Vault Workshop - 101

Application Architect - CoE Team

Introduce Hashicorp Vault to the team and how to integrate with K8S and Gitlab

Agenda

1. HashiCorp Vault Overview
2. Interacting With Vault
3. Vault Secrets Engines
4. Vault Authentication Methods
5. Vault Policies
6. [Lab] Integrate Vault with Kubernetes
7. [Lab] Integrate Vault with GitLab Community version

Chapter 1

HashiCorp Vault Overview

HashiCorp Vault Overview

- HashiCorp Vault is an API-driven, cloud agnostic secrets management system.
- It allows you to safely store and manage sensitive data in hybrid cloud environments.
- You can also use Vault to generate dynamic short-lived credentials, or encrypt application data on the fly.

The Traditional Security Model

- Traditional security models were built upon the idea of perimeter based security.
- There would be a firewall, and inside that firewall it was assumed one was safe.
- Resources such as databases were mostly static. As such rules were based upon IP address, credentials were baked into source code or kept in a static file on disk.

The Traditional Security Model

Also known as the "Castle and Moat" method.

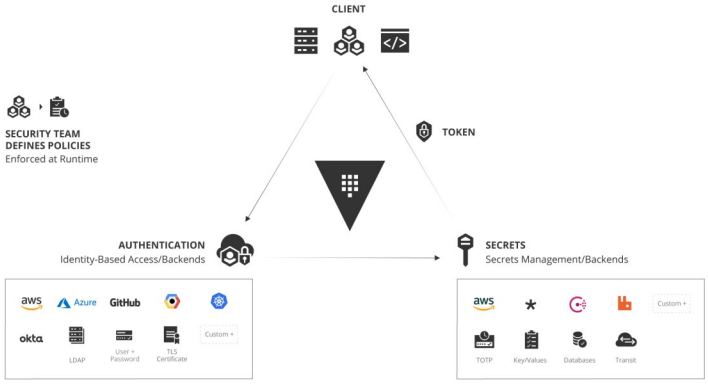
Problems with the Traditional Security Model

- IP Address based rules
- Hardcoded credentials with problems such as:
  - Shared service accounts for apps and users
  - Difficult to rotate, decommission, and determine who has access
  - Revoking compromised credentials could break

Modern Secrets Management

No well defined perimeter; security enforced by identity.

Identity Based Security

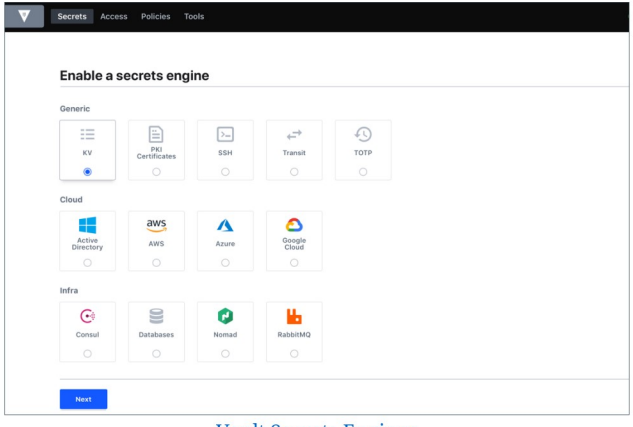


Identity Based Security

Vault was designed to address the security needs of modern applications. It differs from the traditional approach by using:

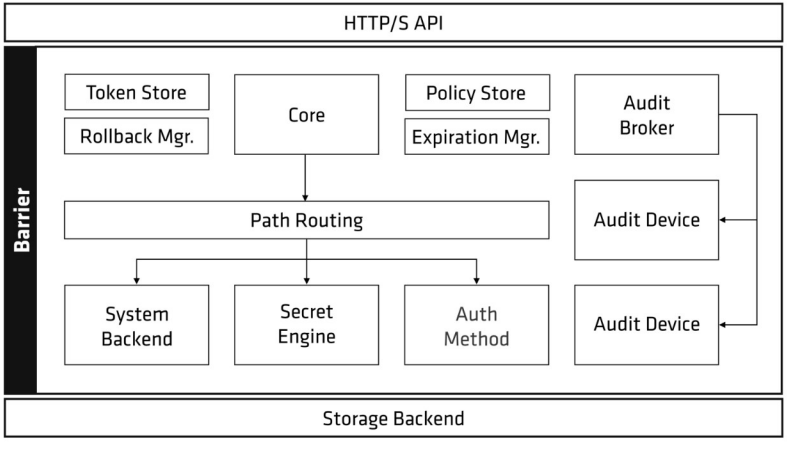
- Identity based rules allowing security to stretch across network perimeters
- Dynamic, short lived credentials that are rotated frequently
- Individual accounts to maintain provenance (tie action back to entity)
- Credentials and Entities that can easily be invalidated

Vault Secrets Engines



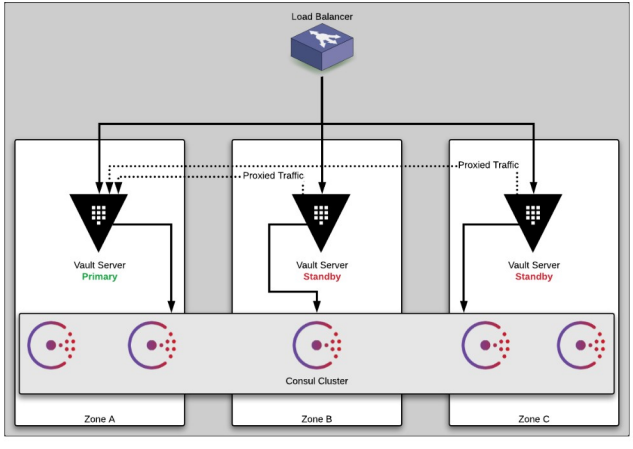
Vault Secrets Engines

Vault Architecture Internals



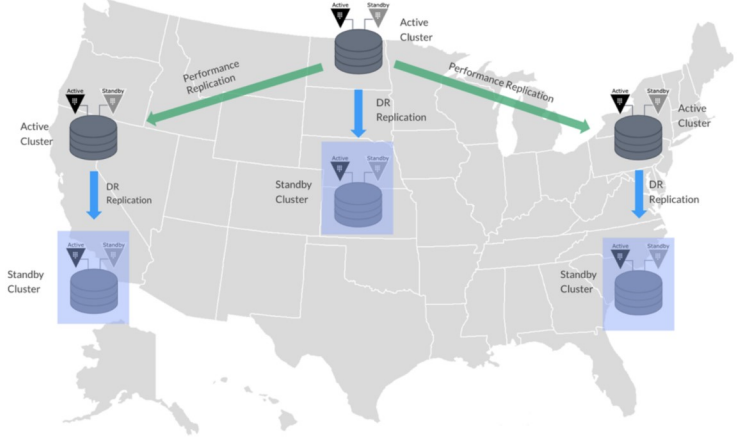
HashiCorp Vault Internals Architecture

Vault Architecture - High Availability



Vault High Availability

Vault Architecture - Multi-Region [Enterprise Only]



Vault Enterprise Replication

Chapter 1 Review

- What is HashiCorp Vault?

Chapter 1 Review

- What is HashiCorp Vault?
  - Vault is a Secrets Management System.
  - It is API-driven and cloud agnostic.
  - It can be used in untrusted networks.
  - It can authenticate users and applications against many systems.
  - It supports dynamic generation of short-lived secrets.
  - It runs in highly available clusters that can be replicated across regions.

Chapter 2

Interacting With Vault

Interacting With Vault

Vault provides several mechanisms for interacting with it:

- The Vault CLI
- The Vault UI
- The Vault API

The Vault CLI

- The Vault CLI is a Go application.
- It runs on macOS, Windows, Linux, and other operating systems.
- You can download the latest version here.

Some Basic Vault CLI Commands

- `vault` by itself will give you a list of many Vault CLI commands.
  - The list starts with the most common ones.
- `vault version` tells you the version of Vault you are running.
- `vault read` is used to read secrets from Vault.
- `vault write` is used to write secrets to Vault.

The `-h`, `-help`, and `--help` flags can be added to get help for any Vault CLI command.

Vault Server Modes

Vault servers can be run in two different modes:

- "Dev" mode that is only intended for development
- "Prod" mode that can be used in QA and production

Vault's "Dev" Mode

- It is not secure.
- It stores everything in memory.
- Vault is automatically unsealed.
- The root token can be specified before launching.

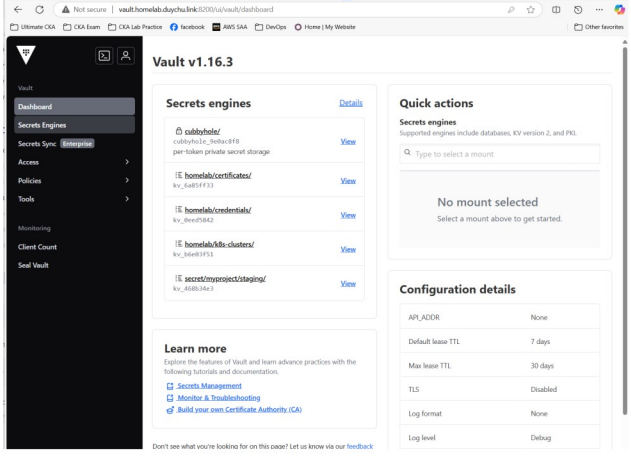
Please never store actual secrets on a server run in "Dev" mode.

The Vault UI

- In order to use the Vault UI, you must sign in.
- Vault supports multiple authentication methods.
- A new Vault server will only have the Token auth method enabled.
- In the challenge you will soon complete, you use the Token auth method and specify "root" as the token.

Signing into the Vault UI

The "Welcome to Vault" Tour



The Vault API

- Vault has an HTTP API that you can use to configure Vault and manage your secrets.
- You can check Vault's health with a simple `curl` command followed by `jq` to format the JSON output.

```
curl http://localhost:8200/v1/sys/health | jq
```


```
{
    "initialized": true,
    "sealed": false,
    "standby": false,
    "performance_standby": false,
    "replication_performance_mode": "disabled",
    "replication_dr_mode": "disabled",
    "server_time_utc": 1736588172,
    "version": "1.17.3",
    "enterprise": false,
    "cluster_name": "vault-cluster-bd2470be",
    "cluster_id": "027cf597-bd0a-731a-a57e-dcd06eb390e4",
    "echo_duration_ms": 0,
    "clock_skew_ms": 0,
    "replication_primary_canary_age_ms": 0
}
```

Authenticating Against the Vault API

- The sys/health endpoint didn't require any authentication.
- But most Vault API calls do require authentication.
- This is done with a Vault token that is provided with the `X-Vault-Token` header.

Bootup with vault with docker-compose

- In production, docker-compose is not used, however, we can still use it to understand what need to do if vault is setup for production use
- We have to unseal vault when vault server start, it is done by vault-init script

```
services:
  vault:
    image: "hashicorp/vault:${VAULT_TAG:-latest}"
    container_name: vault
    restart: always
    ports:
      - "8200:8200"
    volumes:
      - ./vault/config:/vault/config
      - ./vault/data:/vault/data
    command: ["vault", "server", "-config=/vault/config/vault-config.json"]
    cap_add:
      - IPC_LOCK
```

- We have to unseal vault when vault server start, it is done by vault-init script

```
vault-init:
  image: "hashicorp/vault:${VAULT_TAG:-latest}"
  container_name: vault-init
  restart: always
  command:
    # - tail
    # - -f
    # - /dev/null
    - "sh"
    - "-c"
    - "/vault/scripts/vault-init.sh"
  environment:
    VAULT_ADDR: http://vault:8200
  volumes:
    - ./vault/scripts/vault-init.sh:/vault/scripts/vault-init.sh
    - ./vault/keys:/vault/keys
```

- `vault/scripts/vault-init.sh` script will do 2 thing, generate root key to init vault in 1st setup, and unseal the vault if vault is setup but just be restarted

```
#! /bin/sh
set -ex
apk update
apk add jq

INIT_FILE=/vault/keys/vault.init
if [[ -f "${INIT_FILE}" ]]; then
  echo "${INIT_FILE} exists. Vault already initialized."
else
  echo "Initializing Vault..."
  sleep 20 # fixme: need to check tcp port by nc
  vault operator init -key-shares=3 -key-threshold=2 | tee ${INIT_FILE} > /dev/null ### 3 fragments, 2 are required
  ### Store unseal keys to files
  COUNTER=1
  cat ${INIT_FILE} | grep '^Unseal' | awk '{print $4}' | for key in $(cat -); do
    echo "${key}" > /vault/kevs/kev-${COUNTER}
```

- We have to unseal vault when vault server start, it is done by vault-init script. In real, this key should be kept by different person(s) who are in charge of vault

```
if [ ! -s /vault/root/token -o ! -s /vault/keys/key-1 -o ! -s /vault/keys/key-2 ] ; then
  echo "Vault is initialized, but unseal keys or token are mssing"
  return
fi

echo "Unsealing Vault"
export VAULT_TOKEN=$(cat /vault/root/token)
vault operator unseal "$(cat /vault/keys/key-1)"
vault operator unseal "$(cat /vault/keys/key-2)"

vault status
```

Chapter 2 Review

- How can you interact with Vault?
- What options can you use to get help for Vault commands?
- What are the two Vault server modes?

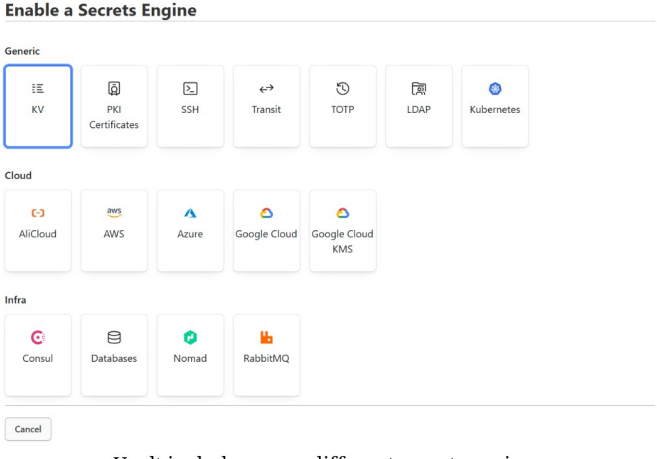
Chapter 2 Review

- How can you interact with Vault?
  - The Vault CLI
  - The Vault UI
  - The Vault API
- What options can you use to get help for Vault commands?
  - `-h`, `-help`, and `--help`
- What are the two Vault server modes?
  - Dev and Prod

Chapter 3

Vault Secrets Engines

Vault Secrets Engines



Vault includes many different secrets engines.

Important Vault Secrets Engines

- Key/Value (KV)
- PKI
- SSH
- TOTP
- Databases
- AWS, Azure, and Google
- Transit

Enabling Secrets Engines

- Most Vault secrets engines need to be explicitly enabled.
- This is done with the `vault secrets enable` command.
- Each secrets engine has a default path.
- Alternate paths can be specified to enable multiple instances:
  `vault secrets enable -path=aws-east aws`
- Custom paths must be specified in CLI commands and API calls:
  `vault write aws-east/config/root`
  instead of
  `vault write aws/config/root`

Vault's KV Secrets Engine

- Vault's KV secrets engine actually has 2 versions:
  - KV v1 (without versioning)
  - KV v2 (with versioning)
- In the second lab challenge, we will use the instance of the KV v2 engine that is automatically enabled for "Dev" mode Vault servers.
- Vault does not enable any instances of the KV secrets engine for "Prod" mode servers.
- So, you'll need to enable it yourself.

KV Secrets Engine Commands

- Use this command to mount an instance of the KV v2 secrets engine on the default path `kv`:
  `vault secrets enable -version=2 kv`
- The `vault kv` commands allow you to interact with KV engines.
  - `vault kv list` lists secrets at a specified path.
  - `vault kv put` writes a secret at a specified path.
  - `vault kv get` reads a secret at a specified path.
  - `vault kv delete` deletes a secret at a specified path.
- Other `vault kv` subcommands operate on versions of KV v2 secrets.

Chapter 3 Review

- What option is added to the `vault secrets enable` command to enable multiple instances?
- What is the difference between the two versions of the KV secrets engine?
- Can an old version of a KV v2 secret be retrieved?

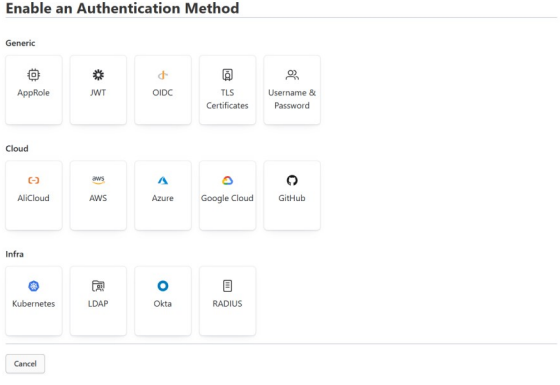
Chapter 3 Review

- What option is added to the `vault secrets enable` command to enable multiple instances?
  - Add the `-path=<path>` option and use `<path>` with the CLI and API.
- What is the difference between the two versions of the KV secrets engine?
  - KV V2 supports versioning of secrets.
- Can an old version of a KV v2 secret be retrieved?
  - Yes. You will do this in Vault UI in the challenge.

Chapter 4

Vault Authentication Methods

Vault Authentication Methods



Vault supports many different authentication methods.

Some Important Vault Auth Methods

Methods for Users

- Userpass
- GitHub
- LDAP
- JWT/OIDC
- Okta

Methods for Applications

- AppRole
- AWS
- Azure
- Google Cloud
- Kubernetes

Enabling Authentication Methods

- Most Vault auth methods need to be explicitly enabled.
- This is done with the `vault auth enable` command.
- Each auth method has a default path.
- Alternate paths can be specified to enable multiple instances:
  `vault auth enable -path=aws-east aws`
- Custom paths must be specified in CLI commands and API calls:
  `vault write aws-east/config/root`
  instead of
  `vault write aws/config/root`

Vault's Userpass Auth Method

- The Userpass method authenticates users with usernames and passwords managed by Vault.

Chapter 4 Review

- What types of entities can Vault authenticate?
- What system manages credentials for the Userpass auth method?
- Can a user that is not assigned any policies other than the default policy access any secrets?

Chapter 4 Review

- What types of entities can Vault authenticate?
  - Users and applications
- What system manages credentials for the Userpass auth method?
  - Vault
- Can a user that is not assigned any policies other than the default policy access any secrets?
  - No

Chapter 5

Vault Policies

Vault Policies

- Vault Policies restrict the secrets users and applications have access to.
- Vault follows the practice of least privilege, *denying* access by default.
- Vault administrators must explicity grant users and applications access to specific paths with policy statements.
- In addition to specifying paths, policies also specify a set of capabilities for those paths.
- Policies are written in HashiCorp Configuration Language (HCL).

A Vault Policy Example

- Here is an example of a Vault policy:

```
# Allow tokens to look up their own properties
path "auth/token/lookup-self" {
capabilities = ["read"]
}
```

- Note that this policy does not allow tokens to change their own properties.

Policy Paths and Capabilities

- The path of a policy maps to a Vault API path.
- The most common capabilities granted are: `create`, `read`, `update`, `delete`, and `list` which correspond to HTTP verbs like POST and GET.
- Two other capabilities do not correspond to HTTP verbs:
  - `sudo` allows access to paths that are root-protected.
  - `deny` denies access to a path and takes precedence over other capabilities.

Configuring Policies for LOBs

- Many organizations organize Vault secrets by line of business (LOB) and department.
- Here's an example policy for line of business A, department 1:

```
path "lob_a/dept_1/*" {
    capabilities = ["read", "list", "create", "delete", "update"]
}
```

- This policy grants all standard capabilities to all secrets mounted under `lob_a/dept_1/` by using the glob character (`*`).

Vault Policy CLI Commands

- Vault policies can be added to a Vault server using Vault's CLI, UI, or API.
- The command to add a policy with the CLI is `vault policy write`.
- Here is a command that creates a policy called "lob-A-dept-1" from the HCL file "lob-A-dept-1-policy.hcl":
  `vault policy write lob-A-dept-1 lob-A-dept-1-policy.hcl`
- Here is a command that associates this policy with a Userpass user:
  `vault write auth/userpass/users/joe/policies policies=lob-A-dept-1`

Chapter 5 Review

- Does Vault grant access to secrets by default?
- What are the policy capabilities that correspond to HTTP verbs?
- What CLI command can be used to add a policy to Vault?

Chapter 5 Review

- Does Vault grant access to secrets by default?
  - No
- What are the policy capabilities that correspond to HTTP verbs?
  - `create`, `read`, `update`, `delete`, and `list`
- What CLI command can be used to add a policy to Vault?
  - `vault policy write`

Chapter 6

[Lab] Integrate Vault with Kubernetes

[Lab] Integrate Vault with Kubernetes

Lab:

- integrate external vault with kubernetes
- vault will be managed by one team and workload is deployed to kubernetes

Pre-requisite:

- Vault Server (dev mode)
- K8S (minikube)

[Lab-1] Testing Pod in K8S can reach to external Vault

Steps to follows:

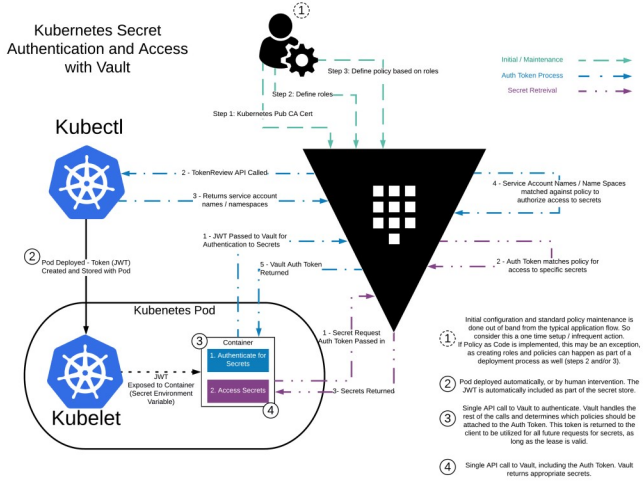
- Create sample KV secret in vault
- Testing that pod in k8s can reach to vault server

```
$ vault kv put secret/devwebapp/config username='giraffe' password='salsa'
======== Secret Path ========
secret/data/devwebapp/config

======= Metadata =======
Key                Value
---                -----
created_time       2025-01-07T15:39:19.396203585Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1

$ vault kv get -format=json secret/devwebapp/config | jq ".data.data"
{
  "password": "salsa",
```

[Lab-1] Testing Pod in K8S can reach to external Vault

```
$ kubectl apply -f devwebapp.yaml
$ kubectl get pod -o wide
NAME        READY   STATUS    RESTARTS   AGE     IP           NODE       NOMINATED NODE   READINESS GATES
devwebapp   1/1     Running   0          2m28s   10.244.0.6   minikube   <none>           <none>

$ kubectl exec devwebapp -- curl -s localhost:8080 ; echo
{"password"=>"salsa", "username"=>"giraffe"}
```

[Lab-2] Install Vault Agent in K8S using helm



[Lab-2] Install Vault Agent in K8S using helm

Steps to follows:

- K8S:
  - install helm `vault` to enable agent mode only
  - create secret that associated to vault service-account that created from helm install
  - verify installation, agent-injector deployment should be up and running
- Vault:
  - retrieve token value, public CA K8S and Kube API Server to enable vault authentication mode k8s in vault
  - define policy and link this policy with k8s authentication mode, which is bound to specific service-account and namespace in K8S

[Lab-2] Install Vault Agent in K8S using helm

Install vault by helm

```
$ helm install vault hashicorp/vault \
>     --set "global.externalVaultAddr=http://192.168.5.49:8200"
NAME: vault
LAST DEPLOYED: Tue Jan  7 16:03:38 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Thank you for installing HashiCorp Vault!

Now that you have deployed Vault, you should look over the docs on using
Vault with Kubernetes available here:

https://developer.hashicorp.com/vault/docs
```

[Lab-2] Install Vault Agent in K8S using helm

checking vault-agent-injector is installed

```
$ kubectl get pod
NAME                                    READY   STATUS    RESTARTS   AGE
devwebapp                               1/1     Running   0          6m34s
vault-agent-injector-6679dc894f-qnlj5   1/1     Running   0          57s
```

service account is created under the hood by helm

```
$ kubectl describe serviceaccount vault
Name:                vault
Namespace:           default
Labels:              app.kubernetes.io/instance=vault
                     app.kubernetes.io/managed-by=Helm
                     app.kubernetes.io/name=vault
                     helm.sh/chart=vault-0.29.1
Annotations:         meta.helm.sh/release-name: vault
                     meta.helm.sh/release-namespace: default
Image pull secrets:  <none>
Mountable secrets:   <none>
Tokens:              <none>
Events:              <none>
```

[Lab-2] Install Vault Agent in K8S using helm

we create a secret which is bound to service-account vault in the same namespace that vault is installed

```
cat > vault-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: vault-token-g955r
  annotations:
    kubernetes.io/service-account.name: vault
type: kubernetes.io/service-account-token
EOF
kubectl apply -f vault-secret.yaml
```

[Lab-2] Install Vault Agent in K8S using helm

Enable vault authentication mode K8S

```
TOKEN_REVIEW_JWT=$(kubectl get secret $VAULT_HELM_SECRET_NAME --output='go-template={{ .data.token }}' | base64 --de
KUBE_CA_CERT=$(kubectl config view --raw --minify --flatten --output='jsonpath={.clusters[].cluster.certificate-auth
KUBE_HOST=$(kubectl config view --raw --minify --flatten --output='jsonpath={.clusters[].cluster.server}')
echo $KUBE_HOST
https://192.168.49.2:8443

$ vault write auth/kubernetes/config \
>      token_reviewer_jwt="$TOKEN_REVIEW_JWT" \
>      kubernetes_host="$KUBE_HOST" \
>      kubernetes_ca_cert="$KUBE_CA_CERT" \
>      issuer="https://kubernetes.default.svc.cluster.local"
Success! Data written to: auth/kubernetes/config

vault write auth/kubernetes/role/devweb-app \
     bound_service_account_names=internal-app \
     bound_service_account_namespaces=default \
```

[Lab-2] Install Vault Agent in K8S using helm

Setup a readonly policy

```
vault policy write devwebapp - <<EOF
path "secret/data/devwebapp/config" {
  capabilities = ["read"]
}
EOF
```

Associated policy `devwebapp` with role `devweb-app`, bound to `internal-sa` service account and `default` namespace

```
vault write auth/kubernetes/role/devweb-app \
     bound_service_account_names=internal-app \
     bound_service_account_namespaces=default \
     policies=devwebapp \
     ttl=24h
```

At this step, we understand that, if pod want to read the sercret, it need to use service account `internal-sa` in namespace `default` with role `devweb-app`

[Lab-2] Install Vault Agent in K8S using helm

It's time to start a pod to test our implementation

```
apiVersion: v1
kind: Pod
metadata:
  name: devwebapp-with-annotations
  labels:
    app: devwebapp-with-annotations
  annotations:
    vault.hashicorp.com/agent-inject: 'true'  # allow injector mutate the pod spec
    vault.hashicorp.com/role: 'devweb-app'  # role that pod use
    # The name of the secret is any unique string after vault.hashicorp.com/agent-inject-secret-
    # The value is the path in Vault where the secret is located
    vault.hashicorp.com/agent-inject-secret-credentials.txt: 'secret/data/devwebapp/config'
spec:
  serviceAccountName: internal-app
  containers:
    - name: app
```

[Lab-2] Install Vault Agent in K8S using helm

By default, vault data will be stored in `/vault` folder, since we use secrets, then the path will be `/vault/secrets/[file-name]`

```
$ kubectl exec -it devwebapp-with-annotations -c app -- cat /vault/secrets/credentials.txt
data: map[password:salsa username:giraffe]
metadata: map[created_time:2025-01-07T15:39:19.396203585Z custom_metadata:<nil> deletion_time: destroyed:false versi
```

[Lab-2] Install Vault Agent in K8S using helm

Try another example, where we can customize the output of secret with annotations

```
apiVersion: v1
kind: Pod
metadata:
  name: devwebapp-with-annotations-template
  labels:
    app: devwebapp-with-annotations-template
  annotations:
    vault.hashicorp.com/agent-inject: 'true'
    vault.hashicorp.com/role: 'devweb-app'
    vault.hashicorp.com/agent-inject-secret-credentials.txt: 'secret/data/devwebapp/config'
    vault.hashicorp.com/agent-inject-template-app-config.txt: |
            {{- with secret "secret/data/devwebapp/config" -}}
            USERNAME={{ .Data.data.username }}
            PASSWORD={{ .Data.data.password }}
            {{- end -}}
spec:
```

You can see that at the end of the day, we can customize the output as we want, to fit with application requirement

```
$ kubectl exec -it devwebapp-with-annotations-template -c app -- ls /vault/secrets
app-config.txt  credentials.txt
$ kubectl exec -it devwebapp-with-annotations-template -c app -- ls /vault/secrets/app-config.txt
/vault/secrets/app-config.txt
$ kubectl exec -it devwebapp-with-annotations-template -c app -- cat /vault/secrets/app-config.txt
USERNAME=giraffe
PASSWORD=salsa
```

[Lab-2] Install Vault Agent in K8S using helm

End of lab

Chapter 7

[Lab] Integrate Vault with GitLab Community version

[Lab-3] Integrate Vault with GitLab Community version

Notes:

- Community version of gitlab does not support vault integration as easy as enterprise version does.
- However, the process of integration can be done through this guide line

References:

- https://docs.gitlab.com/ee/ci/secrets/index.html#configure-your-vault-server
- https://developer.hashicorp.com/well-architected-framework/security/security-cicd-vault
- https://docs.gitlab.com/ee/ci/secrets/hashicorp_vault.html

[Lab-3] Integrate Vault with GitLab Community version

Steps:

- Enable jwt authentication, we need to insert some CA certificate to let vault trust the token it received from gitlab if you are using self-hosted gitlab locally with no public certificates
- Setup sample secret in vault
- Prepare the vault policy
- Setup role in jwt authentication and link to above vault policy
- Setup sample gitlab-ci.yml to test

[Lab-3] Integrate Vault with GitLab Community version

Enable jwt authentication

```
$ vault auth enable jwt
Success! Enabled jwt auth method at: jwt/
```

Setup jwt config authentication with self-hosted CA (since it is locally hosted). oidc_discovery_url is the link that vault can reach to check the public known oidc information

```
cat /usr/local/share/ca-certificates/rootCA.crt

vault write auth/jwt/config \
        oidc_discovery_url="https://gitlab.homelab.duychu.link" \
        bound_issuer="gitlab.homelab.duychu.link" \
        oidc_discovery_ca_pem="$(cat /usr/local/share/ca-certificates/rootCA.crt)"
Success! Data written to: auth/jwt/config
```

[Lab-3] Integrate Vault with GitLab Community version

Prepare vault policy, assume we setup 2 secret for staging and production environment

```
$ vault policy write myproject-staging - <<EOF
# Policy name: myproject-staging
#
# Read-only permission on 'secret/myproject/staging/*' path
path "secret/myproject/staging/*" {
  capabilities = [ "read" ]
}
EOF
Success! Uploaded policy: myproject-staging

$ vault policy write myproject-production - <<EOF
# Policy name: myproject-production
#
# Read-only permission on 'secret/myproject/production/*' path
path "secret/myproject/production/*" {
  capabilities = [ "read" ]
```

[Lab-3] Integrate Vault with GitLab Community version

One role for staging named myproject-staging. The bound claims is configured to only allow the policy to be used for the main branch in all projects. Note that: the bound_claims has many fields, so depend on use case, we can limit access to vault with project_id, branch, ...

```
$ vault write auth/jwt/role/myproject-staging - <<EOF
{
  "role_type": "jwt",
  "policies": ["myproject-staging"],
  "token_explicit_max_ttl": 60,
  "user_claim": "user_email",
  "bound_audiences": ["http://vault.homelab.duychu.link:8200", "https://gitlab.homelab.duychu.link"],
  "bound_claims": {
    "ref": "main",
    "ref_type": "branch"
  }
}
EOF
```

Doing the samething for role `myproject-production`

```
$ vault write auth/jwt/role/myproject-production - <<EOF
{
  "role_type": "jwt",
  "policies": ["myproject-production"],
  "token_explicit_max_ttl": 60,
  "user_claim": "user_email",
  "bound_audiences": ["http://vault.homelab.duychu.link:8200", "https://gitlab.homelab.duychu.link"],
  "bound_claims_type": "glob",
  "bound_claims": {
    "ref_protected": "true",
    "ref_type": "branch",
    "ref": "main"
  }
}
EOF
```

Role is configured properly

```
$ vault list auth/jwt/role
Keys
----
myproject-production
myproject-staging
```

[Lab-3] Integrate Vault with GitLab Community version

Update correct bound issuer, this is tricky and you need to check the jwt token that gitlab issues

```
vault write auth/jwt/config \
    oidc_discovery_url="https://gitlab.homelab.duychu.link" \
    # before without https:// --> bound_issuer="gitlab.homelab.duychu.link" \
    bound_issuer="https://gitlab.homelab.duychu.link" \
    oidc_discovery_ca_pem="$(cat /usr/local/share/ca-certificates/rootCA.crt)"
```

[Lab-3] Integrate Vault with GitLab Community version

Checkout sample jwt issued from gitlab

```
read_secrets:
  image: harbor.homelab.duychu.link/library/platform-swiss-army-knife:latest
  id_tokens:
    VAULT_AUTH_TOKEN:
      # this is mistake: we should use aud is vault, since vault is the audience of this token
      aud: https://gitlab.homelab.duychu.link
  script:
    - echo $VAULT_AUTH_TOKEN
    - echo $VAULT_AUTH_TOKEN | awk -F. '{print $2}'
    - export VAULT_ADDR=http://vault.homelab.duychu.link:8200
    # authenticate and get token. Token expiry time and other properties can be configured
    # when configuring JWT Auth - https://developer.hashicorp.com/vault/api-docs/auth/jwt#parameters-1
    - export VAULT_TOKEN="$(vault write -field=token auth/jwt/login role=myproject-staging jwt=$VAULT_AUTH_TOKEN)"
    # use the VAULT_TOKEN to read the secret and store it in an environment variable
    - export PASSWORD="$(vault kv get -field=password secret/myproject/staging/db)"
    - echo $PASSWORD
```

[Lab-3] Integrate Vault with GitLab Community version