Jack Lei

51 Followers · About Follow

Vault: setting up Kubernetes auth and database secrets engine

Jack Lei Jun 1, 2019 · 8 min read

Implementation details for authenticating services to Vault to retrieve dynamic secrets/credentials.

Source: https://medium.com/@gmaliar/dynamic-secrets-on-kubernetes-pods-using-vault-35d9094d169

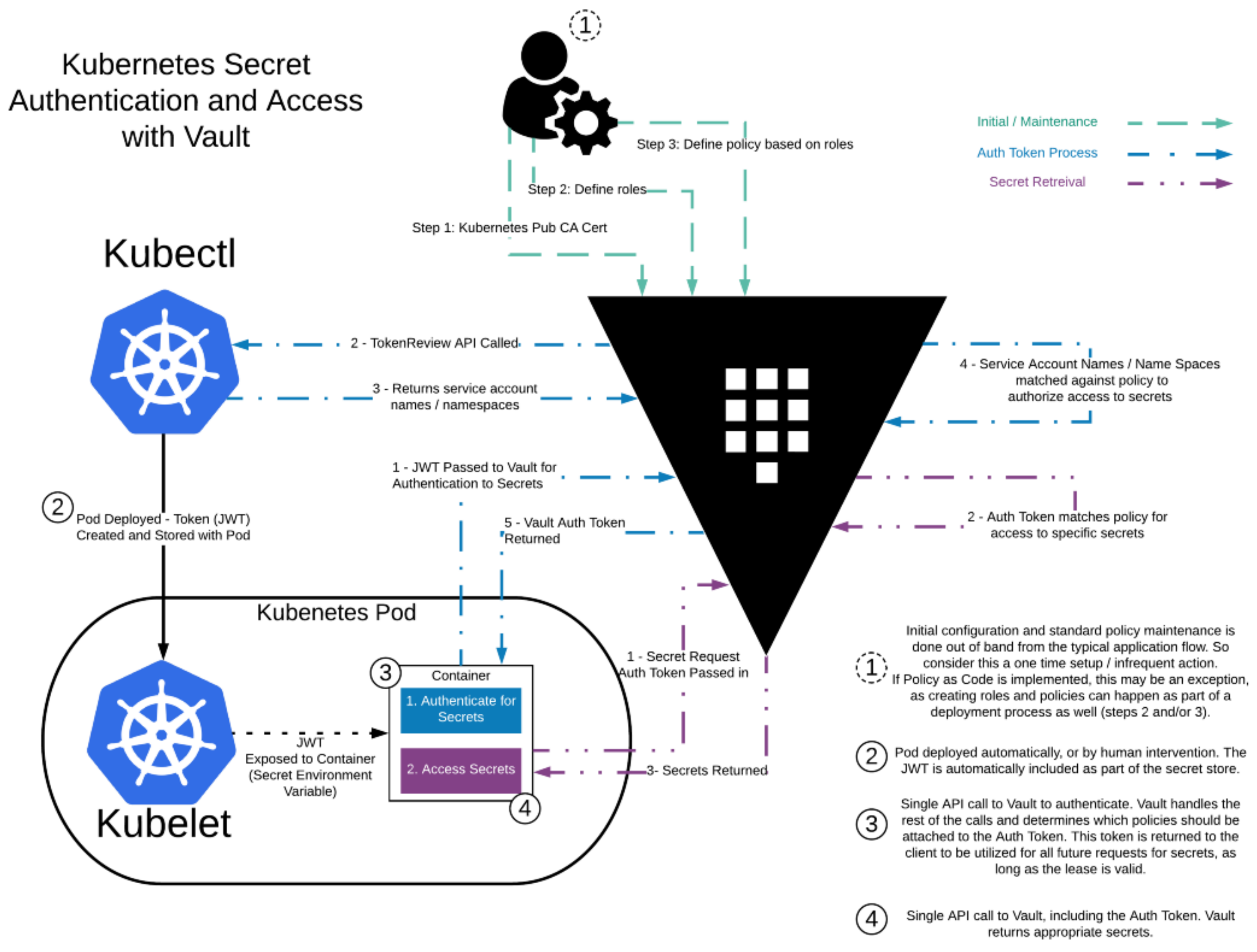
Introduction

Take a simple application that needs to connect to a database. The application just needs to know the host, username, and password. Cool, easy enough. Let's secure the hell out of it.

The username and password are the keys to your castle. In production, nothing should have the secrets except for the application. Not even the pipeline. If the username and password to the database were to be accidentally exposed, the access should not be indefinite. Reduce and mitigate risk.

Kubernetes is the standard for container orchestration. That makes it really easy to scale our applications up. Each instance should have its own user and password combination.

Bonus points: The app should not be Vault aware.


Kubernetes Secret Authentication and Access with Vault
1
Step 3: Define policy based on roles
Step 2: Define roles
Step 1: Kubernetes Pub CA Cert
Initial / Maintenance
Auth Token Process
Secret Retreival
Kubectl
2 - TokenReview API Called
3 - Returns service account names / namespaces
4 - Service Account Names / Name Spaces matched against policy to authorize access to secrets
1 - JWT Passed to Vault for Authentication to Secrets
5 - Vault Auth Token Returned
2 - Auth Token matches policy for access to specific secrets
2 Pod Deployed - Token (JWT) Created and Stored with Pod
Kubenetes Pod
1 - Secret Request Auth Token Passed in
3 Container
1. Authenticate for Secrets
2. Access Secrets
4
JWT Exposed to Container (Secret Environment Variable)
3- Secrets Returned
Kubelet
1 Initial configuration and standard policy maintenance is done out of band from the typical application flow. So consider this a one time setup / infrequent action. If Policy as Code is implemented, this may be an exception, as creating roles and policies can happen as part of a deployment process as well (steps 2 and/or 3).
2 Pod deployed automatically, or by human intervention. The JWT is automatically included as part of the secret store.
3 Single API call to Vault to authenticate. Vault handles the rest of the calls and determines which policies should be attached to the Auth Token. This token is returned to the client to be utilized for all future requests for secrets, as long as the lease is valid.
4 Single API call to Vault, including the Auth Token. Vault returns appropriate secrets.

Source: https://learn.hashicorp.com/vault/identity-access-management/vault-agent-k8s

After consulting with Hashicorp and watching Seth Vargo's talks numerous times, I am convinced this is the proper implementation with my infrastructure at this time (May 2019). This is the breakdown that I needed to understand the concepts. If this can help someone else, awesome.

tl;dr: The goal is to secure secrets and reduce the risk if they are exposed.

What do I have?

- Kubernetes cluster for my apps separated by namespaces.

- Vault instance in its own Kubernetes cluster.
- CloudSQL instance which is accessible by both clusters.

What do I want?

- Secure the secrets retrieval process.
- Reduce the risk when the database user and password are compromised.

This is not a guide to for an initial Vault setup on Kubernetes. I have another write up for that purpose.

Deploying Vault with etcd backend in Kubernetes

I needed a secrets management tool that can be highly available, on-premise, cloud-native, run on low resources, and…

medium.com

Prerequisites

Before we get into the good stuff, I do assume a few things are in place.

- Vault is initialized, unsealed, and set up with storage backend.
- Kubernetes cluster.
- MySQL instance (I'm using CloudSQL).

These environment variables should be accessible.

- VAULT_TOKEN — Vault's root token.
- VAULT_CACERT — Vault's CA cert path.

Setup

In my mind, there are two main workflows for setting up. One is for setting a new cluster to communicate with Vault, primarily focused on the authentication method (this happens per cluster). The second is for onboarding a new application with a new database connection (this happens per application).

In both cases, we can think of four main sections of responsibility that can be configured and pass information.

> *Kubernetes — Home for the application. Responsible for making requests to Vault and validating tokens.*
>
> *Vault Kubernetes Authentication Method — Perform authentication and are responsible for assigning identity and a set of policies to a service account.*
>
> *Vault Policies — Policies provide a declarative way to grant or forbid access to certain paths and operations in Vault.*
>
> *Vault Database Secrets Engine — Generates database credentials dynamically based on configured roles for the MySQL database.*

jacklei/vault-helpers

Contribute to jacklei/vault-helpers development by creating an account on GitHub.

github.com

I have created a small repository of scripts to help. Each section is represented in their own scripts in *vault-helpers*. If you are planning on using my scripts, please read the Assumptions section in README.md before continuing.

A New Cluster

Kubernetes

Create *vault-auth* service account in the *default* namespace.

```
kubectl create serviceaccount vault-auth
```

Give the vault-auth service account the token reviewer role. This allows the service account to validate other service accounts in their namespace.

```
kubectl apply -f -<<EOH
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: role-tokenreview-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: vault-auth
  namespace: default
EOH
```

Using vault-helpers scripts.

```
configure/kubernetes.sh --context apps-cluster --setup
```

Database Secrets Engine

Enable the database secrets engine at the default path *database/*

```
vault secrets enable database
```

using vault-helpers scripts

```
configure/vault-secrets-database.sh --enable
```

Kubernetes Authentication Method

In order for Vault to communicate with our Kubernetes cluster, there are a few pieces of information that are required.

Kubernetes host — Address that Vault can connect with.

```
k8s_host="$(kubectl config view --minify | grep server | cut -f 2- -
d ":" | tr -d " ")"
```

Cluster authority data — Certificate to verify the connection.

```
k8s_cacert="$(kubectl config view --raw --minify --flatten -o
jsonpath='{.clusters[].cluster.certificate-authority-data}' | base64
--decode)"
```

User token — vault-auth service account with token reviewer role. Vault will interact with the cluster using this service account.

```
secret_name="$(kubectl get serviceaccount vault-auth -o go-
template='{{ (index .secrets 0).name }}')"

tr_account_token="$(kubectl get secret ${secret_name} -o go-
template='{{ .data.token }}' | base64 --decode)"
```

Enable Kubernetes auth method at the default path: *auth/kubernetes*

```
vault auth enable kubernetes
```

Configure the Kubernetes auth method.

```
vault write auth/kubernetes/config
token_reviewer_jwt="${token_reviewer_jwt}"
kubernetes_host="${kubernetes_host}"
kubernetes_ca_cert="${kubernetes_cacert}"
```

Using vault-helpers scripts.

```
configure/vault-auth-kubernetes.sh --token-reviewer-jwt
$(configure/kubernetes.sh --token-reviewer-jwt) --k8s-host
```

```
$(configure/kubernetes.sh --k8s-host) --k8s-cacert-base64
$(configure/kubernetes.sh --k8s-cacert) --enable --configure
```

New App

Kubernetes

Create a new namespace. Our demo application will go here, not default. We're not savages.

```
kubectl create namespace demo
```

Create a service account in that namespace. This service account will be used by our deployments that require secrets from Vault. Notice the service account name, this is different than the *vault-auth* service account created earlier. They have different uses.

```
kubectl --namespace=demo create serviceaccount vault
```

Add a configmap with the Vault address, preferably an internal IP or DNS address.

```
kubectl --namespace=demo create configmap vault --from-literal
"vault_addr=https://vault.local:8200"
```

Add Vault's cert to a secret.

```
kubectl --namespace=demo create secret generic vault-tls --from-file
"${VAULT_CACERT}"
```

Using vault-helpers scripts.

```
configure/kubernetes.sh --context apps-cluster --vault-addr
https://vault.local:8200 --vault-cacert $VAULT_CACERT --new-
namespace demo
```

Database Secrets Engine

Configure Vault to connect to the database. The user needs enough privilege to create and delete users and grant access. We will need to allow the *demo-role* role to access this connection, which we will create in a moment.

```
vault write database/config/demo-db plugin_name=mysql-database-plugin connection_url="{{username}}:{{password}}@tcp(demodb.local)/" allowed_roles="demo-role" username="root" password="password"
```

Rotate the root password. A new password will be generated and used. That means only Vault will have the password, not even you.

```
vault write -f database/rotate-root/demo-db
```

Add the role which allows for temporary users to be created in the database.

```
vault write database/roles/demo-role db_name=demo-db creation_statements="CREATE USER '{{name}}'@'%' IDENTIFIED BY '{{password}}';GRANT SELECT ON *.* TO '{{name}}'@'%';" default_ttl="1h" max_ttl="24h"
```

Using vault-helpers scripts.

```
configure/vault-secrets-database.sh --db-name demo-db --host demodb.local:3306 --username root --password password --role-name demo-role --configure --rotate-root --role
```

Vault Policies

Create a policy that would allow the creation of a temporary set of credentials.

```
vault policy write demo-policy -<<EOF
path "database/creds/demo-role" {
  capabilities = ["read"]
}
EOF
```

Using vault-helpers scripts.

```
configure/vault-general.sh --policy demo-db-r
```

`demo-db-r` is the name of the policy. This holds some assumptions. I don't want to keep creating new policies for every database that I have. As long as I keep the Kubernetes namespace `demo` similar to the database role `demo-role` , I can use policy templates. After running the script, the policy will look something like this:

```
path
"database/creds/{{identity.entity.aliases.auth_kubernetes_3626dffe.m
etadata.service_account_namespace}}-role" {
  capabilities = ["read"]
}
```

Kubernetes Auth Method

Create a Kubernetes Auth role that grants the policy (that allows for the creation of credentials) to the service account in a specific namespace. All of which have been set up already.

```
vault write auth/kubernetes/role/demo
bound_service_account_names=vault
bound_service_account_namespaces=demo policies=demo-policy ttl=1h
```

Using vault-helpers scripts.

```
configure/vault-auth-kubernetes.sh --names vault --namespaces demo -
-policies demo-policy --role demo
```

Testing and Workflow

All of those moving parts may have been confusing, connecting the dots may take a while. Don't worry, I got you! I'll explain the important stuff, you can refer to Kubernetes or Vault docs for the others.

Let's create a temporary pod in our *demo* namespace and install some basic tools.

```
kubectl -n demo run -it --rm --image=alpine --serviceaccount=vault
test -- /bin/sh
apk add --update vim curl bash jq mysql-client
bash
```

The *vault* service account from the *demo* namespace was used in this pod. That means the service account's JWT is accessible from within the pod. This is the piece of information that will authenticate you to Vault.

Let's get the JWT and write it to a variable for easy access.

```
JWT="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"
```

If you were paying attention in the setup, we added a Kubernetes auth role (*auth/kubernetes/role/demo*) with the *vault* name in the *demo* namespace. The token that was retrieved is associated with the same service account.

Use the JWT to login and save the token.

```
TOKEN="$(curl --request POST --data '{"jwt": "'"$JWT"'", "role":
"demo"}' -s -k https://vault.local:8200/v1/auth/kubernetes/login |
jq -r '.auth.client_token')"
```

A few things are happening here.

- *auth/kubernetes/login* — we are logging into the *kubernetes* (default path)
- *demo* role is the kubernetes auth role (*auth/kubernetes/role/demo*)
- The *demo* kubernetes role was attached to the *demo-policy* policy
- The *demo-policy* policy has read access to *database/creds/demo-role* database role.

- The output is our client token. Think of this as your session cookie.

Use that user token you crazy person you.

```
curl --header "X-Vault-Token: $TOKEN" -s -k
https://vault.local:8200/v1/database/creds/demo-role | jq -r .data
```

We wrote to the *database/roles/demo-role*, to get the dynamic secret we need to read from *database/creds/demo-role*.

Try it out on your database. Pass in your user and password.

```
mysql -u$USER -p$PASS -h demodb.local
```

What's next?

This is cool and all but how do you apply this into a production-ready environment? I get it, I need my application to implement this workflow in a simple automated manner. Check this out:

Streamlining Application Secrets in Kubernetes with Pipelines

The "automagic" between Vault Secrets Engine, Kubernetes, and GitLab Pipelines.

medium.com

Conclusion

Yayyy!!! You did it!!!

There are obviously many ways to authenticate with Vault. Another approach was to use AppRole in the pipeline and keep Kubernetes Vault unaware. That would work and would be better than our current ways which is to have a flat file at our destination. Unfortunately, that forces us to trust our VCS. We would need to make sure our CI/CD secrets are restricted properly, but then if the job has access to the role id and secret id.. problems arise.

Placing the authentication closer to the edge meant less risk. Only a handful of people have access to the cluster and all accessible pieces can be rotated easily. This includes the service account JWT used for authentication, the user token has a TTL, and even the database credentials (TTL and/or number of uses).

References

This implementation has been **HEAVILY** influenced by Seth Vargo. Specifically the following:

So You Want to Run Vault in Kubernetes? - Seth Vargo, Google

sethvargo/vault-kubernetes-workshop

Steps and scripts for running @HashiCorp Vault on @GoogleCloudPlatform Kubernetes - sethvargo/vault-kubernetes-...

github.com

More links:

jacklei/vault-helpers

Contribute to jacklei/vault-helpers development by creating an account on GitHub.

github.com

Policies - Vault by HashiCorp

Policies are how authorization is done in Vault, allowing you to restrict which parts of Vault a user can access.

www.vaultproject.io

Kubernetes - Auth Methods - Vault by HashiCorp

The Kubernetes auth method allows automated authentication of Kubernetes Service Accounts.

www.vaultproject.io

MySQL/MariaDB - Database - Secrets Engines - Vault by HashiCorp

MySQL is one of the supported plugins for the database secrets engine. This plugin generates database credentials...

www.vaultproject.io

Vault Kubernetes DevOps