Securing Kubernetes Secrets with HashiCorp Vault

What is HashiCorp Vault?

HashiCorp Vault is a secret management tool that is used to store sensitive values and access it securely. A secret can be anything, such as API encryption keys, passwords, or certificates. Vault provides encryption services and supports authentication and authorization.

We can run Vault in high-availability (HA) mode and standalone mode. The standalone mode runs a single Vault server which is less secure and less resilient that is not recommended for production grade setup.

Installation of HashiCorp Vault:

```
kubectl create ns vault
```

Now add vault repo to Helm

```
helm repo add hashicorp https://helm.releases.hashicorp.com
```

you can use Helm to install the vault repository you added:

```
helm install vault hashicorp/vault  -n vault
```

Let's verify the installation by running the below-mentioned command:

```
helm install vault hashicorp/vault  -n vault
```

Let's verify the installation by running the below-mentioned command:

```
kubectl get pods -n vault
```

```
vault-0                                    0/1     Running   1 (5d21h ago)   37d
vault-agent-injector-58dc8b8c87-m29v2      1/1     Running   1 (2d6h ago)    38d
root@k8s-jump-host:~#
```

To unseal Vault, we need to initialize it. Just exec into pod vault-0 and initialize your Vault instance

```
kubectl exec -it vault-0 vault operator init -n vault
```

Your output will look similar to the following:

```
Recovery Key 1: FKjt5wkzN5bUBIuR52KrPP1c2Il/f7RZdn5E+ipfNF8s
Recovery Key 2: FCzUyduESPyavh6QtqWZpdnUDKa3bEEpBHbX3NgTrCiU
Recovery Key 3: Tf7FVEpj5tdJLqQqNw/Jt0OytRI5FAZYig/yafSVz3Xg
Recovery Key 4: duLpa/6IozTOR0mkO7sp0CwmnI+1DsC6d2+oZG/A1CIZ
Recovery Key 5: pyVFs/rRFEk9rSn57Ru+KeuAQzW6eurl3j0/pS/JRpXD
Initial Root Token: s.d0LAlSnAerb4a7d6ibkfxrZy
Success! Vault is initialized
Recovery key initialized with 5 key shares and a key threshold of 3. Please
securely distribute the key shares printed above.
```

After unsealing we can see the Vault unsealer in 1/1 state.

In order to make vault-0 visible, we need to login using our Initial Root Token.

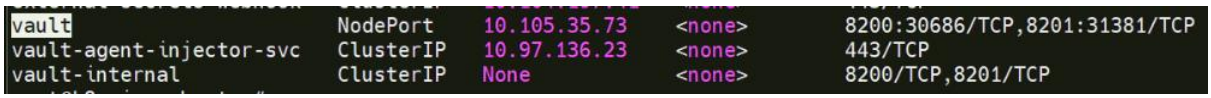
```
kubectl exec -it vault-0 -- vault login hvs.sjZCfzqEWoxDEIZ5AhYujvk7 -n vault
```

Accessing Vault GUI:

To access Vault Web UI we need to change our service type from ClusterIp to NodePort.

```
kubectl get svc -n vault
```

Now edit the svc ‘vault’ and change the svc type to NodePort



- Kubectl get pods -o wide -n vault [to check on which node the vault pod is running]
- Kubectl get nodes [to get ip of node on which Prometheus pod is running]

Now, access the vault GUI by enter http://Nodeport:30686 in your web browser [you can see the port in above image]

Inject Vault secrets into pod

We will be focusing on injecting a Vault secret into an application using External Secrets.

An ExternalSecret object declares how to fetch the secret data, while the controller converts all ExternalSecrets to Secrets.

By using ExternalSecrets object, now we can inject secrets inside the pods that are stored in Vault.

But the external-secrets controller needs to be authenticated with Vault before making any request to retrieve the secrets, so we need to enable the Kubernetes authentication method and attach a policy to it.

Enable Kubernetes Auth Method:

```
- kubectl exec -it vault-0 sh -n vault
- vault login s.d0LAlSnAerb4a7d6ibkfxrZy
- vault auth enable Kubernetes
```

Configure the kubernetes authentication method by running below command from inside the active Vault pod:

```
vault write auth/kubernetes/config \
token_reviewer_jwt="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \
kubernetes_host=https://${KUBERNETES_PORT_443_TCP_ADDR}:443 \
kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

Attach a policy to Kubernetes auth method:

```
vault write auth/kubernetes/role/k8s-role \
bound_service_account_names=* \
bound_service_account_namespaces=* \
policies=readonly \
ttl=1h
```

Installing the External Secrets Operator:

In this step, you will install the External Secrets Operator via Helm into your Kubernetes cluster.

you can then add the **external-secrets** repo to Helm:

```
helm repo add external-secrets https://charts.external-secrets.io
```

Next, make sure you have the most up-to-date repo:

```
helm repo update
```

Once the repo has been added, you can use Helm to install the external-secrets repository you added:

```
helm install external-secrets \
external-secrets/external-secrets \
-n vault \
--create-namespace \
--set installCRDs=true
```

With the -n vault option, you're specifying that the external-secrets repository should be installed in the vault namespace. With --create-namespace, the namespace will be created it if it doesn't exist. Lastly, you set --set installCRDS to true to make sure that the Custom Resource Definitions for External Secrets Operator are installed with it.

In this step, you installed the External Secrets Operator via Helm into your Kubernetes cluster, which will allow you to create a SecretStore and an ExternalSecret. In the next step, you'll create a SecretStore.

Creating a Secret Store:

In this step, you will create a SecretStore, which is what External Secrets Operator uses to store information about how to communicate with the given secrets provider. But before you work with the External Secrets Operator, you'll need to add your Vault token inside Kubernetes so that the External Secrets Operator can communicate with the secrets provider.

You are going to put this token inside of Kubernetes as a secret so that External Secrets Operator can use it to communicate with Vault. You can do this by running the following command, replacing the highlighted portion with your Vault token:

```
kubectl create secret generic vault-token --from-literal=token=YOUR_VAULT_TOKEN
```

You can verify this by running the following command, which gets all secrets in the default namespace

The output will look something like the following:

NAME	TYPE	DATA	AGE
Vault-token	Opaque	1	2m

create a secret-store.yaml file and paste the following contents into it. Be sure to fill in the data that is specific to your setup, such as the server field (highlighted), which is the endpoint where your Vault server can be accessed:

Secret-store.yaml

```yaml
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: vault-backend
spec:
  provider:
    vault:
      server: http://vault.vault.svc:8200
      path: kv
      version: v1
      auth:
        tokenSecretRef:
          name: "vault-token"
          key: "token"
```

Create the SecretStore by running the following command:

```
kubectl apply -f secretstore.yaml -n vault
```

Creating an External Secret:

In this step, you will create an ExternalSecret, which is the main resource in the External Secrets Operator. The ExternalSecret resource tells ESO to fetch a specific secret from a specific SecretStore and where to put the information. This resource is very important because it defines what secret you'd like to get from the external secret provider, where to put it, which secret store to use, and how often to sync the secret, among several other options.

To create an ExternalSecret, begin by creating a file called external-secret.yaml and add the following code:

external-secret.yaml

```yaml
apiVersion: external-secrets.io/v1alpha1
kind: ExternalSecret
metadata:
  name: eureka-test
  namespace: default
spec:
  refreshInterval: "15s" # How often this secret is synchronized
  secretStoreRef:
    name: vault-backend
    kind: SecretStore
  target:
    name: # If not present, then the secretKey field under data will be used
    creationPolicy: Owner # This will create the secret if it doesn't exist
  data:
    - secretKey: eureka
```

Next, create this resource with the following command:

```
kubectl apply -f ./external-secret.yaml
```

After it has been applied, you can then make sure that all is well by running the following command:

```
kubectl get ExternalSecret eureka-test
```

Injecting secrets in Pod:

Now, let's understand how to Inject this secret object inside a pod using an environment variable.

Deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: album-service
spec:
  containers:
  - name: nginx
    image: cybersolutioncontainer/familymicroservices:album-service-55
    env:
      - name: DISCOVERY_URL
        valueFrom:
          secretKeyRef:
            name: eureka-test
            key: eureka
```

Let's run the below command to check the status of external-secret

```
kubectl get externalsecret
```

```
NAME                                              STORE           REFRESH INTERVAL   STATUS
externalsecret.external-secrets.io/eureka-test    vault-backend   15s                SecretSynced
```

So here, External Secret has been created successfully

Vault GUI:

You can view added secret's key and value in vault's GUI