Vault

Install vault

```
kubectl create ns vault
kubectl config set-context --current --namespace vault
kubectl get all

helm repo add hashicorp https://helm.releases.hashicorp.com
helm install vault --set='ui.enabled=true' --set='ui.serviceType=NodePort' --
set='server.dataStorage.enabled=false'   hashicorp/vault --version 0.24.0
NAME: vault
LAST DEPLOYED: Tue Apr 18 14:40:08 2023
NAMESPACE: vault
STATUS: deployed
REVISION: 1
NOTES:
Your release is named vault. To learn more about the release, try:
  $ helm status vault
  $ helm get manifest vault
```

Vault Initialization

```
kubectl get pod
kubectl exec -it vault-0 -- vault status
kubectl exec -it vault-0 -- /bin/sh

vault operator init
Unseal Key 1: 9yKQv2wqbE8eFDV2lXf6pQDlqxRwe625A2nemYHUIrKE
Unseal Key 2: k1AyFsjiqb4DGeBlviYWAyKiSdvamAppmUiV/lNoiMn+
Unseal Key 3: LQnEmWY/Ao6q7tB3MQppiYMDhkXfWNtF/gyVNSBggMZw
Unseal Key 4: sfioFbDoOLxF6lsUvosNIEL3lA7Y7TtfAbVkUjfMz1KM
Unseal Key 5: wG/MfmU+ZvjI8yMHO49K75VrwbN9OrtmMsTPWiTAC1yr
Initial Root Token: s.VF5xrdAHb04InF4ov91SUGif
Vault initialized with 5 key shares and a key threshold of 3. Please securely
distribute the key shares printed above. When the Vault is re-sealed,
restarted, or stopped, you must supply at least 3 of these keys to unseal it
before it can start servicing requests.
Vault does not store the generated master key. Without at least 3 keys to
reconstruct the master key, Vault will remain permanently sealed!
It is possible to generate new unseal keys, provided you have a quorum of
existing unseal keys shares. See "vault operator rekey" for more information.
```

The vault operator init command is used to initialize a new Vault server instance. It is part of the vault operator command group, which is used to manage and operate a Vault server.

When you run the vault operator init command, Vault generates and outputs a set of five unseal keys and a root token. These are critical pieces of information that are needed to manage and access the Vault server.
Here is a brief overview of what happens when you run vault operator init:

Vault generates a new encryption key that is used to protect sensitive data stored in the Vault server.

Vault generates five unseal keys, which are used to unlock the encryption key and decrypt the sensitive data. Each unseal key is a 26-character string that is randomly generated by Vault.

Vault generates a root token, which is a privileged access token that can be used to perform administrative tasks on the Vault server.

Vault outputs the five unseal keys and root token to the console.

It's important to keep the unseal keys and root token safe and secure, as they provide access to sensitive data stored in the Vault server. Typically, the unseal keys are distributed among multiple trusted parties to prevent a single point of failure. The root token should be stored securely and used only when necessary.

```
vault operator unseal wG/MfmU+ZvjI8yMHO49K75VrwbN9OrtmMsTPWiTAC1yr
Unseal Key (will be hidden):
Key             Value
---             -----
Seal Type         shamir
Initialized       true
Sealed            true
Total Shares      5
Threshold         3
Unseal Progress   1/3
Unseal Nonce      cccf48d6-b7e9-5014-36b4-66894e0f3b0f
Version           1.8.3
Storage Type      file
HA Enabled        false
```

This command is used to unseal a Vault server that has been initialized but is not yet active. When a Vault server is initialized, it is sealed to protect its sensitive data. To start using the Vault server, you must unseal it by providing a certain number of unseal keys (as determined by the initialization process). Each time the vault operator unseal command is executed with an unseal key, the Vault server is moved one step closer to being fully unsealed.

```
vault login s.VF5xrdAHb04InF4ov91SUGif
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.
Key               Value
---               -----
token             s.VF5xrdAHb04InF4ov91SUGif
token_accessor    KI5fSY50wm2U3DEkyGBfEPKZ
token_duration    ∞
token_renewable   false
```

```
token_policies      ["root"]
identity_policies   []
policies            ["root"]
```

This command is used to authenticate with the Vault server using a specified method, such as using a username and password or a client token. Once authenticated, the user is granted an access token, which is used to access Vault resources.

Vault Secrets Engine

```
vault secrets enable -path=crds kv-v2
```

This command is used to enable the KV (key-value) version 2 secrets engine on a Vault server. The -path option specifies the path at which to mount the secrets engine.

In Vault, a secrets engine is a component that is responsible for generating, storing, and managing secrets. A secret is any piece of sensitive data that needs to be kept secure, such as passwords, API keys, and certificates.
Vault supports multiple types of secrets engines, each of which is designed to handle a specific type of secret. Some of the commonly used secrets engines in Vault include:

- KV (Key-Value) secrets engine: Used to store and retrieve arbitrary key-value pairs.
- Database secrets engine: Used to dynamically generate database credentials on behalf of users and applications.
- AWS secrets engine: Used to generate and manage AWS access keys and secret access keys.
- PKI (Public Key Infrastructure) secrets engine: Used to issue and manage X.509 certificates.

Each secrets engine in Vault is mounted at a specific path in the Vault hierarchy, and can be enabled and configured using Vault commands. Once a secrets engine is enabled, applications and users can use Vault APIs or CLI commands to generate and retrieve secrets from that engine.
Overall, the secrets engines in Vault make it easy for developers and operators to manage sensitive data securely, while also providing a centralized location for managing all of their secrets.

```
vault policy list
```

This command is used to list all of the policies that have been defined on the Vault server. Policies are used to define access control rules for Vault resources.

```
vault  kv get crds/mysql
```

This command is used to read a key-value pair from a Vault server. The key-value pair is specified using a path, such as secret/foo.

```
vault  kv put crds/mysql username=root password=09132
```

This command is used to write a key-value pair to a Vault server. The key-value pair is specified using a path, such as secret/foo, and a set of key-value pairs, such as key=value.

```
vault  kv metadata get crds/mysql
```

This command is used to retrieve the metadata for a key-value pair in a Vault server. The metadata includes information such as the time the key-value pair was created and last updated, and the version number of the pair.

Vault Authorization

```
cat <<EOF > /home/vault/app-policy.hcl
path "crds/data/mongodb" {
  capabilities = ["create", "read", "update", "patch", "delete", "list"]
}
path "crds/data/mysql" {
  capabilities = ["read"]
}
EOF
```

In Vault, a policy is a set of rules that define what actions a user or application can perform on a given set of paths within the Vault hierarchy. Policies are written in the HCL (HashiCorp Configuration Language) syntax and can be defined using a text editor or using Vault CLI commands.
In the example policy you provided, there are two paths specified with different capabilities assigned to them:
"crds/data/mongodb": This path specifies that the policy allows the capabilities to create, read, update, patch, delete and list key-value pairs located in the "mongodb" path under the "crds/data" path in Vault.
"crds/data/mysql": This path specifies that the policy allows the capability to read key-value pairs located in the "mysql" path under the "crds/data" path in Vault.
The capabilities defined in the policy specify what actions the user or application is authorized to perform on the given path(s). In this case, the policy allows a user or application to perform a range of actions on MongoDB data, but only to read MySQL data.
Once the policy is written, it can be applied to a user or application in Vault, allowing them to perform the actions defined in the policy on the paths specified.
Overall, policies in Vault provide a powerful tool for managing access to sensitive data, allowing administrators to define granular access controls based on the specific needs of their users and applications.

```
vault policy write app /home/vault/app-policy.hcl
```

This command is used to create or update the policy named "app" on the Vault server. The policy specifies access control rules for Vault resources that are used by a particular application.

```
vault policy read app
```

This command is used to read the policy named "app" from the Vault server. The "app" policy specifies access control rules for Vault resources that are used by a particular application.

Authentication

```
vault token create
```

This command is used to create a new token in HashiCorp Vault. Tokens are used to authenticate and authorize clients to interact with Vault. When you run this command, Vault will generate a new token with a specified set of permissions and a TTL (time to live) that determines how long the token will be valid.

```
vault token revoke
```

This command is used to revoke a token in HashiCorp Vault. When you run this command, Vault will immediately invalidate the specified token, meaning that it can no longer be used to authenticate or authorize clients.

```
vault auth enable kubernetes
```

This command is used to enable the Kubernetes authentication method in HashiCorp Vault. When you run this command, you are telling Vault to configure itself to accept authentication requests from Kubernetes.

```
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
```

This command is used to configure the Kubernetes authentication method in HashiCorp Vault. Specifically, it sets the configuration parameters for the method, such as the location of the Kubernetes API server and the location of the Kubernetes CA certificate.

```
SET ROLE
vault write auth/kubernetes/role/phpapp \
bound_service_account_names=app \
bound_service_account_namespaces=vault \
policies=app \
ttl=1h
```

This command is used to configure a role in the Kubernetes authentication method in HashiCorp Vault. When you run this command, you are specifying which Kubernetes service account (in this case, app) can be used to authenticate with Vault, which namespace the service account belongs to (in this case, vault), which policies the role should have access to (in this case, app), and how long the token issued by this role will be valid (in this case, 1h).

```
exit
kubectl describe clusterrolebinding vault-server-binding
```

This command is used to describe a ClusterRoleBinding in Kubernetes. A ClusterRoleBinding is a way to bind a ClusterRole (a set of permissions in Kubernetes) to a specific Kubernetes service account. When you run this command, you are getting information about the ClusterRoleBinding named vault-server-binding.

PHP APPLICATION DEMO

```
git clone https://github.com/sidd-harth/php-vault-example.git
cd php-vault-example/
docker build -t php:vault .
kubectl apply -f php-app-k8s-deploy.yaml
```

In php-app-k8s-deploy.yaml, we have code below:

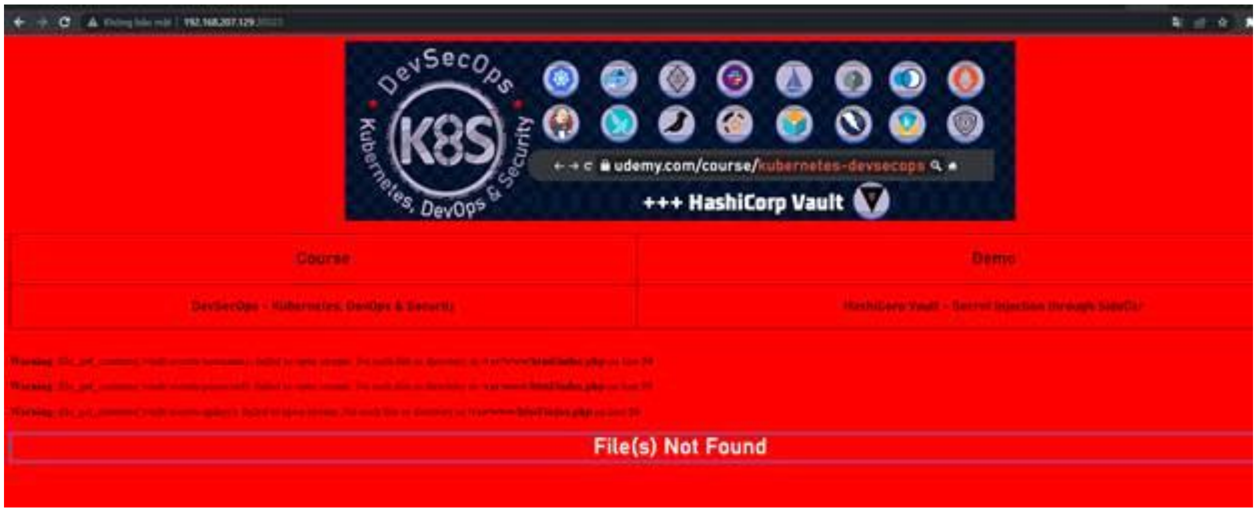
```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app
  labels:
    app: php
```

It is will be interactive with command:
SET ROLE vault write auth/kubernetes/role/phpapp \ bound_service_account_names=app \ bound_service_account_namespaces=vault \ policies=app \ ttl=1h

After run apply, The Kubernetes ServiceAccount named "app" in your deployment file is connected to the Vault command vault write auth/kubernetes/role/phpapp through the bound_service_account_names parameter.
When you run the Vault command with bound_service_account_names=app, it tells Vault that any authentication requests coming from the Kubernetes ServiceAccount named "app" should be authorized by this Vault role (phpapp).
Additionally, the bound_service_account_namespaces parameter specifies the Kubernetes namespace where the ServiceAccount is located (vault in this case).
Finally, the policies parameter specifies the Vault policies that should be granted to any tokens issued by this role. In this case, the app policy is being granted.
The ttl parameter specifies the time-to-live for any tokens issued by this role. In this case, the token will expire and be revoked after 1 hour.

```
kubectl get svc
NAME            TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
php             NodePort   10.106.55.59   <none>        80:30323/TCP   57s
```

We try to access the app and now the app has a red background and displays secret information


DevSecOps
K8S
Kubernetes, DevOps & Security
udemy.com/course/kubernetes-devsecops
+++ HashiCorp Vault

Course	Demo
DevSecOps - Kubernetes, DevOps & Security	HashiCorp Vault - Secret Injection through SideCar

Warning: [illegible] **/var/www/html/index.php** on line [illegible]

Warning: [illegible] **/var/www/html/index.php** on line [illegible]

Warning: [illegible] **/var/www/html/index.php** on line [illegible]

File(s) Not Found

```
cat patch-annotations.yaml
spec:
  template:
    metadata:
      annotations:
        vault.hashicorp.com/agent-inject: "true"
        vault.hashicorp.com/agent-inject-secret-username: "crds/data/mysql"
        vault.hashicorp.com/role: "phpapp"
```

Annotations in Vault HashiCorp are a way to specify metadata about Kubernetes objects that indicate how Vault should interact with them. Annotations are key-value pairs that are added to Kubernetes objects such as pods, deployments, and services to instruct Vault on how to inject secrets or otherwise interact with the objects.
Some common annotations used in Vault include:

- vault.hashicorp.com/agent-inject: "true": This annotation enables the Vault Agent sidecar injector to inject secrets into a pod.
- vault.hashicorp.com/agent-inject-secret-<key>: "<path>": This annotation specifies the path to a secret in Vault and the key in the secret that should be injected into the pod.
- vault.hashicorp.com/role: "<name>": This annotation specifies the name of a Vault role that should be used to authenticate requests from the Kubernetes object.

Annotations can be specified in a Kubernetes manifest file or added to an existing object using the kubectl annotate command. Annotations provide a flexible way to integrate Vault with Kubernetes applications and manage secrets securely.

```
kubectl patch deploy php -p "$(cat patch-annotations.yaml)"
```

This command applies the annotations defined in the patch-annotations.yaml file to the Kubernetes Deployment named php. Specifically, it adds the annotations vault.hashicorp.com/agent-inject: "true", vault.hashicorp.com/agent-inject-secret-username: "crds/data/mysql", and vault.hashicorp.com/role: "phpapp"to the metadata of the pod template.

```
kubectl describe pod php-7744998f8f-lcwx8
 Args:
     echo ${VAULT_CONFIG?} | base64 -d > /home/vault/config.json && vault agent -
config=/home/vault/config.json    State:         Terminated
```

This command provides a detailed description of the Kubernetes pod named php-7744998f8f-lcwx8.

```
 kubectl exec -it pod/php-7744998f8f-lcwx8 -n vault -c php -- cat /vault/secrets/username
data: map[apikey:813290vjlkad password:09132 username:root]
metadata: map[created_time:2023-04-19T02:07:39.660745457Z deletion_time:
destroyed:false version:1]
```

This command executes a command inside the container named php in the php-7744998f8f-lcwx8 pod, and prints the contents of the /vault/secrets/username file. Specifically, it decodes the value of the VAULT_CONFIG environment variable, writes it to a file at /home/vault/config.json, and starts the Vault agent process with that configuration file. The agent then retrieves the secret named username from Vault and writes it to the /vault/secrets/username file.

```
cat patch-annotations-template.yaml
spec:
  template:
    metadata:
      annotations:
        vault.hashicorp.com/agent-inject: "true"
        vault.hashicorp.com/agent-inject-status: "update"
        vault.hashicorp.com/agent-inject-secret-username: "crds/data/mysql"
        vault.hashicorp.com/agent-inject-template-username: |
          {{- with secret "crds/data/mysql" -}}
                    {{ .Data.data.username }}
          {{- end }}
        vault.hashicorp.com/agent-inject-secret-password: "crds/data/mysql"
        vault.hashicorp.com/agent-inject-template-password: |
          {{- with secret "crds/data/mysql" -}}
                    {{ .Data.data.password }}
          {{- end }}
        vault.hashicorp.com/agent-inject-secret-apikey: "crds/data/mysql"
        vault.hashicorp.com/agent-inject-template-apikey: |
          {{- with secret "crds/data/mysql" -}}
                    {{ .Data.data.apikey }}
          {{- end }}
        vault.hashicorp.com/role: "phpapp"
```

It using the Vault Agent Sidecar Injector.
The annotations are divided into two sections: agent and vault. The agent annotations change the Vault Agent containers templating configuration, such as what secrets they want, how to render them, etc. The vault annotations change the Vault Agent containers authentication configuration, such as what role they use, what certificates they need, etc.

Here is a brief explanation of each annotation in the template:

- `vault.hashicorp.com/agent-inject: "true"`: This enables injection for the pod[1].
- `vault.hashicorp.com/agent-inject-status: "update"`: This blocks further mutations by adding the value `injected` to the pod after a successful mutation[1].
- `vault.hashicorp.com/agent-inject-secret-username: "crds/data/mysql"`: This tells the Vault Agent to retrieve the secret `username` from the path `crds/data/mysql` in Vault[1].
- `vault.hashicorp.com/agent-inject-template-username: | ...`: This tells the Vault Agent how to render the secret `username` using a Go template[1].
- `vault.hashicorp.com/agent-inject-secret-password: "crds/data/mysql"`: This tells the Vault Agent to retrieve the secret `password` from the same path as `username`[1].
- `vault.hashicorp.com/agent-inject-template-password: | ...`: This tells the Vault Agent how to render the secret `password` using a Go template[1].
- `vault.hashicorp.com/agent-inject-secret-apikey: "crds/data/mysql"`: This tells the Vault Agent to retrieve the secret `apikey` from the same path as `username` and `password`[1].
- `vault.hashicorp.com/agent-inject-template-apikey: | ...`: This tells the Vault Agent how to render the secret `apikey` using a Go template[1].
- `vault.hashicorp.com/role: "phpapp"`: This tells the Vault Agent what role to use for authenticating with Vault using the Kubernetes auth method[1].

Reference:
(1) Agent Sidecar Injector Annotations | Vault | HashiCorp Developer. https://developer.hashicorp.com/vault/docs/platform/k8s/injector/annotations.
(2) Running Vault with Kubernetes - HashiCorp. https://www.hashicorp.com/products/vault/kubernetes.
(3) » Vault Agent Injector Examples - Vault by HashiCorp. https://developer.hashicorp.com/vault/docs/platform/k8s/injector/examples
https://testdriven.io/blog/managing-secrets-with-vault-and-consul
https://craftech.io/blog/manage-your-kubernetes-secrets-with-hashicorp-vault/

```
kubectl patch deploy php -p "$(cat patch-annotations-template.yaml)"
kubectl exec -it pod/php-697475985f-n58bq -n vault -c php -- cat
/vault/secrets/username
Root
```

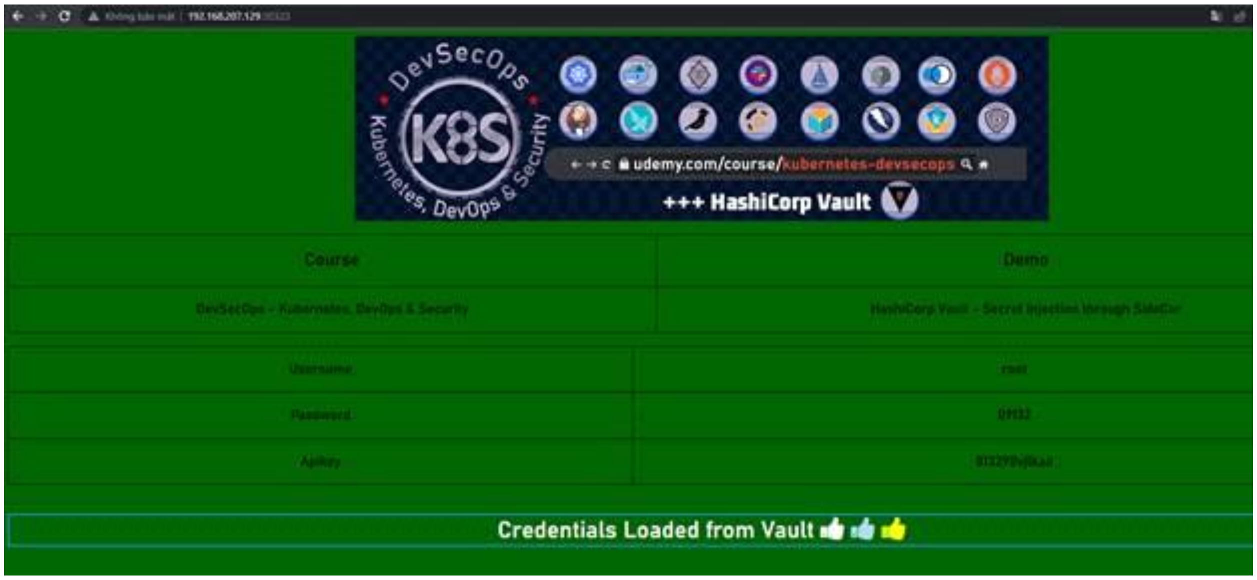
- kubectl: This is the command-line tool used to interact with Kubernetes clusters.
- patch: This sub-command is used to update or modify resources in a Kubernetes cluster.
- deploy: This specifies the type of resource to be patched, which in this case is a

deployment.

- php: This is the name of the deployment that we want to patch.
- -p "$(cat patch-annotations-template.yaml)": This is an option that specifies the patch to be applied to the deployment. The patch is specified using a YAML or JSON file that contains the updated metadata for the deployment. The $(cat patch-annotations-template.yaml) command substitution is used to read the contents of the file patch-annotations-template.yaml and pass it as a string to the -p option.

Overall, the command kubectl patch deploy php -p "$(cat patch-annotations-template.yaml)" is used to update the metadata for the deployment named "php" by applying a patch contained in the YAML file patch-annotations-template.yaml. The specific changes made to the metadata will depend on the contents of the patch file.

Now, we refresh the app and we see that the app has get the secrets

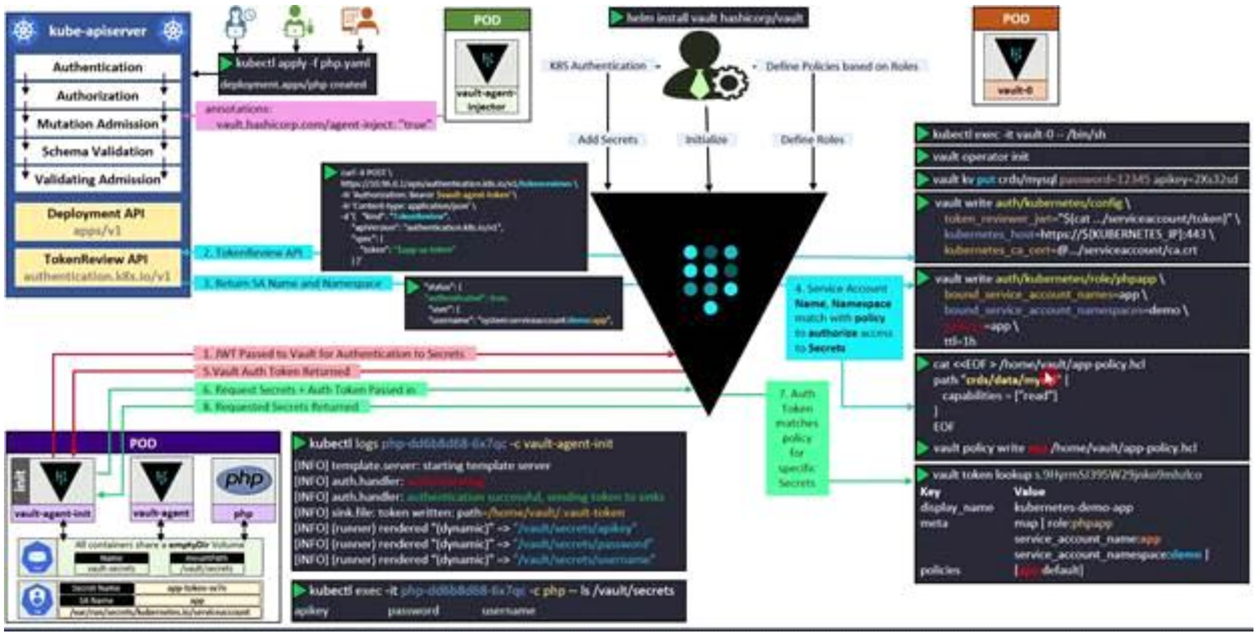

DevSecOps
K8S
Kubernetes, DevOps & Security
udemy.com/course/kubernetes-devsecops
+++ HashiCorp Vault

Course	Demo
DevSecOps – Kubernetes, DevOps & Security	HashiCorp Vault – Secret Injection through SideCar

Username	root
Password	09132
ApiKey	813279vj8kal

Credentials Loaded from Vault 👍👍👍

How it works Internally


kube-apiserver
Authentication
Authorization
Mutation Admission
Schema Validation
Validating Admission
Deployment API
apps/v1
TokenReview API
authentication.k8s.io/v1
annotations:
vault.hashicorp.com/agent-inject: "true"
POD
vault-agent-injector
helm install vault hashicorp/vault
K8S Authentication
Define Policies based on Roles
Add Secrets
Initialize
Define Roles
POD
vault-0
2. TokenReview API
3. Return SA Name and Namespace
4. Service Account Name, Namespace match with policy to authorize access to Secrets
1. JWT Passed to Vault for Authentication to Secrets
5.Vault Auth Token Returned
6. Request Secrets + Auth Token Passed in
8. Requested Secrets Returned
7. Auth Token matches policy for specific Secrets
POD
init
vault-agent-init
vault-agent
php
All containers share a emptyDir Volume

```
kubectl apply -f php.yaml
deployment.apps/php created
```

```
kubectl exec -it vault-0 -- /bin/sh
```

```
vault operator init
```

```
vault kv put crds/mysql password=12345 apikey=2Xs32sd
```

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

```
vault write auth/kubernetes/role/phpapp \
  bound_service_account_names=app \
  bound_service_account_namespaces=demo \
  policies=app \
  ttl=1h
```

```
cat <<EOF > /home/vault/app-policy.hcl
path "crds/data/mysql" {
  capabilities = ["read"]
}
EOF
vault policy write app /home/vault/app-policy.hcl
```

```
vault token lookup s.9Hym5J395W29jsko9mhzlco
Key            Value
display_name   kubernetes-demo-app
meta           map [ role:phpapp
               service_account_name:app
               service_account_namespace:demo ]
policies       [app default]
```

```
kubectl logs php-dd6b8d68-6x7qc -c vault-agent-init
[INFO] template.server: starting template server
[INFO] auth.handler: authenticating
[INFO] auth.handler: authentication successful, sending token to sinks
[INFO] sink.file: token written: path=/home/vault/.vault-token
[INFO] (runner) rendered "(dynamic)" => "/vault/secrets/apikey"
[INFO] (runner) rendered "(dynamic)" => "/vault/secrets/password"
[INFO] (runner) rendered "(dynamic)" => "/vault/secrets/username"
```

```
kubectl exec -it php-dd6b8d68-6x7qc -c php -- ls /vault/secrets
apikey          password          username
```

Now, we'll try to see how Vault works internally.
The recommended installation method is through the latest Vault helm site, which supports the Vault-k8s injection functionality. It launches two pods, Vault-0 and the Vault-agent-injector. An admin or Vault user can exit into the Vault-0 pod to initialize the Vault, add secrets, define policies, configure authentication and roles.
The vault-agent-injector pod is a Kubernetes mutation webhook controller. The controller intercepts pod events and applies mutation to the pod if annotation exist within the request. When we deploy our application to Kubernetes, it generally goes through the kube-apiserver.
It goes through the authentication, authorization, mutation, admission controller if any, does a round of schema validation after mutation, and it also does a validation admission controller before getting persisted in the etcd data store. Then a scheduler will deploy the pod, mount the required service accounts on one of the available notes. If we execute into this pod and search for /vault/secretsdirectory, it would return not found because the secrets are not yet mounted from Vault.

How does this container get the secrets from the Vault server?
We can use the kubectl patch command to apply the annotations to an existing pod object. This will be intercepted by the Vault-agent-injector webhook service which will invoke the mutation admission controller and then inject the correct init and the sidecar containers. The init container will pre-populate the shared memory volume with the requested secrets prior to the other container starting. The sidecar container, which is the Vault agent container, will continue to authenticate and render secrets to the same location as the pod runs.

How do these containers fetch secrets from the Vault server?
Checking the logs of the Vault init connector, initially, it tries to authenticate with the Vault server. The primary method of authentication with Vault when using the Vault-agent-injector is the service account attached to the pod. It is going to send the JSON Web Token to Vault for authentication. Vault now has to validate the pod service account JSON Web Token.
For Kubernetes authentication, the service account must be bond to a Vault role and a

policy granting access to the secrets desire. For accessing the Kubernetes TokenReview API to validate the provided part service account token is still valid or not, the Vault service account used in this auth method will need to have access to the TokenReview API. The service account should be granted system auth delegator permission.
Once the TokenReview API request has been sent, the TokenReview API in the queue, API server, it will respond back the status authenticated through as a payload along with the details of the service account like the namespace and the service account name. This is how the curl command generally looks like. It's a post command to the TokenReview API. Within the headers, you pass in the Vault-agent-injector's service account token. Within the payload, you send in the pod service account token to check if it is still valid or not. All this is sent to the TokenReview API which will try to validate it.
If the pod app service account token is valid, it's going to respond back with a payload of status authenticated as true. Within the payload, it also sends back the service account name and to which namespace it belongs to. The service account name and namespace will be matched with the policy to authorize access to the secrets. Once authorized, Vault is going to create auth token and return it to the init connectors.
The token is stored in the /home/vault/.vault-token directory. Once the authentication is successful, it is stored in this directory. Next, the init container will request the Vault for the secrets and passing the auth token which it received in the previous call. Vault will match the auth token with policy for specific secrets, and secrets are returned to the init connectors. The init container will store the secret in empty directory volume which is shared by all the three containers within the pod.
Now, when we execute into the pod and search for /vault/secrets directory, it would display all the rendered secrets. This is a pretty straightforward workflow pattern for injecting a secret into a running application that has no native vault logic built in. The application only needs to concern themselves with finding a secret at a specific file path. That's it for now. Thank you.