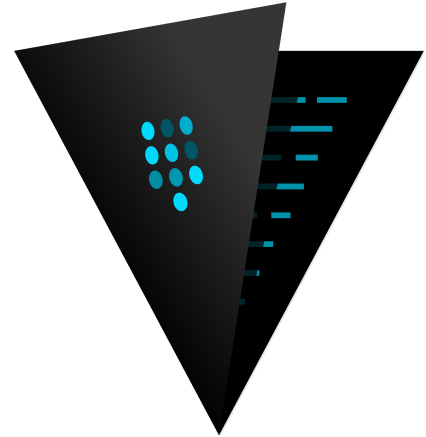


Workshop

Manage your cloud native secrets with Vault



Mission objectives:

In this workshop, we will deploy Vault into kubernetes and Vault will delegate access to secrets based on Service accounts in the Api-server. We will also create a username/password connection from a Vault front UI running in its own container.

Prerequisites:

A Kubernetes cluster up and running

- Ingress controller or cloud load balancer
- Kubectl configured

Need-to-know-stuff

Vault in this exercise is not hardened (no ssl).

Vault is using a simple file backend.

Table of content

1.0 Clone the repo	2
2.0 Setup Vault	3
2.1 Deploy Vault	3
2.2 Init Vault	4
2.3 Unseal Vault	6
2.4 Create policies	7
3.0 Setup username and password for UI	8
3.1 Setting up ingress and accessing the UI	9
4.0 Integrate Vault with Kubernetes	10
4.1 Create service account	10
4.2 Create cluster role binding	10
4.3 Enable the Kubernetes auth plugin	10
4.4 Create two ServiceAccounts for Vault access	11
4.5 Test Admin from other pod	12
4.6 Test Dev from other pod	12

1.0 Clone the repo

Clone our github repo to get the code. You will need it throughout this workshop. You will find it here :

In a terminal, go to a suitable place to put the repo, and then clone it.

```
cd /development
```

```
git clone
```

```
https://github.com/pragma-training/vault-workshop-docl-2018.git
```

```
cd vault-workshop-docl-2018
```

In this project, you will find

- scripts to deploy Vault
- Kubernetes manifests
- service accounts
- A Vault UI.
- Slides
- This document

2.0 Setup Vault

In this exercise we will deploy and init Vault, to get it ready.

2.1 Deploy Vault

In our Github repo you cloned, we have created a script called `deploy.sh`. Please read it to see what it does. The project is setup to use a nfs share to persist the data produced by the Vault pod.

In this workshop we will not have time to set up a NFS server, so instead, we made a special `-workshop.sh` file for you to run.

Now run the script `deploy-workshop.sh`

You should see the following output:

```
$ ./deploy-workshop.sh
```

Output:

```
deploying Vault to Cluster
configmap "vault-config" created
service "vault" created
service "vault-ui" created
deployment "vault" created
deployment "vault-ui" created
ingress "vault-ui" created
```

/Output:

Wait for the pods to download the container images and start. Then check the status like this:

```
$ kubectl get pods
```

Output:

NAME	READY	STATUS	RESTARTS	AGE
vault-1980104489-sb382	1/1	Running	0	3m
vault-ui-133818932-q77tj	1/1	Running	0	3m

/Output:

2.2 Init Vault

Now we are ready to initialize Vault. Lets get the pod-id and save it in a variable like this:

```
$ POD_NAME=$(kubectl get pods -l app=vault -o jsonpath="{.items[0].metadata.name}")
```

Exec interactive inside it like

```
$ kubectl exec -it $POD_NAME /bin/sh
```

You should now get a terminal inside the pod.

We are going to use the vault client to talk to the Vault server process. The client needs to know where to connect to, so we set the environment variable VAULT_ADDR to point to localhost on Vault's default port 8200 like this:

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

Now test that Vault is responding:

```
$ vault status
```

Output:

```
Error checking seal status: Error making API request.
```

```
URL: GET http://127.0.0.1:8200/v1/sys/seal-status
```

```
Code: 400. Errors:
```

```
* server is not yet initialized
```

/Output:

From the output we can see that Vault has not been initialized yet. So let's do that via the vault operator init command:

```
$ vault operator init
```

Output:

```
Unseal Key 1: K5Z0urzppKJU2o4+hxYCeTdoJTwPKsGVrPi57RfY4eD0
Unseal Key 2: 9AQe6Vi4T41vFi5Kgn1RsfaT8aCmY0a5DjAXwyfjXBtV
Unseal Key 3: bqAPMwndoCP6d7yjs7Zej8y0oeLTWTfNSFwaczmgIdy
Unseal Key 4: tfuY/7EH67dex4Fo80+B4eF0BvyWFsgPNy8u0Y0U5+zK
Unseal Key 5: oHkPJk5vFX0ijZE+WLW6+VGdF0kDjv4CdsWsDtUKenYL
```

Initial Root Token: 9e2c8535-bf25-619b-d50d-c806e6cf056b

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.

...

/Output:

NB: We will use the keys during the workshop, so copy them somewhere

Now Vault gave us 5 key shares and tells us to use at least 3 of them to unseal Vault. Let's check the status again

```
$ vault status
```

Output:

Key	Value
---	-----
Seal Type	shamir
Sealed	true
Total Shares	5
Threshold	3

```
Unseal Progress    0/3
Unseal Nonce       n/a
Version            0.9.3
HA Enabled         true
HA Mode            sealed
```

/Output:

2.3 Unseal Vault

we see that Vault is sealed. We will use 3 of our 5 keys to unseal it, like this:

```
$ vault operator unseal
K5Z0urzppKJU2o4+hxYCeTdoJTwPKsGVrPi57RfY4eD0
$ vault operator unseal
bqAPMwndoCP6d7yjs7Zej8y0oeLTWTfNSFwaczmjgIdy
$ vault operator unseal
oHkPJk5vFX0ijZE+WLW6+VGdF0kDjv4CdsWsDtUKenYL
```

Lets check the status again

```
$ vault status
```

Output:

Key	Value
---	-----
Seal Type	shamir
Sealed	false
Total Shares	5
Threshold	3
Version	0.9.3
Cluster Name	vault-cluster-e4cdf961
Cluster ID	4b9395f4-3e87-ad6a-f3d0-5db0bae69c81
HA Enabled	false

/Output:

Now Vault is not sealed.

2.4 Create policies

In Vault we map roles to permissions via Policies. In this example we will create two simple policies. One that can read and write, and another that can only read.

```
$ cat > /vault/admin-policy.hcl <<EOF
path "*" {
    capabilities = [ "create", "read", "update", "list" ]
}
EOF
```

```
$ cat > /vault/dev-policy.hcl <<EOF
path "*" {
    capabilities = [ "read", "list" ]
}
EOF
```

The first is for our Admin role. This can both create, read, update and list secrets. In path we have specified an asterix giving it these capabilities everywhere.

The other one is our Developer role. This only has read and list capabilities.

These are over simplified and should not be used in production.

In order to apply these policies, we need to login. We login as root, using the “Initial Root Token” we got after initializing Vault.

```
$ vault login 9e2c8535-bf25-619b-d50d-c806e6cf056b
```

Now apply the policies.

```
$ vault policy write admin /vault/admin-policy.hcl
$ vault policy write dev /vault/dev-policy.hcl
```

Check that the policies were created:

```
$ vault read sys/policy
```

Output:

Key	Value
---	-----
keys	[dev default admin root]
policies	[dev default admin root]

/Output:

3.0 Setup username and password for UI

Now that Vault is ready and has the policies we need, we will activate the userpass authenticator. This will allow us to login into Vault by a simple username and password.

Lets enable the userpass backend:

```
$ vault auth enable userpass
```

Now we create an admin user and bind it to the admin policy:

```
$ vault write auth/userpass/users/praqma \
    password="password" \
    policies="admin"
```

Note: you can set the password to something more secret if you want.

We do the same for our developer role:

```
$ vault write auth/userpass/users/praqma-dev \
    password="password" \
    policies="dev"
```

Check that the userpass backend works from cli by running :

```
$ vault login -method="userpass" username="praqma"
```

Output: (write 'password' if you didn't change the example)

Password (will be hidden):

Output:

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	e959aa4e-f542-5692-f2ed-cac1229396a5


```
token_accessor      2c09f529-1b25-a734-c304-59d797424a68
token_duration      768h
token_renewable     true
token_policies      [admin default]
token_meta_username praqma
/Output:
```

Now try from the UI.

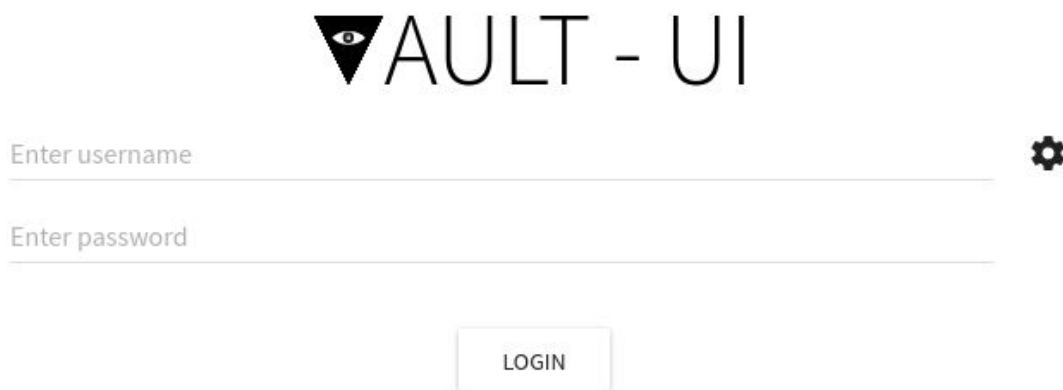
3.1 Setting up ingress and accessing the UI

The ingress for the ui will map vault-ui.example.com to the service of the ui. So if you are running on virtual machines with an ingress, you need to edit your /etc/hosts and add the ip of the machine running your ingress-controller with the domain vault-ui.example.com

Then in a browser, browse to <https://vault-ui.example.com>

On GKE run `kubectl get ingress` to get the ip. Wait for the load balancer to be ready.

You should see the following window:



VAULT - UI

Enter username

Enter password

LOGIN

Then login as the admin, and create a new secret.

The admin policy should allow you do to this.

Then logout and back in as the praqma-dev and try to do the same. You should be able to see the secret created by the admin, but not able to create a new one, as defined by our policies.

4.0 Integrate Vault with Kubernetes

4.1 Create service account

This service account is the one Vault will use to connect and talk with Kubernetes. When ever Vault needs to verify a token used by the kubernetes backend, this service account is in play. Without it, it wont work.

Keep your terminal that is exec into the Vault pod, and start a new for running this section.

We have a recipe for the service account in the repo, so simply run

```
$ kubectl apply -f sa/vault-auth.yaml
```

4.2 Create cluster role binding

Now that we have a service account, we need to bind it to a Kubernetes Role. We bind it to a ClusterRole called system:auth-deligator. This is sufficient for Vault.

We have a recipe for the ClusterRoleBinding in the repo, so simply run

```
$ kubectl apply -f clusterrolebinding/vault-auth-crb.yaml
```

4.3 Enable the Kubernetes auth plugin

Going back to your terminal inside your Vault pod, login again as root:

```
$ vault login 012fccfd-1ed4-66b8-c030-36b2260d75c7
$ vault auth enable kubernetes
```

For testing, let's give two accounts access to Vault, an "admin" and "dev" account.

First map the service account in the default namespace to the role and policy in Vault.

```
$ vault write auth/kubernetes/role/admin \
  bound_service_account_names=vault-admin \
  bound_service_account_namespaces=default \
  policies=admin \
  ttl=10h

$ vault write auth/kubernetes/role/dev \
  bound_service_account_names=vault-dev \
  bound_service_account_namespaces=default \
  policies=dev \
  ttl=10h
```

Then connect the Kubernetes Auth plugin to our Kubernetes master.

For that, we need to get the secret provided by Kubernetes that matches our Service Account.

In a new terminal, back on your host machine, run the following

```
$ kubectl get secrets
```

Output:

NAME	TYPE	DATA	AGE
default-token-5kz4b	kuber...-token	3	4h
vault-auth-token-5jzv8	kuber...-token	3	17m

/Output:

Locate the name of the vault secret, it'll be something like
"vault-auth-token-5jzv8"

Run the following, to print the token

```
$ kubectl get secrets vault-auth-token-5jzv8 -o
jsonpath="{.data.token}" | base64 -d
```

Copy the token, and export it in the terminal, inside the Vault pod,

```
$ TOKEN=ABCD...
```

And run,

```
$ vault write auth/kubernetes/config \
  token_reviewer_jwt="$TOKEN" \
  kubernetes_host=https://kubernetes \
  kubernetes_ca_cert=@/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

4.4 Create two ServiceAccounts for Vault access

We will create two ServiceAccounts. One admin and one dev. These will show different access level in Vault for different ServiceAccounts. They will be used by our applications running in the cluster.

We have a recipe for the service accounts in the repo, so simply run

```
kubectl apply -f sa/vault-admin.yaml
kubectl apply -f sa/vault-dev.yaml
```

Get the secret for each Service account

```
kubectl get secrets
```

NAME	TYPE	DATA	AGE
default-token-6njhv	kubernetes.io/service-account-token	3	118d
vault-admin-token-cq97w	kubernetes.io/service-account-token	3	9m
vault-auth-token-xtxf2	kubernetes.io/service-account-token	3	13m
vault-dev-token-8gd4f	kubernetes.io/service-account-token	3	1s

```
kubectl describe secret vault-admin-token-cq97w
```

```
Name:          vault-admin-token-cq97w
Namespace:     default
Labels:        <none>
Annotations:   kubernetes.io/service-account.name=vault-admin
```

```
kubernetes.io/service-account.uid=3c0783f0-0a67-11e8-9dbc-525400122de4
```

```
Type:  kubernetes.io/service-account-token
```

```
Data
```

====

```
token:      eyJhbGciOiJSUzI1NiIsInR5c.....xbYXyeG0ws5D_-BqA
ca.crt:     1025 bytes
namespace:  7 bytes
```

Do the same for vault-dev-token-xxx secret and save both of them in a textfile. We will use them later.

4.5 Test Admin from other pod

Go to the directory deployments and deploy our network tool.

```
cd deployment
cat nwtool-deployment.yaml
kubectl create -f nwtool-deployment.yaml
kubectl get pods

kubectl exec -it nwtool-982403051-qr96n bash
```

```
cat /var/run/secrets/kubernetes.io/serviceaccount/token
curl \
  --request POST \
  --data '{"jwt": "eyJhbGciOi.....-NxAJbaefV-A", "role": "admin"}' \
  http://vault.default.svc.cluster.local:8200/v1/auth/kubernetes/login | jq
```

4.6 Test Dev from other pod

```
curl \
  --request POST \
  --data '{"jwt": "eyJhbGciOi.....BKdQa908IsA", "role": "dev"}' \
  http://vault.default.svc.cluster.local:8200/v1/auth/kubernetes/login | jq
```

You should get a response like this :

```
{
  "request_id": "4b790854-1fd1-93ad-5aab-16e8309c93e2",
  "lease_id": "",
  "renewable": false,
  "lease_duration": 0,
  "data": null,
```

```

"wrap_info": null,
"warnings": null,
"auth": {
  "client_token": "b7922c1e-82c7-2341-9adc-f73103838e36",
  "accessor": "07e5fca8-32b0-dea3-c425-86b3414c1052",
  "policies": [
    "default",
    "dev"
  ],
  "metadata": {
    "role": "dev",
    "service_account_name": "vault-dev",
    "service_account_namespace": "default",
    "service_account_secret_name": "vault-dev-token-xtt46",
    "service_account_uid": "3c1c3b02-0a67-11e8-9dbc-525400122de4"
  },
  "lease_duration": 36000,
  "renewable": true,
  "entity_id": "68804bed-f7ad-cd1d-8a03-d34852b1e5e6"
}
}

```

Write down the `client_token`, we will use it to work with vault.

Now we create an entry using our service account and curl

```

export VAULT_TOKEN="b7922c1e-82c7-2341-9adc-f73103838e36"
curl \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  --request POST \
  --data '{"cloud": "native"}' \
  http://vault:8200/v1/secret/foo

```

Verify in the UI (login and check under secrets).

Now lets retrieve the secret, also via curl.

```

export VAULT_TOKEN="b7922c1e-82c7-2341-9adc-f73103838e36"
curl -s \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  http://vault:8200/v1/secret/foo | jq

```

4.7 Test Vault sync with Kubernetes

Now try and delete the vault-dev Service account in Kubernetes, and then try the curl command again.

You should now get this:

```
{
  "errors": [
    "lookup failed: [invalid bearer token, [Token has been
invalidated, invalid bearer token]]"
  ]
}
```