Hashicorp Vault

Overview of Vault components and their roles

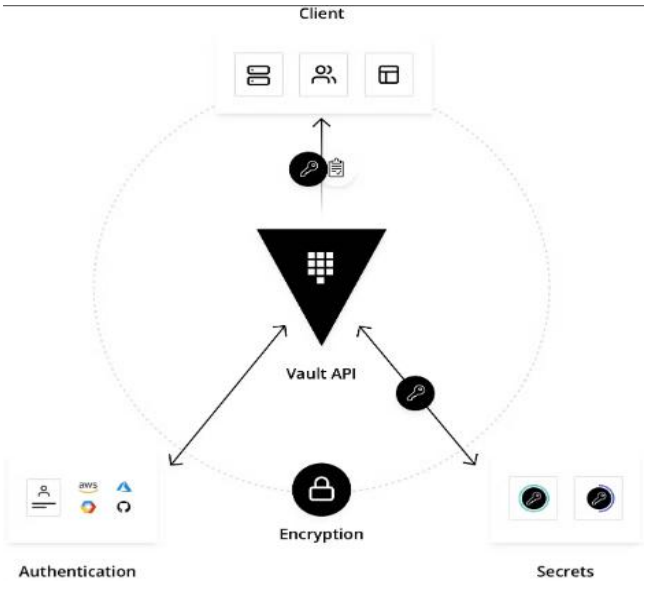
Hashicorp vault is a fully managed implementation of Vault. Hashicorp operates the infrastructure, allowing organizations to get up and running quickly. HCP Vault provides a consistent user experience compared to a self-managed Vault cluster.

Its components:

1. **Authenticate**: The client proves their identity to Vault, which then generates a token tied to a specific policy.
2. **Validation**: Vault checks the client's identity against trusted sources like GitHub, LDAP, or AppRole.
3. **Authorize**: Vault compares the client's token against security policies to determine what actions the client can perform.
4. **Access**: Vault allows the client to access secrets, keys, and encryption services using the token. The client uses this token for future operations.

Understanding architecture

Vault primarily uses tokens, each linked to a client's policy. These policies are path-based and define the actions and access each client has to specific paths. You can manually create tokens and assign them to clients, or clients can log in to obtain a token themselves.



Benefits

1. **Secure Secret Storage**:
    - Stores key/value secrets securely.
    - Encrypts secrets before saving them, so raw storage access doesn't reveal secrets.
    - Supports various storage backends like disk and Consul.
2. **Dynamic Secrets**:
    - Generates secrets on-demand for systems like AWS or SQL databases.
    - Provides temporary credentials, which are automatically revoked after their lease expires.
3. **Data Encryption**:
    - Encrypts and decrypts data without storing it.
    - Enables secure storage of encrypted data in external locations (e.g., SQL databases) without custom encryption solutions.
4. **Leasing and Renewal**:
    - Each secret has a lease, which Vault manages.
    - Automatically revokes secrets after the lease ends.
    - Allows clients to renew leases through APIs.
5. **Revocation**:
    - Supports revoking individual or multiple secrets.
    - Can revoke secrets based on user or type, useful for security incidents and key rotation.

Vault Installation:

Here are the CLI(Command Line Interface) command for installing vault

Step 1 - Add PGP for the package signing key.

```
sudo apt update && sudo apt install gpg
```

Step 2 - Add the HashiCorp GPG key.

```
wget -O https://apt.releases.hashicorp.com/gpg | gpg --dearmor | sudo tee
/usr/share/keyrings/hashicorp-archive-keyring.gpg
```

Step 3 - Verify the key's fingerprint.

```
gpg --no-default-keyring --keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg
--fingerprint
```

4 - Add the official HashiCorp Linux repository.

```
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee
/etc/apt/sources.list.d/hashicorp.list
```

Step 5 - Update and install.

```
sudo apt update && sudo apt install vault
```

Start and Stop in Development Mode:

Step 1 : CLI Command for starting the server –

```
vault server -dev
```

Step 2 : Set VAULT_ADDR by exporting to environment variable –

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

Step 3 : Set Root Token by exporting to environment variable –

```
export VAULT_TOKEN="hvs.6j4cuewowBGit65rheNoceI7"
```

Step 4 : Verify the status of vault server by running the command –

```
$ vault status
```

Read, Write, and Delete Secrets:

1. READ SECRET-

   $ vault kv get my/path

2. WRITE SECRET -

   $ vault kv put my/path my-key-1=vaule-1

3. DELETE SECRET -

   $ vault kv delete my/path

4. READ the secrets in JSON format -

   $ vault kv get -format=json my/path

5. Enable secret path in Hashi corp -

   $ vault secrets enable -path=my kv

Secret Engine and Path:

1. Enable custom secret engine

   $ vault secrets enable -path=my-custom-secret-engine-1 kv

2. List all the secret engine available in vault

   $ vault secrets list -detailed

3. Disable secret engine

   $ vault secrets disable my-custom-secret-engine-1

Dynamic Secrets Generation:

1. Enable the secret engine path for AWS

   $ vault secrets enable -path=aws aws

2. View the secret list

   $ vault secrets list

3. Write AWS root config inside your hashicorp vault

```
$ vault write aws/config/root \ access_key=YOUR_ACCESS_KEY \
secret_key=YOUR_SECRET_KEY \ region=eu-north-1
```

4. Setup role

```
$ vault write aws/roles/my-ec2-role \

credential_type=iam_user

\ policy_document=-EOF

{ "Version": "2012-10-17",

"Statement": [

{

 "Sid": "Stmt1426528957000",

"Effect": "Allow",

 "Action": [ "ec2:*" ],

 "Resource": [ "*" ]

 }

 ]

 }

EOF
```

5. Generate access key and secret key for that role

```
$ vault read aws/creds/my-ec2-role
```

6. Revoke the secrets if you do not want it any longer

```
$ vault lease revoke aws/creds/my-ec2-role/J8WHZJ5NItdH23KYYHdORv3K
```

Token and GitHub Authentication:

1. List vault policies

```
$ vault policy list
```

2. Write your custom policy

$ vault policy write my-policy – EOF

(Dev servers have version 2 of KV secrets engine mounted by default, so will need these paths to grant permissions:)

path "secret/data/*"

{ capabilities = ["create", "update"]

}

path "secret/data/foo" {

capabilities = ["read"]

}

EOF

3. Read Vault policy details

   $ vault policy read my-policy

4. Delete Vault policy by policy name $ vault policy delete my-policy

   Attach token to policy $ export VAULT_TOKEN="$(vault token create -field token -policy=my-policy)"

5. Associate auth method with policy

   $ vault write auth/approle/role/my-role \

   secret_id_ttl=10m \ t

   oken_num_uses=10 \

   token_ttl=20m \

   token_max_ttl=30m \

   secret_id_num_uses=40 \

   token_policies=my-policy

6. Generate and Export Role ID

   export ROLE_ID="$(vault read -field=role_id auth/approle/role/my-role/role-id)"

7. Generate and Export Secret ID

   export SECRET_ID="$(vault write -f -field=secret_id auth/approle/role/my-role/secret-id)"

Policies:

Write and apply policies using HCL.

- HashiCorp Configuration Language (HCL) is used to define policies in Vault.
- Policies are written in a declarative syntax that specifies what actions can be performed on which paths within Vault.
- Each policy includes rules that grant or deny permissions for certain operations (e.g., read, write, delete) on specific paths.

1. Write the Policy to Vault:

   vault policy write example-policy example-policy.hcl

```
path "secret/data/*" {

capabilities = ["create", "read", "update", "delete", "list"]

}

path "secret/data/finance/*" {

capabilities = ["read", "list"]

 }

path "auth/token/lookup-self" {

 capabilities = ["read"]

 }
```

2. Apply the Policy to a Token:

   vault token create -policy="example-policy"

Control access to secrets and paths through policies.

- Policies control access to secrets and paths in Vault by defining permissions for clients.
- When a client authenticates, they receive a token linked to one or more policies.
- The policy associated with the token dictates what the client can and cannot do within Vault.

- This fine-grained access control ensures that clients can only access the secrets and perform actions they are explicitly allowed to.Use the Token:

- The client can now authenticate with the generated token and perform actions as per the policy.

```
# Write a secret to the path "secret/data/example"
VAULT_TOKEN=<generated_token> vault kv put secret/data/example
username="user1" password="pass123"

# Read the secret from the path "secret/data/example"
VAULT_TOKEN=<generated_token> vault kv get secret/data/example

# List secrets under the path "secret/data/"
VAULT_TOKEN=<generated_token> vault kv list secret/data/

# Attempt to write a secret to the path "secret/data/finance"
# This will fail because the policy only allows read and list operations for this path
VAULT_TOKEN=<generated_token> vault kv put secret/data/finance
account="finance123
```

Part 8: Deploy Vault, HTTP API & UI:

1. Configure and start Vault in server mode for production.

Create a configuration file (vault.hcl). Example configuration for a production setup:

```
storage "aws" {

 bucket = "my-vault-bucket"

 region = "us-west-2"

 access_key = "YOUR_AWS_ACCESS_KEY"

 secret_key = "YOUR_AWS_SECRET_KEY"

}

listener "tcp" {

 address = "0.0.0.0:8200"

 tls_disable = 0

 tls_cert_file = "/path/to/tls_cert.pem"

 tls_key_file = "/path/to/tls_key.pem"}
```

```
seal "transit" {
  address = "https://transit-seal.example.com:8200"
  token = "YOUR_TRANSIT_SEAL_TOKEN"
  key_name = "my-key"
}
ui = true
```

Start Vault:

Run the following command to start Vault with your configuration file:

```
vault server -config=/path/to/vault.hcl
```

Initialize and Unseal Vault:

Open a new terminal and initialize Vault:

```
vault operator init
```

This command will output unseal keys and a root token. Store these securely.

Unseal Vault using the unseal keys:

```
vault operator unseal <unseal-key-1>
vault operator unseal <unseal-key-2>
vault operator unseal <unseal-key-3>
```

Authenticate and Set Up Vault:

Authenticate with Vault using the root token:

```
export VAULT_TOKEN=<your-root-token>
```

2. Interacting with Vault Using HTTP API and UI

HTTP API:

Authenticate:

Obtain a client token (if using a different method than root token):

```
curl --request POST \
 --data '{"password": "my-password"}' \
 http://127.0.0.1:8200/v1/auth/userpass/login/my-username
```

Write a Secret:

Store a secret in Vault:

```
curl --request POST \
 --data '{"value": "my-secret-value"}' \
 http://127.0.0.1:8200/v1/secret/my-secret
```

Read a Secret:

Retrieve the secret from Vault:

- curl http://127.0.0.1:8200/v1/secret/my-secret

UI:

Access Vault UI:

- Open your browser and navigate to http://<vault-server-ip>:8200.

Log In:

- Use the root token or another method to log in.

Manage Secrets:

- Navigate to the "Secrets" section to add or manage secrets via the web interface.

Example

To store a database password:

Using HTTP API:

```
curl --request POST \

 --data '{"value": "db-password-123"}' \

 http://127.0.0.1:8200/v1/secret/database_password
```

Using UI:

- Log in to the Vault UI.
- Go to the "Secrets" section.
- Add a new secret at path database_password with value db-password-123.

Use Cases:

1. **General Secret Storage**:
    - Generates short-lived, just-in-time credentials to reduce security risks from leaked or outdated credentials.
2. **Static Secrets**:
    - Stores long-lived, unchanging credentials securely behind a cryptographic barrier.
    - **Resource**: Versioned Key/Value Secrets Engine tutorial.
3. **Dynamic Secrets**:
    - Dynamically generates credentials when needed and manages their lifecycle.
    - **Resource**: Dynamic Secrets: Database Secrets Engine tutorial.
4. **Data Encryption**:
    - Provides encryption as a service with centralized key management.
    - Encrypts/decrypts data in transit and storage, simplifying encryption for developers.
    - **Resource**: Encryption as a Service: Transit Secrets Engine and Advanced Data Protection tutorial series.
5. **Identity-Based Access**:
    - Unifies access control across multiple identity providers and platforms.
    - **Resource**: Identity: Entities and Groups tutorial and Policies tutorial series.
6. **Key Management**:
    - Manages encryption keys across cloud providers, maintaining centralized control in Vault.
    - **Resources**: Key Management Secrets Engine with Azure Key Vault and GCP Cloud KMS.