur la JVM et nécessite une base de données relationnelle pour persister sa configuration et ses comptes. Sur la machine `srv-sec-core`, nous installons donc Java 17 et PostgreSQL, puis créons une base dédiée à Keycloak.

```
sudo apt-get update
sudo apt-get install -y openjdk-17-jdk postgresql postgresql-contrib

sudo -u postgres psql -c "CREATE DATABASE keycloak;"
sudo -u postgres psql -c \
  "CREATE USER vault WITH SUPERUSER ENCRYPTED PASSWORD 'vault-root-password';"
```

Le compte PostgreSQL `vault` joue un rôle particulier : il dispose de droits étendus (`SUPERUSER`) et sert de compte d'administration pour Vault, qui l'utilisera pour créer et révoquer dynamiquement des utilisateurs destinés à Keycloak.

La figure 1.10 illustre l'installation de Java 17 et de PostgreSQL, ainsi que la création de la base `keycloak` et de l'utilisateur `vault`.

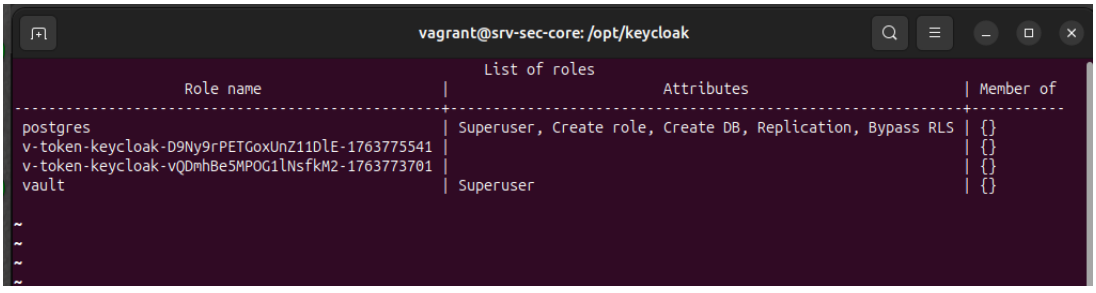

Figure 1.10: Installation de Java 17 et PostgreSQL, création de la base `keycloak` et de l'utilisateur `vault`.

1.4.2 Configuration de Vault comme gestionnaire d'identifiants PostgreSQL

Afin d'éviter d'avoir un mot de passe de base de données écrit en dur dans un fichier de configuration, Vault est configuré pour jouer le rôle de maître des clés pour PostgreSQL. Pour cela, nous activons le moteur de secrets `database` et lui apprenons comment se connecter à la base `keycloak` avec le compte `vault` :

```
vault secrets enable database

vault write database/config/postgresql \
    plugin_name=postgresql-database-plugin \
    allowed_roles="keycloak-role" \
    connection_url="postgresql://{{username}}:{{password}}@localhost:5432/keycloak" \
    username="vault" \
    password="vault-root-password"
```

La directive `allowed_roles="keycloak-role"` restreint les rôles que Vault est autorisé à servir via cette configuration. Nous définissons ensuite le rôle `keycloak-role`, qui décrit la manière dont doivent être créés les utilisateurs SQL éphémères pour Keycloak :

```
vault write database/roles/keycloak-role \
    db_name=postgresql \
    creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}';
                         GRANT ALL PRIVILEGES ON DATABASE keycloak TO \"{{name}}\";
                         GRANT ALL ON SCHEMA public TO \"{{name}}\";" \
    default_ttl="1h" \
    max_ttl="24h"
```

Ce rôle indique à Vault :

- comment créer un nouveau rôle PostgreSQL (`CREATE ROLE`) avec un mot de passe aléatoire ;
- quels droits lui accorder sur la base `keycloak` et son schéma `public` ;
- quelle durée de vie par défaut attribuer aux identifiants générés (1 heure), avec un maximum de 24 heures.

La figure 1.11 montre la configuration du moteur `database` et la création du rôle `keycloak-role`.

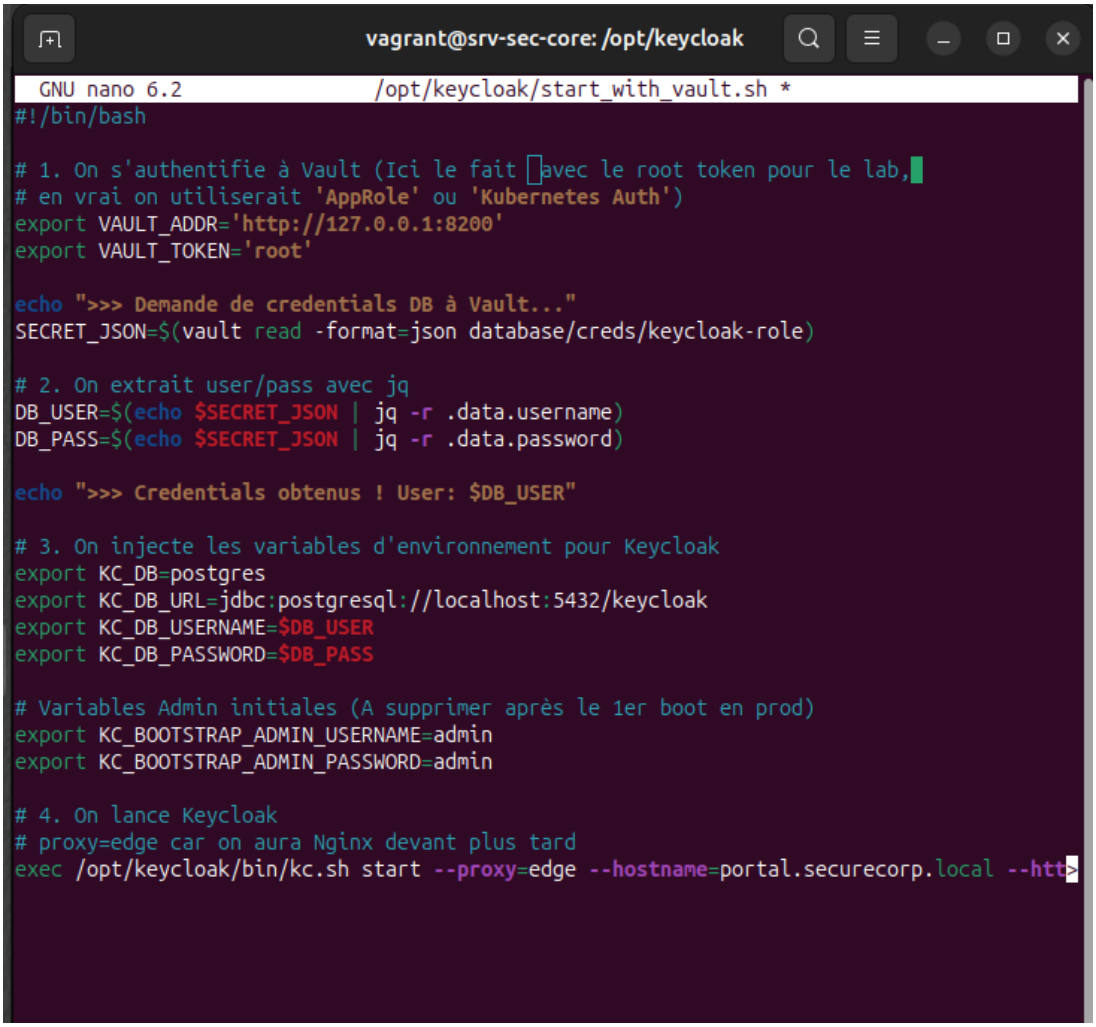

Figure 1.11: Configuration du moteur `database` de Vault pour PostgreSQL et définition du rôle `keycloak-role`.

1.4.2.1 Test de génération d'identifiants éphémères

Pour valider la configuration, nous demandons à Vault de générer une paire identifiant/-mot de passe pour le rôle `keycloak-role` :

```
vault read database/creds/keycloak-role
```

Vault renvoie un document JSON contenant notamment un champ `username` (de la forme `v-root-keycloak-role-xxxxx`) et un `password` aléatoire, ainsi qu'un TTL indiquant la durée de validité des identifiants. Ceci confirme que Vault est capable de créer à la demande des comptes PostgreSQL éphémères pour Keycloak.

La figure 1.12 présente un exemple de sortie de la commande `vault read database/creds/keycloak-role`.

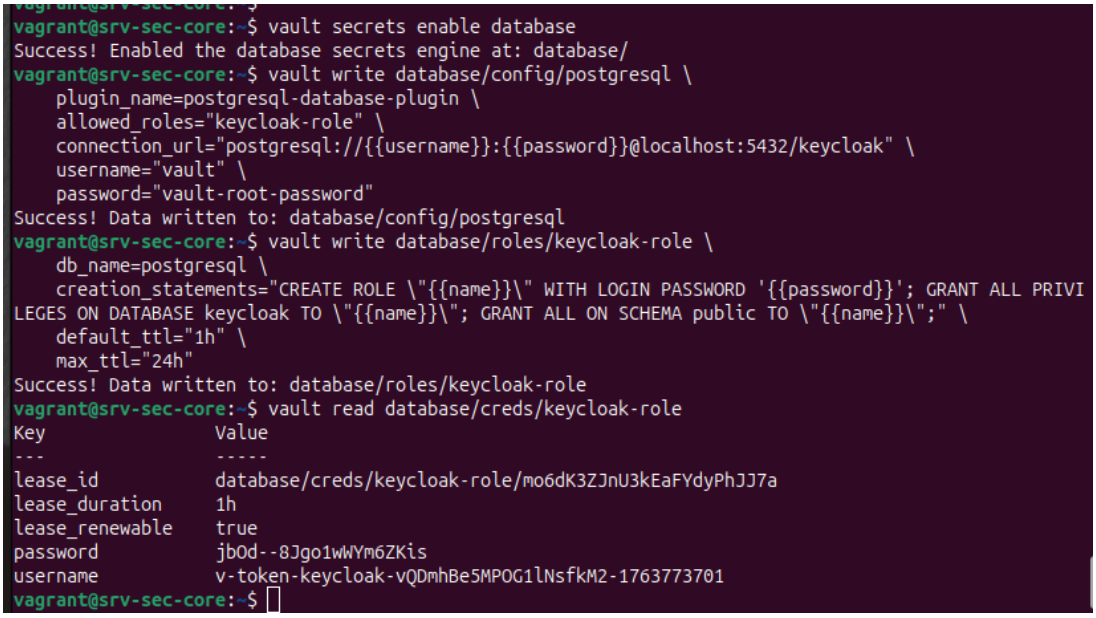

Figure 1.12: Génération d'identifiants PostgreSQL éphémères pour Keycloak via Vault (`database/creds/keycloak-role`).

1.4.3 Installation de Keycloak en mode binaire

Keycloak est installé sous forme d'archive binaire dans le répertoire `/opt`. Nous utilisons une version moderne basée sur Quarkus, et préparons Keycloak pour une utilisation avec PostgreSQL :

```
cd /opt
sudo wget https://github.com/keycloak/keycloak/releases/download/24.0.1/keycloak-24.0
sudo tar -xvzf keycloak-24.0.1.tar.gz
sudo mv keycloak-24.0.1 keycloak
sudo chown -R root:root /opt/keycloak
sudo chmod -R o+rwx /opt/keycloak/

cd /opt/keycloak
bin/kc.sh build --db=postgres
```

La commande `kc.sh build -db=postgres` permet de préconfigurer et optimiser Keycloak pour une base de données PostgreSQL. La figure 1.13 illustre le processus de build de Keycloak.

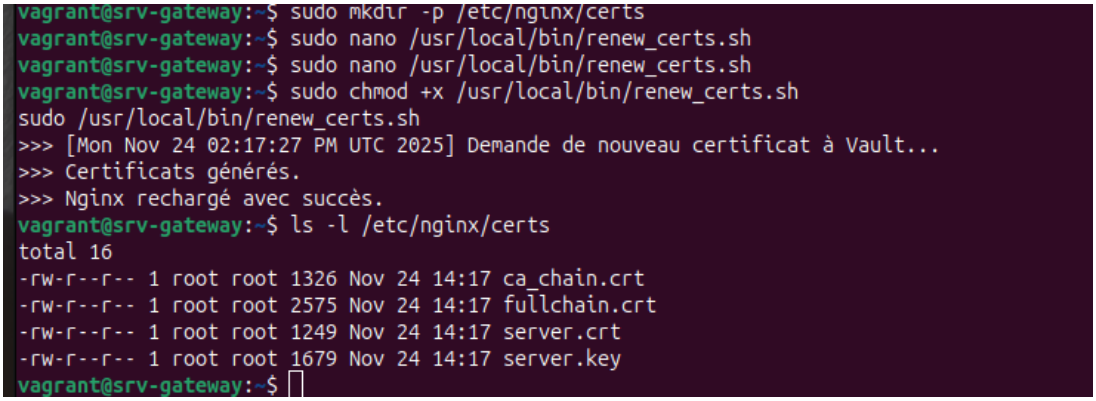

Figure 1.13: Installation et build de Keycloak pour une utilisation avec PostgreSQL.

1.4.4 Script de démarrage intégré à Vault

Keycloak ne gère pas nativement la récupération de ses identifiants de base de données depuis Vault. Pour implémenter ce comportement de manière propre, nous créons un script wrapper `/opt/keycloak/start_with_vault.sh` qui :

1. s'authentifie à Vault ;
2. demande des identifiants PostgreSQL pour le rôle `keycloak-role` ;
3. extrait le nom d'utilisateur et le mot de passe à l'aide de `jq` ;
4. injecte ces informations dans les variables d'environnement attendues par Keycloak ;
5. lance finalement Keycloak.

Le script est le suivant :

```
#!/bin/bash

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

echo ">>> Demande de credentials DB à Vault..."
SECRET_JSON=$(vault read -format=json database/creds/keycloak-role)

DB_USER=$(echo $SECRET_JSON | jq -r .data.username)
DB_PASS=$(echo $SECRET_JSON | jq -r .data.password)

echo ">>> Credentials obtenus ! User: $DB_USER"

export KC_DB=postgres
export KC_DB_URL=jdbc:postgresql://localhost:5432/keycloak
export KC_DB_USERNAME=$DB_USER
```

```
export KC_DB_PASSWORD=$DB_PASS

export KC_BOOTSTRAP_ADMIN_USERNAME=admin
export KC_BOOTSTRAP_ADMIN_PASSWORD=admin

exec /opt/keycloak/bin/kc.sh start \
    --proxy=edge \
    --hostname=portal.securecorp.local \
    --http-enabled=true
```

Dans un contexte de production, l'authentification à Vault utiliserait un mécanisme plus robuste (par exemple *AppRole* ou un backend d'authentification Kubernetes), et les identifiants d'administration de Keycloak ne seraient pas fixés dans le script. Pour ce laboratoire, ces simplifications sont assumées et documentées.

La figure 1.14 montre la création et l'exécution du script `start_with_vault.sh`.

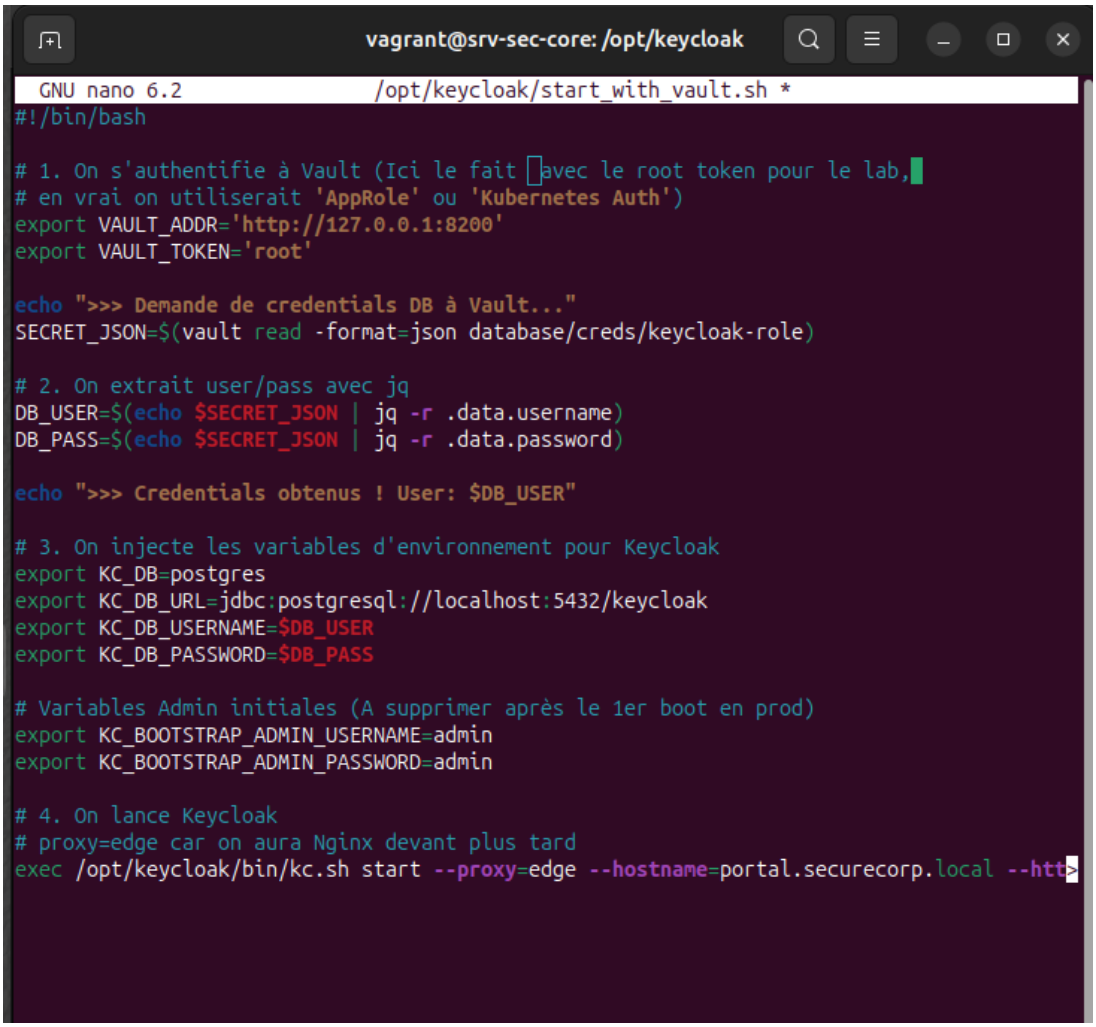

Figure 1.14: Script de démarrage de Keycloak intégrant une récupération des identifiants DB depuis Vault.

1.4.5 Intégration dans `systemd` : service Keycloak

Dans la continuité de la Phase 1, Keycloak est également géré comme un service `systemd`. Un fichier d'unité `/etc/systemd/system/keycloak.service` est créé :

```
[Unit]
Description=Keycloak IAM with Vault Secrets
After=network.target vault.service postgresql.service
Requires=vault.service postgresql.service

[Service]
User=root
ExecStart=/opt/keycloak/start_with_vault.sh
Restart=always

[Install]
WantedBy=multi-user.target
```

Les directives `After` et `Requires` garantissent que Vault et PostgreSQL sont démarrés avant Keycloak. Le service est ensuite rechargé, activé et démarré :

```
sudo systemctl daemon-reload
sudo systemctl enable keycloak
sudo systemctl start keycloak
```

La commande suivante permet de suivre les logs en temps réel :

```
journalctl -u keycloak -f
```

Après un certain temps (le premier démarrage initialisant le schéma de base de données), Keycloak indique qu'il écoute sur le port HTTP 8080. À ce stade, le service est fonctionnel du point de vue système, mais plusieurs incidents applicatifs sont apparus lors des premiers tests, comme décrit dans la section suivante.

1.4.6 Incident : erreur interne Keycloak et diagnostic

Lors du premier test d'accès via le navigateur, en passant par la passerelle et le reverse proxy, l'URL exposée renvoie une page d'erreur rouge Internal Server Error de Keycloak (et non une erreur Nginx de type 502 Bad Gateway).

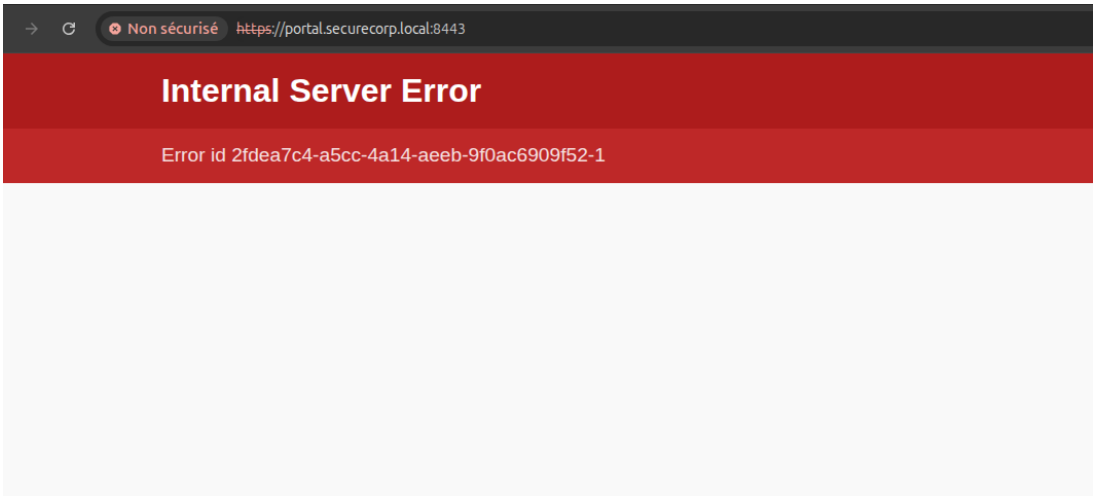

Figure 1.15: Erreur interne keycloak.

Cette observation est importante car elle montre que :

- la résolution DNS et le routage fonctionnent (le navigateur atteint bien la VM Gateway) ;
- la couche TLS fonctionne (la connexion HTTPS est établie, malgré l'avertissement attendu sur le certificat interne) ;
- le reverse proxy transmet correctement la requête jusqu'à Keycloak.

L'erreur ne provient donc pas du réseau, mais bien de Keycloak lui-même. Le réflexe adopté a été d'analyser les journaux du service :

```
vagrant ssh sec-core
journalctl -u keycloak -n 200 -e
```

Dans ces logs, une erreur explicite apparaît côté PostgreSQL :

```
org.postgresql.util.PSQLException:
  ERROR: permission denied for table realm_attribute
```

Cette erreur met en évidence un problème subtil lié à l'utilisation de secrets dynamiques pour la base de données .

Au premier démarrage, Vault a généré un premier utilisateur PostgreSQL (par exemple `v-root-keycloak-role-...`) pour Keycloak, qui a ensuite créé les tables et en est devenu propriétaire. Après un redémarrage, ce premier utilisateur a expiré ou a été abandonné. Vault a alors généré un nouvel utilisateur (par exemple `v-root-keycloak-role-...2`), avec accès à la base `keycloak`, mais sans être propriétaire des tables existantes. Lorsque Keycloak, exécuté sous ce nouvel utilisateur, tente de lire ou modifier la table `realm_attribute`, PostgreSQL refuse l'accès, d'où le permission denied.

Ce comportement illustre une limite classique des secrets dynamiques si les droits sur les objets (tables, séquences) ne sont pas correctement anticipés.

1.4.6.1 Ajustement des droits via Vault

Dans un premier temps, une réinitialisation complète de la base `keycloak` a été envisagée via :

```
sudo -u postgres psql -c "DROP DATABASE keycloak;"
```

Cependant, PostgreSQL a refusé l'opération car plusieurs services utilisaient déjà cette base (connexions actives). Plutôt que de forcer le drop, l'approche retenue a été de corriger la recette SQL utilisée par Vault lors de la création des comptes dynamiques.

Le rôle `keycloak-role` de Vault a donc été mis à jour pour inclure explicitement des droits sur l'ensemble des tables et séquences du schéma `public` :

```
vault write database/roles/keycloak-role \
    db_name=postgresql \
    creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}';
                         GRANT ALL PRIVILEGES ON DATABASE keycloak TO \"{{name}}\";
                         GRANT ALL ON SCHEMA public TO \"{{name}}\";
                         GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO \"{{na
                         GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO \"
    default_ttl="1h" \
    max_ttl="24h"
```

Après cette modification, le service Keycloak a été redémarré :

```
sudo systemctl restart keycloak
```

Les nouvelles instances d'utilisateurs dynamiques générées par Vault disposent ainsi de droits complets sur les objets de la base `keycloak`, ce qui corrige l'erreur permission denied for table realm_attribute.

1.4.7 Accès local requis et mise en place d'un tunnel SSH

Une fois l'erreur de droits corrigée, le message renvoyé par Keycloak lors de l'accès à l'interface d'administration a évolué : au lieu d'une erreur interne, Keycloak affichait la page Local access required. Ce comportement correspond à la politique de sécurité par défaut de Keycloak : tant qu'aucun compte administrateur n'a été créé, l'interface de création de l'utilisateur admin n'est accessible qu'en local (`http://localhost:8080/`) et non via un reverse proxy distant.

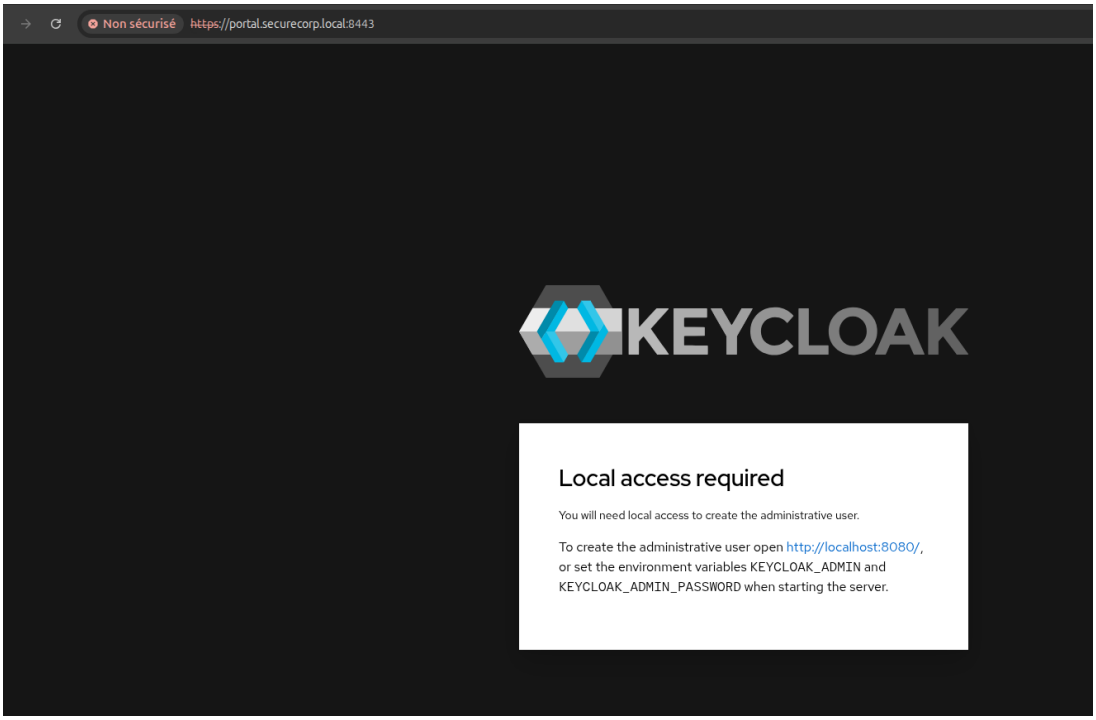

Figure 1.16: Premier accès à keycloak

Pour respecter cette contrainte tout en travaillant depuis le poste de développement, un tunnel SSH a été mis en place depuis la machine hôte vers la VM `srv-sec-core` :

```
vagrant ssh sec-core -- -L 8888:localhost:8080
```

Cette commande établit un tunnel chiffré entre le port 8888 de la machine locale et le port 8080 de la VM, de sorte que l'URL suivante dans le navigateur du poste ressemble, du point de vue de Keycloak, à un accès local :

```
http://localhost:8888
```

En utilisant ce tunnel, l'interface de création du premier utilisateur administrateur apparaît correctement.

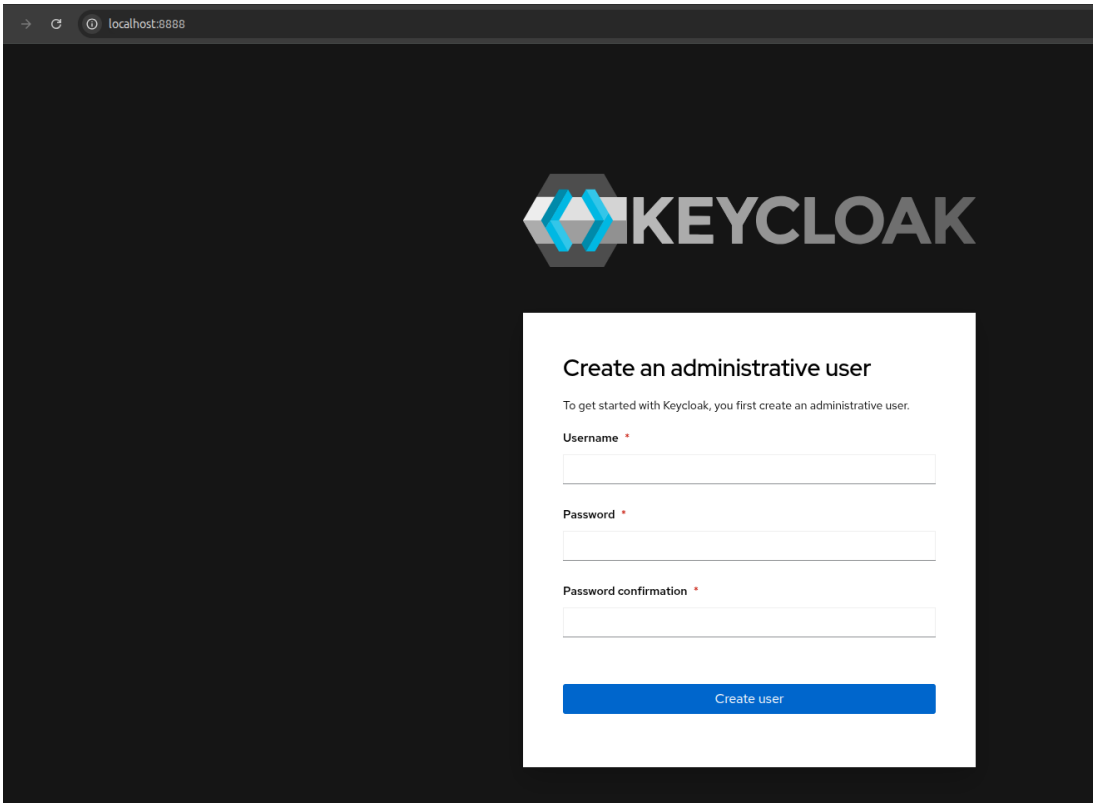

Figure 1.17: Contournement accès formulaire login keycloak

Un compte `admin` est alors créé via le formulaire (nom d'utilisateur et mot de passe saisis et confirmés), mais une nouvelle erreur générique An internal server error occurred est affichée après validation.

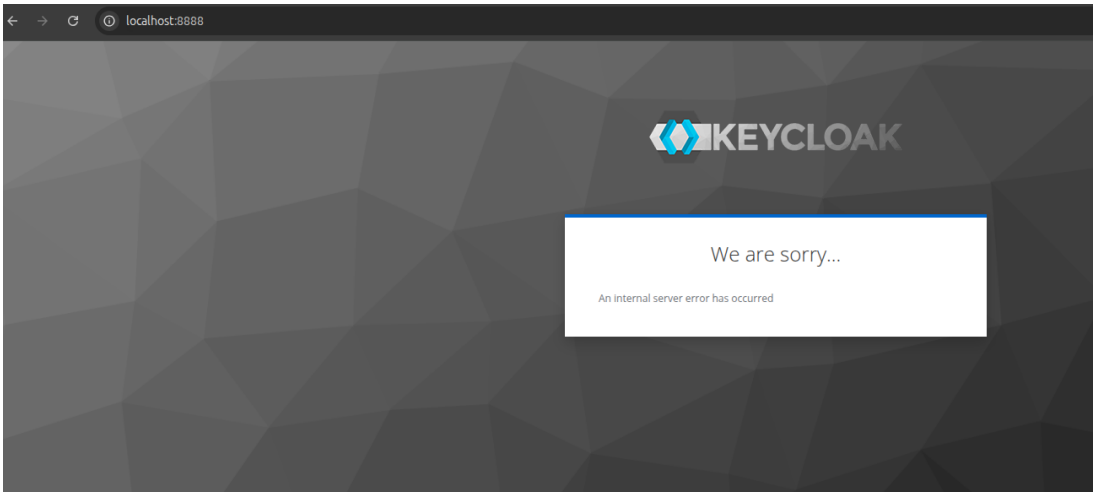

Figure 1.18: Impossibilité de creer l'utilisateur admin

Ce symptôme indique un dysfonctionnement applicatif subsistant côté Keycloak (par exemple une exception lors de l'enregistrement du compte), qui nécessite une analyse complémentaire des logs applicatifs et sera documenté dans un post-mortem ultérieur.

1.4.8 Vérification côté PostgreSQL : création automatique des rôles

Malgré ces erreurs applicatives, la vérification côté PostgreSQL montre que l'intégration avec Vault fonctionne comme prévu pour la création des utilisateurs dynamiques. Une commande permet de lister les rôles existants :

```
sudo -u postgres psql -c "\du"
```

La liste des rôles affiche des utilisateurs supplémentaires au nom généré, de type `v-root-keycloak-role-xxxxx`, correspondant aux comptes éphémères créés par Vault lors des différents démarrages de Keycloak. Cela confirme que la partie gestion dynamique des identifiants est opérationnelle, même si la création du premier administrateur côté Keycloak reste à stabiliser.

1.4.9 Bilan de la phase 2

À l'issue de cette phase, l'Identity Provider Keycloak est installé, intégré à Vault et géré comme un service `systemd` sur `srv-sec-core`. Cette étape a été l'occasion de rencontrer et de traiter plusieurs incidents réalistes :

- une erreur d'accès à une table PostgreSQL liée au changement d'utilisateur dynamique généré par Vault ;
- la nécessité d'enrichir les droits SQL (tables et séquences) dans le rôle `keycloak-role` de Vault ;
- la découverte et le contournement de la contrainte Local access required via un tunnel SSH ;
- une erreur interne lors de la création du premier compte administrateur, qui ouvre la voie à une analyse plus fine des logs applicatifs.

Même si l'interface d'administration de Keycloak n'est pas encore totalement stable à ce stade, l'intégration Vault → PostgreSQL → Keycloak est fonctionnelle et illustre concrètement une approche Zero Trust, dans laquelle les composants ne détiennent que des secrets temporaires, délivrés à la demande par un gestionnaire central. L'incident restant sera formalisé et analysé dans un *post-mortem* dédié.

Les phases suivantes consisteront à exposer Keycloak via la passerelle `srv-gateway` (Nginx en reverse proxy), à sécuriser les échanges en HTTPS à l'aide de la PKI construite en Phase 1, puis à intégrer l'authentification OIDC dans le portail applicatif.

1.4.10 Résolution critique : Droits PostgreSQL et redémarrage propre

L'erreur interne (An internal server error occurred) lors de la création de l'administrateur via le tunnel SSH a révélé que la mise à jour des droits dans Vault effectuée précédemment n'était pas rétroactive sur la session de base de données active ou était insuffisante pour l'initialisation complète des tables système de Keycloak.

Pour résoudre ce blocage bloquant, une approche Table Rase (Tabula Rasa) a été appliquée pour garantir que le premier utilisateur généré par Vault dispose des droits `SUPERUSER` sur une base vierge.

1.4.10.1 Reconfiguration du Rôle Vault en Superuser

La politique de sécurité a été temporairement assouplie pour l'initialisation afin d'éviter les conflits de permission sur la création des schémas :

```
# Sur srv-sec-core
vault write database/roles/keycloak-role \
    db_name=postgresql \
    creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}' S
                         GRANT ALL PRIVILEGES ON DATABASE keycloak TO \"{{name}}\";" \
    default_ttl="1h" \
    max_ttl="24h"
```

1.4.10.2 Réinitialisation forcée de la base de données

Afin de supprimer les objets corrompus par les tentatives précédentes, la base a été détruite en forçant la déconnexion de tous les clients actifs :

```
sudo -u postgres psql -c "DROP DATABASE IF EXISTS keycloak WITH (FORCE);"
sudo -u postgres psql -c "CREATE DATABASE keycloak;"
```

Après un redémarrage du service Keycloak et la réouverture du tunnel SSH, la création du compte administrateur `admin` a réussi. L'accès à la console d'administration était alors fonctionnel.

1.5 Phase 3 : Exposition via la Gateway et résolution des erreurs 502

L'objectif suivant était d'accéder à l'application via l'URL publique `https://portal.securecorp.local:8443`, passant par le reverse proxy Nginx (VM `srv-gateway`).

1.5.1 Incident : Erreur 502 Bad Gateway persistante

Lors de la tentative d'accès à l'application via la Gateway, une erreur bloquante 502 Bad Gateway est apparue, remplaçant les erreurs précédentes.

Figure 1.19: Erreur Nginx 502 Bad Gateway

Ce code d'erreur indique que Nginx (le proxy) reçoit la requête du client, mais n'arrive pas à obtenir une réponse valide de la part du serveur en amont (l'application Python ou Keycloak).

1.5.1.1 Diagnostic de la chaîne de communication

Une série de tests de connectivité a été réalisée depuis la machine `srv-gateway` pour isoler le composant défaillant :

1. **Test vers l'App Python (Gunicorn) :**

   ```
   curl http://192.168.56.20:5000
   ```

   Résultat : Succès (Code HTML renvoyé). Cela a validé que l'application Python fonctionnait et que le pare-feu ne bloquait pas. Le problème de configuration Nginx (upstream) a été corrigé pour pointer vers le bon port (5000).

2. **Test vers Keycloak :**

   ```
   curl http://192.168.56.10:8080
   ```

   Résultat : `Connection refused`.

Ce test a révélé la cause racine majeure : bien que le service Keycloak semblait actif (`systemctl status keycloak` : running), il ne répondait pas sur l'interface réseau.

1.5.1.2 Analyse approfondie : Le problème d'écoute réseau

L'analyse des ports sur la VM `srv-sec-core` a montré l'absence d'écoute sur le port 8080. L'examen des journaux a mis en évidence deux problèmes critiques :

1. **Binding Local :** Keycloak était configuré par défaut pour n'écouter que sur `localhost`. Le script de démarrage `start_with_vault.sh` a été modifié pour forcer l'écoute sur toutes les interfaces :

   ```
   --http-host=0.0.0.0
   ```

2. **Rupture de la chaîne de confiance Vault :** Plus critique encore, les logs montraient que Keycloak redémarrait en boucle car il ne trouvait plus les identifiants de base de données :

   ```
   No value found at database/creds/keycloak-role
   ```

Ce problème était dû à l'expiration du jeton Vault ou à la désactivation du moteur de secret après les multiples redémarrages.

1.5.1.3 Restauration de l'infrastructure de secrets

Pour corriger la chaîne de confiance, l'environnement Vault a été rechargé :

1. Réactivation du moteur de secret `database` dans Vault.
2. Reconfiguration de la connexion Vault-PostgreSQL.
3. Modification de `pg_hba.conf` pour accepter l'authentification standard (contournement des erreurs SCRAM).

Une fois ces services stabilisés, la commande `netstat` a confirmé que Keycloak écoutait bien sur `0.0.0.0:8080`. L'erreur 502 a alors disparu, laissant place à la mire de connexion Keycloak.

1.6 Phase 4 : Intégration OIDC et Finalisation

1.6.1 Correction de l'URI de redirection

Après la saisie des identifiants sur la mire SSO, une erreur Invalid parameter: redirect_uri est survenue. Cela indiquait que l'URL de retour générée par l'application Python (`https://portal.../oidc_callback`) n'était pas déclarée dans la liste blanche de Keycloak.

La correction a été effectuée via l'outil CLI `kcadm.sh` en mettant à jour le client :

```
./kcadm.sh update clients/<ID_CLIENT> -r SecureCorp \
  -s "redirectUris=[\"https://portal.securecorp.local:8443/oidc_callback\"]"
```

1.6.2 Stabilisation de la session et succès final

Une dernière erreur de type `ERR_CONNECTION_REFUSED` ou une boucle de redirection persistait après l'authentification. Le diagnostic a montré que l'application Python tentait de valider le jeton (Token Exchange) en contactant Keycloak via son URL publique HTTPS, créant une boucle réseau impossible.

La configuration de l'application (`app.py`) a été ajustée pour utiliser le réseau interne ("Service Mesh" simplifié) :

```
KEYCLOAK_URL = "http://192.168.56.10:8080/realms/SecureCorp"
```

De plus, le middleware `ProxyFix` a été ajouté à Flask pour qu'il interprète correctement les en-têtes `X-Forwarded-Proto` envoyés par Nginx et accepte de générer des cookies sécurisés.

1.6.3 Résultat

Après ces ajustements, le parcours utilisateur est complet et sécurisé :

- L'utilisateur accède au portail.
- Il est redirigé vers Keycloak pour s'authentifier.

- Après succès, il est redirigé vers l'application qui valide son identité.
- La page d'accueil sécurisée s'affiche.

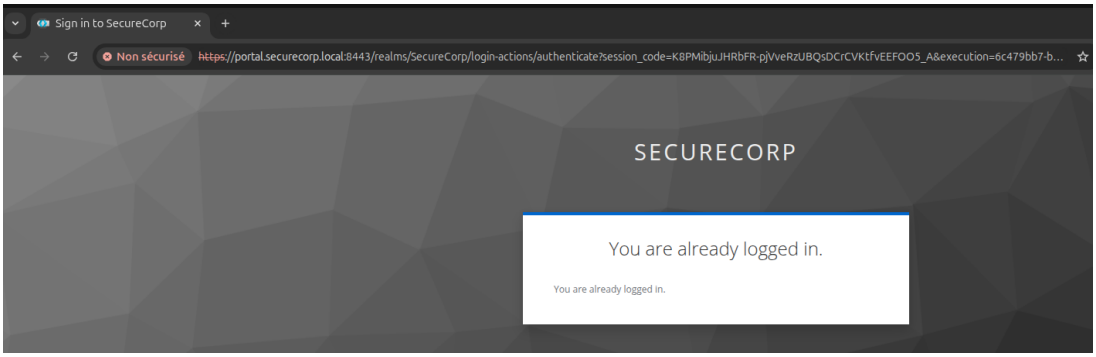

Figure 1.20: Authentification réussie et session active

L'architecture Zero Trust, incluant PKI, rotation de secrets, IAM centralisé et Reverse Proxy, est désormais pleinement opérationnelle.

1.7 Description de l'infrastructure cible

1.7.1 Architecture réseau

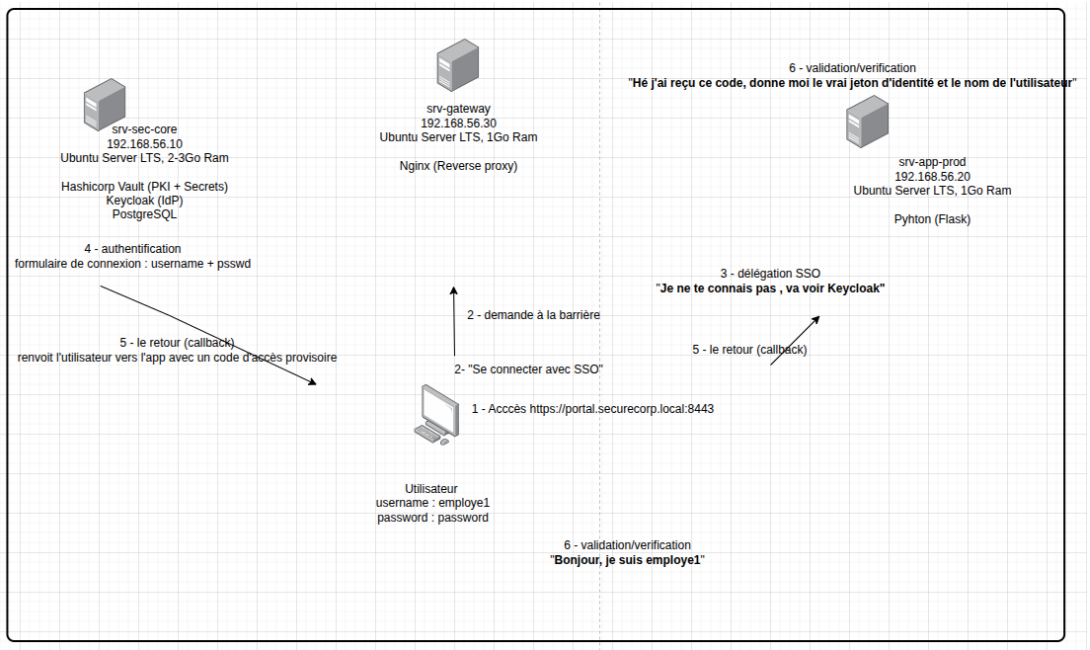

Figure 1.21: Architecture de l'infrastructure

2 Annexe - VagrantFile pour l'installation de l'infrastructure

```
 # -*- mode: ruby -*-
# vi: set ft=ruby :

# --- Configuration Globale ---
# OS : Ubuntu 22.04 LTS (Jammy Jellyfish) - Stable et standard entreprise
BOX_IMAGE = "bento/ubuntu-22.04"

# --- Script de Provisioning (s'excute au premier dmarrage) ---
# Ce script installe Python (pour Ansible) et configure la rsolution de noms (DNS
    simul)
$script_common = <<-SCRIPT
  echo ">>> Mise  jour des  dpts  et installation des outils de base..."
  apt-get update -q
  apt-get install -y python3 python3-pip curl vim git net-tools

  echo ">>> Configuration du DNS local (/etc/hosts)..."
  # On supprime les anciennes entres pour viter les doublons si on reprovisionne
  sed -i '/192.168.56./d' /etc/hosts

  # On ajoute la cartographie complte du rseau
  cat >> /etc/hosts <<EOF
192.168.56.10  srv-sec-core.securecorp.local  srv-sec-core
192.168.56.20  srv-app-prod.securecorp.local  srv-app-prod
192.168.56.30  srv-gateway.securecorp.local   srv-gateway
EOF
  echo ">>> Configuration rseau termine."
SCRIPT

Vagrant.configure("2") do |config|
  config.vm.box = BOX_IMAGE

  # ================================================================
  # VM 1: Security Core (Vault, Keycloak, DB)
  # ================================================================
  config.vm.define "sec-core" do |sec|
    sec.vm.hostname = "srv-sec-core"
    # IP Statique sur rseau priv (Host-only)
    sec.vm.network "private_network", ip: "192.168.56.10"

    sec.vm.provider "virtualbox" do |vb|
      vb.name = "SecureCorp-Sec-Core"
      # 3 Go de RAM ncessaire pour Java (Keycloak) + Vault
```

```
      vb.memory = 3072
      vb.cpus = 2
    end

    sec.vm.provision "shell", inline: $script_common
  end

  # ============================================================
  # VM 2: App Production (Python Backend)
  # ============================================================
  config.vm.define "app-prod" do |app|
    app.vm.hostname = "srv-app-prod"
    app.vm.network "private_network", ip: "192.168.56.20"

    app.vm.provider "virtualbox" do |vb|
      vb.name = "SecureCorp-App-Prod"
      # 1 Go suffit pour un script Python lger
      vb.memory = 1024
      vb.cpus = 1
    end

    app.vm.provision "shell", inline: $script_common
  end

  # ============================================================
  # VM 3: Gateway (Nginx Reverse Proxy)
  # ============================================================
  config.vm.define "gateway" do |gw|
    gw.vm.hostname = "srv-gateway"
    gw.vm.network "private_network", ip: "192.168.56.30"

    # Port Forwarding : Pour accder au site depuis ton vrai navigateur
    # Tu taperas http://localhost:8080 pour atteindre le port 80 de la gateway
    gw.vm.network "forwarded_port", guest: 80, host: 8080
    gw.vm.network "forwarded_port", guest: 443, host: 8443

    gw.vm.provider "virtualbox" do |vb|
      vb.name = "SecureCorp-Gateway"
      # Nginx est trs  lger
      vb.memory = 1024
      vb.cpus = 1
    end

    gw.vm.provision "shell", inline: $script_common
  end

end
```

En tête du fichier, une constante `BOX_IMAGE` définit l'image système utilisée pour toutes les machines virtuelles :

```
BOX_IMAGE = "bento/ubuntu-22.04"
```

Le choix d'Ubuntu 22.04 LTS (Jammy Jellyfish) permet de s'appuyer sur une distribution stable, largement utilisée en entreprise et bien supportée par les différents outils (Ansible, Vault, Keycloak, etc.).

La directive `config.vm.box = BOX_IMAGE` applique cette image par défaut à toutes les VMs définies dans le fichier, ce qui garantit l'homogénéité de l'environnement.

2.0.0.1 Script de provisioning commun

Le bloc `$script_common` définit un script shell qui sera exécuté sur chaque machine lors du premier démarrage :

- Mise à jour des dépôts APT et installation des outils de base : `python3`, `curl`, `git`, `net-tools`, etc. L'installation de Python est nécessaire pour permettre ensuite l'utilisation d'Ansible comme outil de configuration automatisée.
- Configuration d'un "DNS local" en modifiant le fichier `/etc/hosts` :
  - Suppression préalable des anciennes entrées correspondant au réseau privé (`192.168.56.x`) pour éviter les doublons en cas de reprovisionnement.
  - Ajout des correspondances IP → noms d'hôtes pour les trois machines :
    - `192.168.56.10 srv-sec-core.securecorp.local srv-sec-core`
    - `192.168.56.20 srv-app-prod.securecorp.local srv-app-prod`
    - `192.168.56.30 srv-gateway.securecorp.local srv-gateway`

Grâce à cette configuration, chaque machine peut joindre les autres par leur nom (par exemple `ping srv-gateway`), ce qui est indispensable pour la suite du projet (certificats TLS basés sur des noms DNS, configuration de reverse proxy, etc.).

2.0.0.2 Définition des machines virtuelles

Le bloc principal `Vagrant.configure("2") do |config|` contient la définition des trois machines virtuelles du laboratoire.

VM `sec-core` : coeur sécurité La première machine, définie avec `config.vm.define "sec-core"`, porte le nom d'hôte `srv-sec-core` et reçoit l'adresse IP privée `192.168.56.10`. Ce serveur a vocation à héberger les composants cœur de la sécurité (par exemple Vault, la base IAM, ou la base de données selon l'architecture retenue).

Les ressources VirtualBox allouées sont de 3 Go de RAM et 2 vCPU, ce qui est adapté à l'exécution de services relativement lourds comme Keycloak ou Vault :

- `vb.name = "SecureCorp-Sec-Core"` : nom de la VM dans VirtualBox.
- `vb.memory = 3072`, `vb.cpus = 2` : ressources matérielles.

Enfin, la directive `sec.vm.provision "shell", inline: $script_common` applique le script de provisioning commun décrit précédemment (installation des outils, configuration de `/etc/hosts`).

VM `app-prod` : serveur d'application La seconde machine, `config.vm.define "app-prod"`, représente le serveur d'application de production (`srv-app-prod`) et est adressée en `192.168.56.20`. Elle hébergera par exemple l'API ou le portail web interne.

Les ressources sont plus modestes (1 Go de RAM, 1 vCPU), suffisantes pour une application Python légère ou un service backend de démonstration. Elle bénéficie également du script de provisioning commun.

VM `gateway` : passerelle et reverse proxy La troisième machine, `config.vm.define "gateway"`, correspond à la passerelle d'entrée (`srv-gateway`) avec l'adresse IP `192.168.56.30`. Elle doit héberger un reverse proxy (par exemple Nginx) exposant l'architecture interne aux clients.

Deux règles de redirection de ports (*port forwarding*) sont définies :

- `gw.vm.network "forwarded_port", guest: 80, host: 8080` : le port 80 de la machine virtuelle (HTTP) est accessible depuis la machine hôte via le port 8080.
- `gw.vm.network "forwarded_port", guest: 443, host: 8443` : le port 443 (HTTPS) est accessible depuis la machine hôte via le port 8443.

Cela permet de tester l'accès au portail sécurisé depuis le navigateur de la machine hôte en utilisant des URLs du type :

```
http://localhost:8080
https://localhost:8443
```