Hashicorp Vault Installation on Azure Kubernetes Services

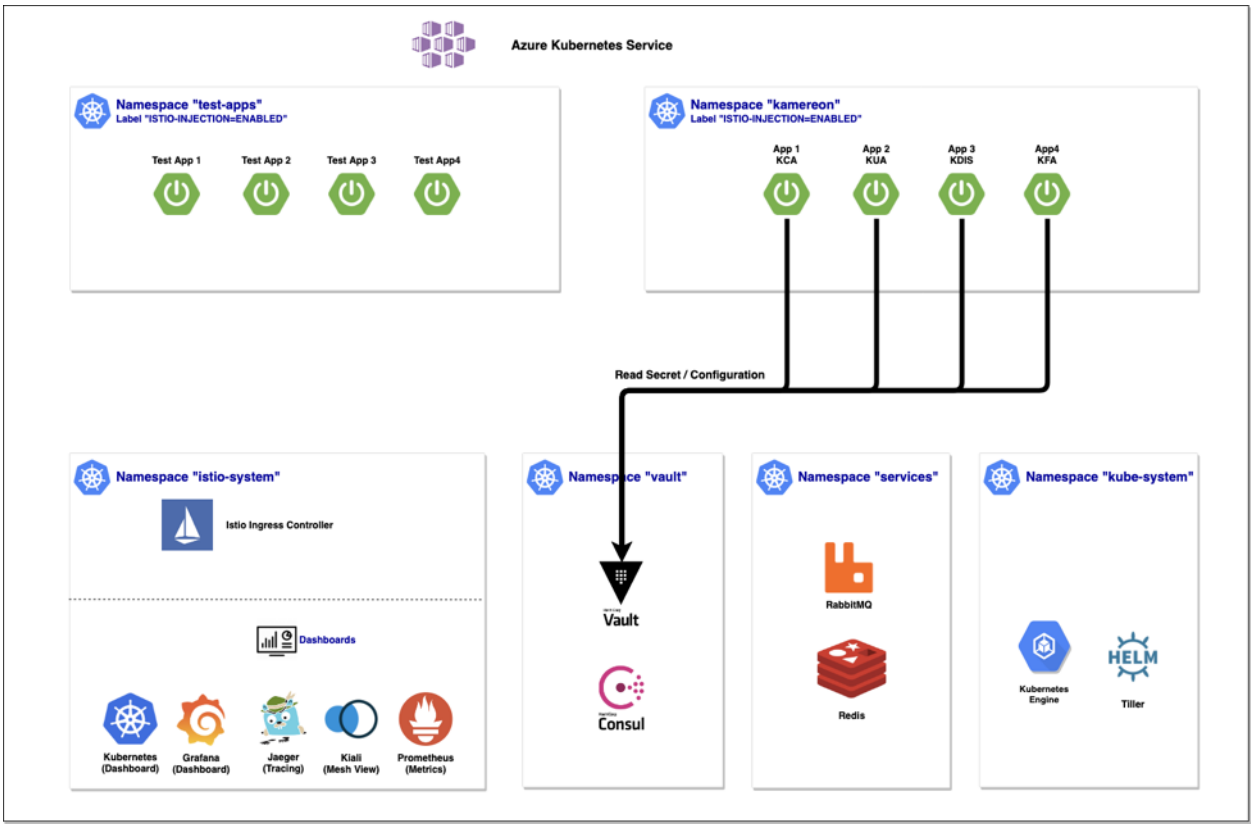
Content

1. Changes

Date	Version	Author	Changes
01/08/2019	1.0	Hichem BOUSSETTA	Initial version - Vault installation and basic setup and policy definition - Vault HA - Vault auto-unseal with Azure
09/08/2019	1.1	Hichem BOUSSETTA	- Integration with Spring Cloud Vault - Use of Kubernetes auth method in Vault
14/08/2019	1.2	Hichem BOUSSETTA	- Service account role configuration - Comparison between Vault / Kubernetes Secrets from client perspective
05/11/2019	1.3	Hichem BOUSSETTA	Added Vault OIDC Configuration

2. Overview

In this document, we will install a Consul-backed Hashicorp Vault Server on Kubernetes and configure it to store the secrets and credentials of applications.



After installation and creation of root account, vault must be configured with admin and provisioner policies.
https://www.hashicorp.com/resources/policies-vault

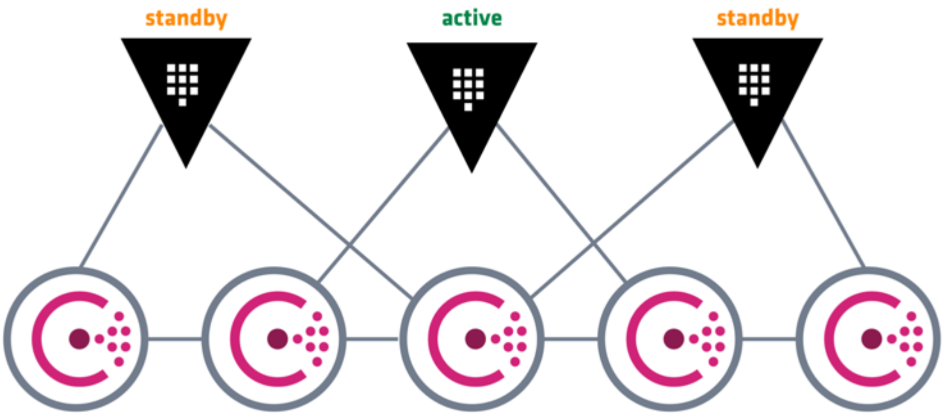
- **root** sets up initial policies for admin
- **admin** is empowered with managing a Vault infrastructure for a team or organizations
- **provisioner** configures secret backends and creates policies for client apps

The document will also cover how to auto-unseal the Vault cluster on a Azure Kubernetes Service by relying on the native cloud vault support.

Microsoft Recommendations regarding using Hashicorp Vault with AKS can be referred to from the following location:
https://azure.microsoft.com/fr-fr/resources/videos/azure-friday-using-hashicorp-vault-with-azure-kubernetes-service-aks/

3. Installation & Configuration

This paragraph describes a standard Hashicorp Vault installation on a Kubernetes cluster.
The installation will rely on helm installer and official vault and consul helm charts.
Vault can be installed with different storage backends. The one that is used here is Consul as explained in the graphics below:



- Vault and Consul are installed in high availability mode
- Each Consul pod will have its own dedicated persistent volume storage

Vault pods will be installed in master / slave configuration

3.1. Installation

- Install Consul

```
helm install --name consul stable/consul
```

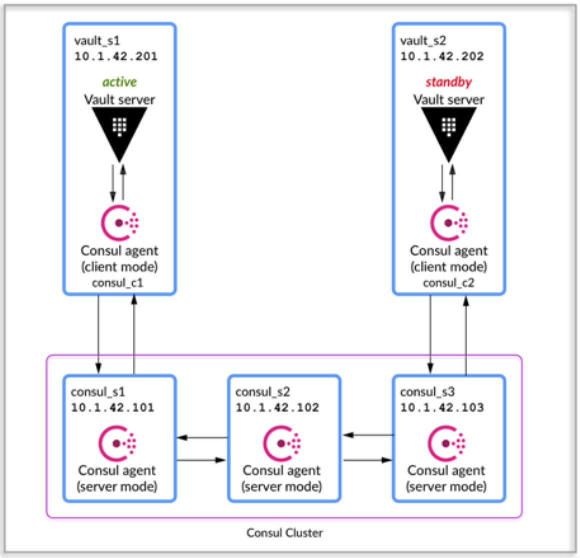
- Install Vault (Vault helm repo to be added beforehand)

```
helm repo add incubator http://storage.googleapis.com/kubernetes-charts-incubator

helm install incubator/vault --set vault.dev=false --set
vault.config.storage.consul.address="consul:8500",vault.config.storage.consul.path=
"vault"
```

Github page
https://github.com/helm/charts/tree/master/incubator/vault

3.2. Highly Available Vault Installation

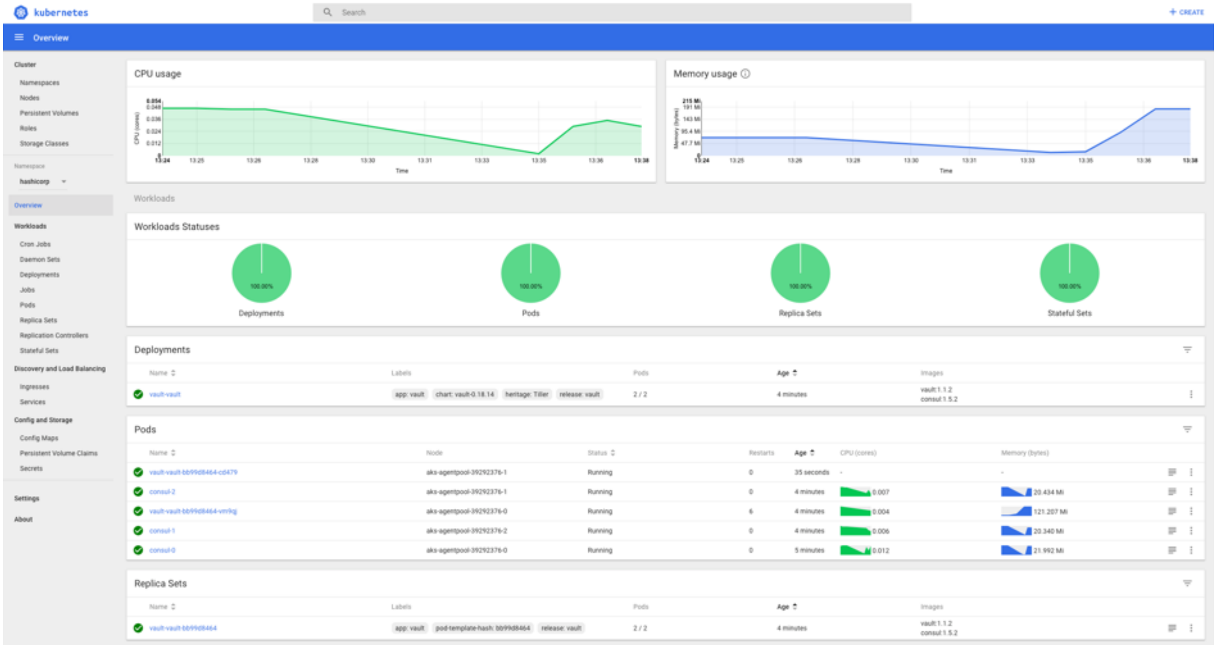
The reference Diagram for HA installation of Vault / Consul is described at the following link:
https://learn.hashicorp.com/vault/operations/ops-vault-ha-consul



In this installation:

- Vault does not communicate directly with Consul but with a local Consul Agent in the same pod
- The Local Consul Agent communicate with the Consul server and are responsible of registering health checks, service discovery and cluster HA failover coordination

3.3. Vault / Consul on Kubernetes Dashboard



Deployments

Name	Labels	Pods	Age	Images
vault-vault	app: vault chart: vault-0.18.14 heritage: Tiller release: vault	2 / 2	4 minutes	vault:1.1.2 consul:1.5.2

Pods

Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
vault-vault-bb99d8464-cd479	aks-agentpool-39292376-1	Running	0	35 seconds	-	-
consul-2	aks-agentpool-39292376-1	Running	0	4 minutes	0.007	20.434 Mi
vault-vault-bb99d8464-vm9qj	aks-agentpool-39292376-0	Running	6	4 minutes	0.004	121.207 Mi
consul-1	aks-agentpool-39292376-2	Running	0	4 minutes	0.006	20.340 Mi
consul-0	aks-agentpool-39292376-0	Running	0	5 minutes	0.012	21.992 Mi

Replica Sets

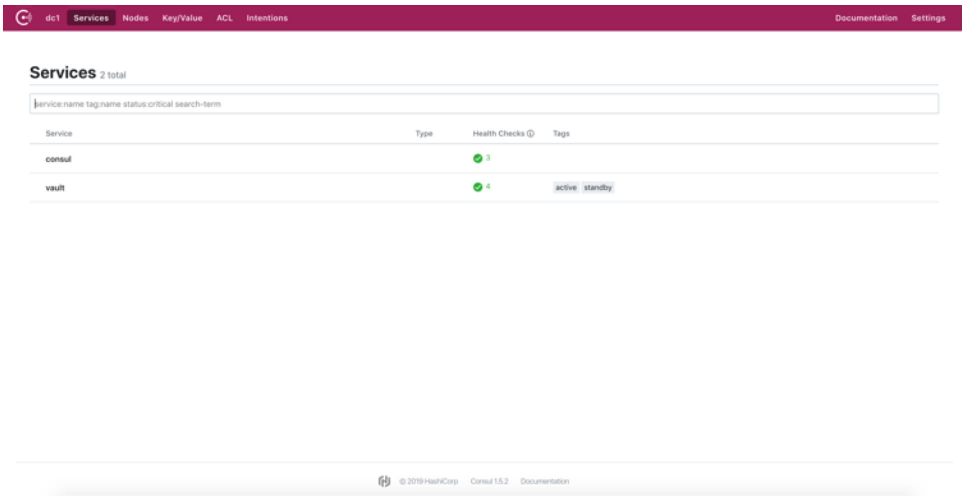
Name	Labels	Pods	Age	Images
vault-vault-bb99d8464	app: vault pod-template-hash: bb99d8464 release: vault	2 / 2	4 minutes	vault:1.1.2 consul:1.5.2

3.4. Check Vault Status in Consul Dashboard

- Enable local port forwarding to a consul pod:

```
$ kubectl -n hashicorp port-forward consul-0 8500:8500
```

- Open URL in browser: http://localhost:8500/ui



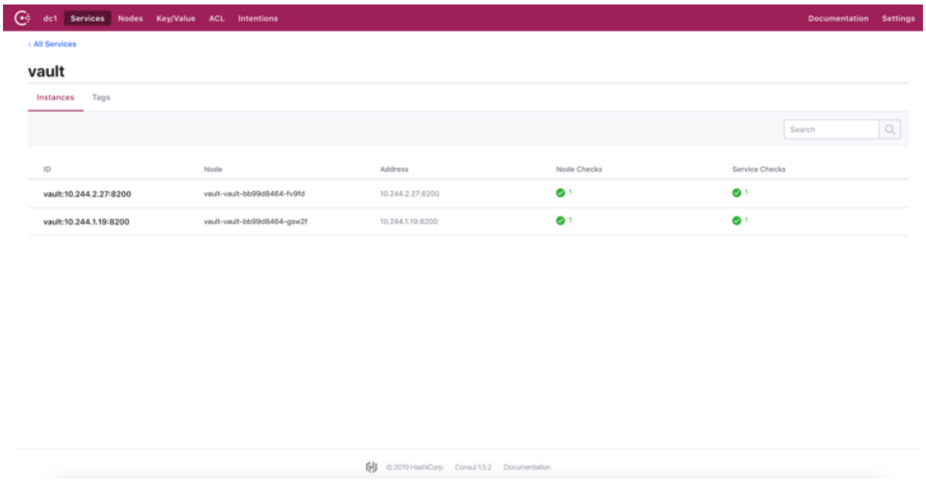
Services 2 total

service:name tag:name status:critical search-term

Service	Type	Health Checks	Tags
consul		✓ 3	
vault		✓ 4	active standby

© 2019 HashiCorp Consul 1.5.2 Documentation

- Click on Vault to check its status: 2 nodes in active state can be seen

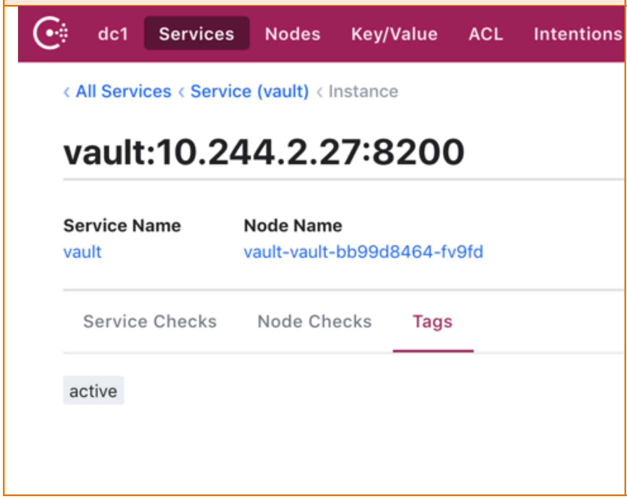
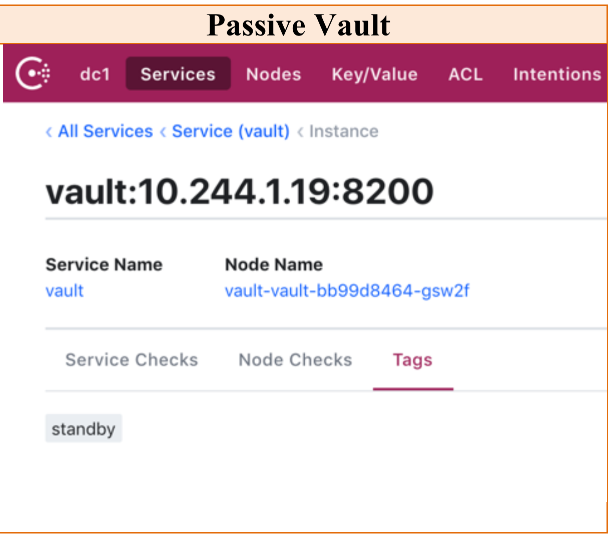


vault

ID	Node	Address	Node Checks	Service Checks
vault:10.244.2.27:8200	vault-vault-bb99d8464-fv9fd	10.244.2.27:8200	✓ 1	✓ 1
vault:10.244.1.19:8200	vault-vault-bb99d8464-gsw2f	10.244.1.19:8200	✓ 1	✓ 1

© 2019 HashiCorp Consul 1.5.2 Documentation

- Click on each node and take a look at its tag attributes:
  - One node is in **active** mode
  - One node is in **standby** mode (passive)

Active Vault	Passive Vault
	
dc1 Services Nodes Key/Value ACL Intentions ‹ All Services ‹ Service (vault) ‹ Instance vault:10.244.2.27:8200 Service Name: vault Node Name: vault-vault-bb99d8464-fv9fd Service Checks Node Checks Tags active	dc1 Services Nodes Key/Value ACL Intentions ‹ All Services ‹ Service (vault) ‹ Instance vault:10.244.1.19:8200 Service Name: vault Node Name: vault-vault-bb99d8464-gsw2f Service Checks Node Checks Tags standby

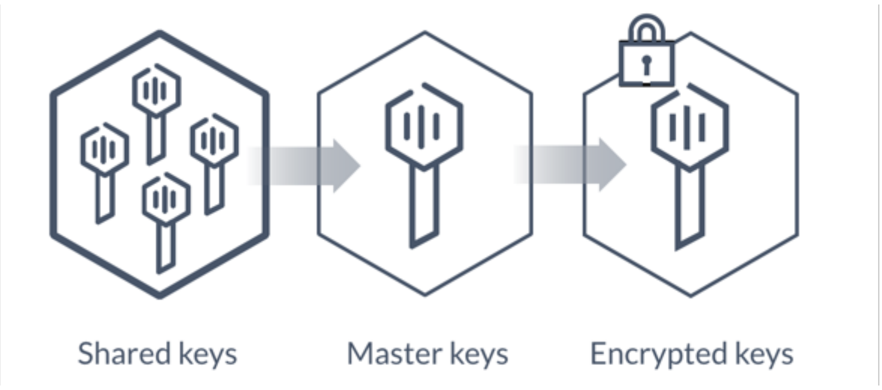
4. Vault Initialization

Vault initialization consists of the following steps:

- Create Vault Master Key and split it into shares to be hold by operators (custodians)
- Unseal Vault
- Create Admin Policy
- Create Provisioner Policy

4.1. Unseal Vault Manually

Vault is configured by default to use Shamir's Secret Sharing to split the master encryption key into shards. A certain threshold of shards is required to reconstruct the master key and unseal Vault data. The shards are added one at a time (in any order) until enough are present to reconstruct the master key and decrypt the data.



- Install vault client binary from official hashicorp page
- Enable port-forwarding between the vault agent running on kubernetes and your local machine

```
$ kubectl port-forward $(kubectl get pod -l app=vault-vault -o
jsonpath='{.items[0].metadata.name}') 8200:8200
```

If vault is installed in a dedicated namespace named **hashicorp** within kubernetes, then call the aforementioned command with -n option to indicate the namespace

```
$ kubectl -n hashicorp port-forward $(kubectl -n hashicorp get pod -l app=vault -o
jsonpath='{.items[0].metadata.name}') 8200:8200
```

Set the following environment variables to be used by vault client locally

```
$ export VAULT_ADDR=http://localhost:8200
$ export VAULT_SKIP_VERIFY=true
```

Initialize the vault cluster by running this command (we don't split the key to make the example easier).

```
$ vault operator init -key-shares=1 -key-threshold=1
```

```
Unseal Key 1: DhzoEw3uKbiyXFmqRiIO7cz8goZp9LbPM2QXVSlwUos=
Initial Root Token: s.4jYP6KL1T7lyrnIzbIsjP61g

Vault initialized with 1 key shares and a key threshold of 1. Please securely distribute the
key shares printed above. When the Vault is re-sealed, restarted, or stopped, you must
supply at least 1 of these keys to unseal it before it can start servicing requests.
Vault does not store the generated master key. Without at least 1 key 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.
```

Unseal the vault cluster using the generated unseal key

```
$ vault operator unseal


Unseal Key (will be hidden):
Handling connection for 8200
Key                Value
---                -----
Seal Type          shamir
Initialized        true
Sealed             false
Total Shares       1
Threshold          1
Version            1.1.2
Cluster Name       vault-cluster-32099219
Cluster ID         5ad76e1b-0ef3-8e97-bcbd-cd67d199b559
HA Enabled         true
HA Cluster         https://10.244.1.26:8201
HA Mode            active
```

4.2. Create Vault Policies

4.2.1. Login with Root Token

Authenticate to vault using the root token

```
$ vault login <your-generated-root-token>

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.4jYP6KL1T7lyrnIzbIsjP61g
token_accessor          G7E9m9Y1tqEZXC42dl4c2S6a
token_duration          ∞
token_renewable         false
token_policies          ["root"]
identity_policies       []
policies                ["root"]
```

4.2.2. Create Admin Policy

Create admin policy file admin-policy.hcl. Example hereafter:

```
# Manage auth methods broadly across Vault
path "auth/*"
{
capabilities = ["create", "read", "update", "delete", "list", "sudo"]
}

# Create, update, and delete auth methods
path "sys/auth/*"
{
capabilities = ["create", "update", "delete", "sudo"]
}
```

```
# List auth methods
path "sys/auth"
{
capabilities = ["read"]
}

# List existing policies
path "sys/policies/acl"
{
capabilities = ["read"]
}

# Create and manage ACL policies
path "sys/policies/acl/*"
{
capabilities = ["create", "read", "update", "delete", "list", "sudo"]
}

# List, create, update, and delete key/value secrets
path "secret/*"
{
capabilities = ["create", "read", "update", "delete", "list", "sudo"]
}

# Manage secret engines
path "sys/mounts/*"
{
capabilities = ["create", "read", "update", "delete", "list", "sudo"]
}

# List existing secret engines.
path "sys/mounts"
{
capabilities = ["read"]
}

# Read health checks
path "sys/health"
{
capabilities = ["read", "sudo"]
}
```

Create admin policy by running the following command

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

4.2.3. Create Provisioner Policy

Create provisioner policy HCL file provisioner-policy.hcl. Example:

```
# Manage auth methods broadly across Vault
path "auth/*"
{
capabilities = ["create", "read", "update", "delete", "list", "sudo"]
}

# Create, update, and delete auth methods
path "sys/auth/*"
{
capabilities = ["create", "update", "delete", "sudo"]
}

# List auth methods
path "sys/auth"
{
capabilities = ["read"]
}

# List existing policies
path "sys/policies/acl"
{
```

```
capabilities = ["read"]
}

# Create and manage ACL policies via API & UI
path "sys/policies/acl/*"
{
capabilities = ["create", "read", "update", "delete", "list", "sudo"]
}

# List, create, update, and delete key/value secrets
path "secret/*"
{
capabilities = ["create", "read", "update", "delete", "list"]
}
```

Create provisioner policy

```
$ vault policy write provisioner provisioner-policy.hcl
```

4.2.4. List Policies

Run the following command to list policies

```
$ vault policy list

admin
default
provisioner
root
```

4.2.5. Enable KV Secret Engine

Get a token for admin policy

```
vault token create -policy=admin

Key                  Value
---                  -----
token                s.HDoPEp5aKJs7HsdcLB46xy1Z
token_accessor       etUitQFBLxdGwbiisOgETLiB
token_duration       768h
token_renewable      true
token_policies       ["admin" "default"]
identity_policies    []
policies             ["admin" "default"]
```

Login with admin token

```
vault login s.OQInhur8aVH7r6RRk814RZxP

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.HDoPEp5aKJs7HsdcLB46xy1Z
token_accessor       etUitQFBLxdGwbiisOgETLiB
token_duration       767h59m53s
token_renewable      true
token_policies       ["admin" "default"]
identity_policies    []
policies             ["admin" "default"]
```

Enable kv secret engine for path secret

```
$ vault secrets enable -path=secret kv
```

4.3. Write / Read Secrets

Connect with provisioner token as it is the one responsible for creating secrets

```
$ vault token create -policy=provisioner

Key                  Value
---                  -----
token                s.bKpKQG0ajvaw7ya1LaVyosRo
token_accessor       wbhRSxOFL8l8YcS06r3C0UvU
token_duration       768h
token_renewable      true
token_policies       ["default" "provisioner"]
identity_policies    []
policies             ["default" "provisioner"]

$ vault login s.bKpKQG0ajvaw7ya1LaVyosRo
```

Write a secret to vault. Let's name it car because it contains the parameters of car app. 2 commands can be used in order to write the secret

```
$ vault write secret/car var1=1 var2=2 var3=3
or
$ vault kv put secret/car var1=1 var2=2 var3=3

Success! Data written to: secret/car
```

Read the data from the secret

```
$ vault read secret/car
or
$ vault kv get secret/car

==== Data ====
Key     Value
---     -----
var1    1
var2    2
var3    3
```

5. Auto-Unseal using Azure Key Vault

Unsealing Vault manually as we have seen in the previous paragraph required manual intervention of security teams or admins. In production environment, it is mandatory to make sure this procedure is done automatically. Here comes the auto-unseal feature of Vault.

Starting from Vault 1.0, Hashicorp has open sourced the auto-unseal feature that previously required Vault Enterprise Pro. It is possible now to opt-in for automatic unsealing via your trusted cloud provider (for example Azure).

Source:
https://www.hashicorp.com/blog/vault-learning-resources-auto-unseal-agent-kubernetes

Auto-Unseal using Azure Key Vault:
https://learn.hashicorp.com/vault/operations/autounseal-azure-keyvault

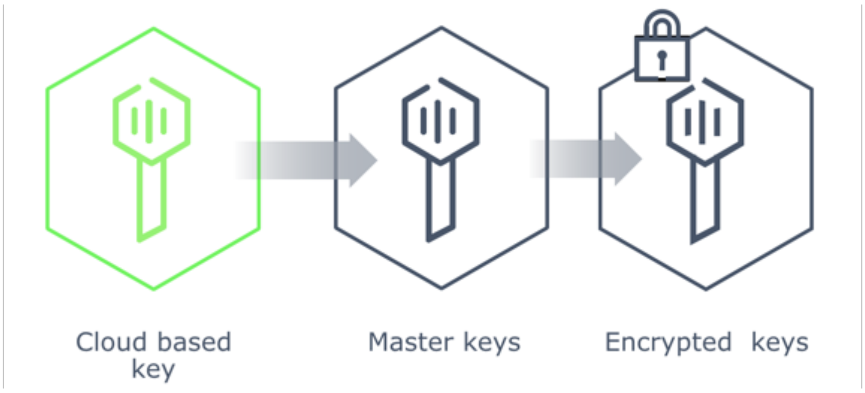
In this paragraph, we explain how to configure Vault Auto-Unseal feature based on Azure Vault.

5.1. Auto-Unseal Principle

Auto unseal was developed to aid in reducing the operational complexity of unsealing Vault while keeping the master key secure. This feature delegates the responsibility of securing the master key from *operators* to a trusted *device* or *service*.

Instead of only constructing the key in memory, **the master key is encrypted with a cloud-based Key Management System (KMS) or an on-premises Hardware Security Module (HSM) and then stored in the storage backend allowing Vault to decrypt the master key at startup and unseal automatically**.

→ This eliminates the need for multiple operators or agents to provide parts of the shared key in either a manual or custom-built automated process.

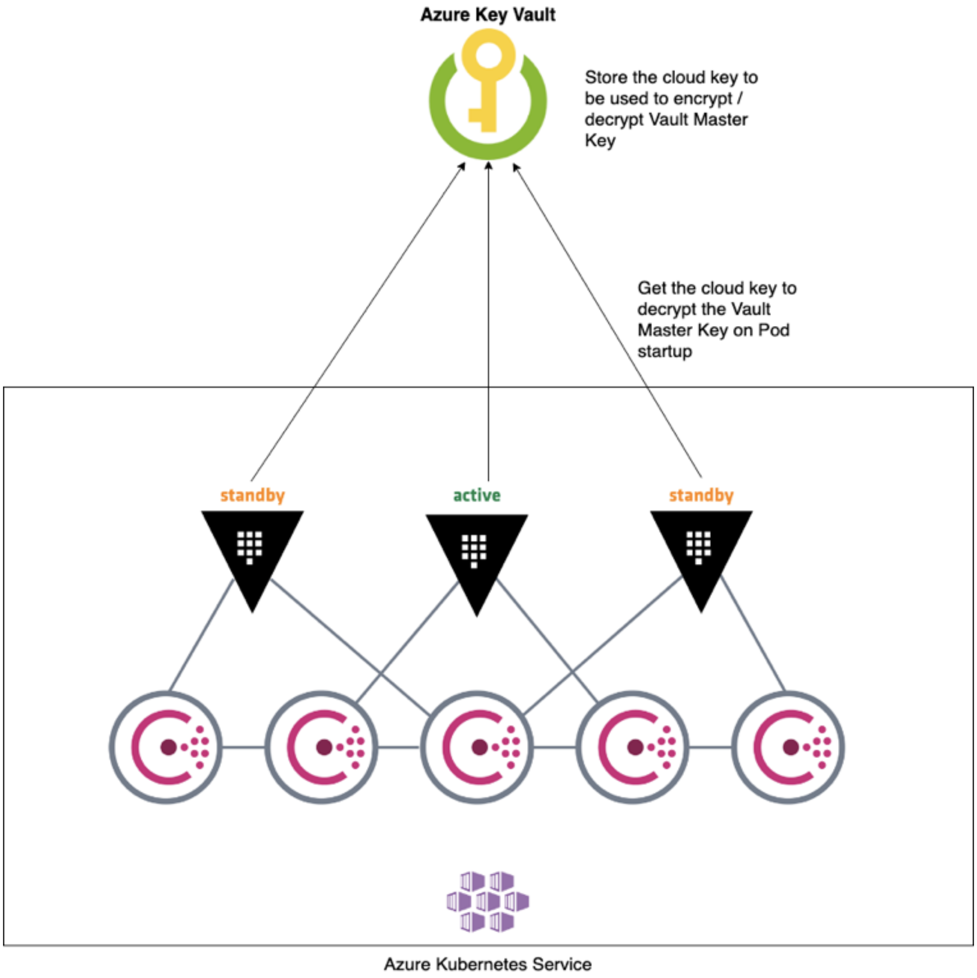


Auto unseal using a cloud-based KMS is available in the open source version as of Vault 1.0. **Auto unseal with a HSM remains a Vault Enterprise feature**.

When using auto unseal, there are certain operations in Vault that still require a quorum of users to perform, such as generating a root token. During the initialization process a set of Shamir keys are generated that are called *Recovery Keys* and are used for these operations.

Source:
https://www.hashicorp.com/blog/enabling-cloud-based-auto-unseal-in-vault-open-source

5.2. Vault Cluster Topology


Azure Key Vault
Store the cloud key to be used to encrypt / decrypt Vault Master Key
Get the cloud key to decrypt the Vault Master Key on Pod startup
standby
active
standby
Azure Kubernetes Service

- Azure Key Vault stores the encryption **Cloud Key** used to encrypt **Vault Master Key**
- **Cloud Key** is created using Azure CLI or Terraform (example is provided afterwards)
- **Vault Master Key** is generated during Vault installation and stored encrypted on Vault backend (Consul)
- Each Vault Pod requests the **Cloud Key** on startup, and use it to decrypt the **Master Key**, and then decrypt the data encrypted in Vault backend.

5.3. Create Azure Key Vault & Cloud Key

We will use **Terraform** tool in order to provision and Azure Key Vault and create a Cloud Key that will be used in order to encrypt Vault Master Key. Below is the terraform script:

5.3.1. Setup Azure Key Vault

```
resource "random_id" "keyvault" {
  byte_length = 4
}

resource "azurerm_key_vault" "vault" {
  name                        = "vault-${random_id.keyvault.hex}"
  location                    = "${var.location}"
  resource_group_name         = "${azurerm_resource_group.k8s.name}"
  enabled_for_deployment      = true
  enabled_for_disk_encryption = true
  tenant_id                   = "${var.tenant_id}"

  sku_name = "standard"

  tags = {
    environment = "${var.environment}"
  }

  access_policy {
    tenant_id = "${var.tenant_id}"

    object_id = "${var.sp_object_id}"
    #object_id = "${data.azurerm_client_config.current.service_principal_object_id}"

    key_permissions = [
      "get",
      "list",
      "create",
      "delete",
      "update",
      "wrapKey",
      "unwrapKey",
    ]
  }

  network_acls {
    default_action = "Allow"
    bypass         = "AzureServices"
  }
}
```

5.3.2. Setup Cloud Key

```
resource "azurerm_key_vault_key" "generated" {
  name         = "${var.az_vault_key_name}"
  key_vault_id = "${azurerm_key_vault.vault.id}"
  key_type     = "RSA"
  key_size     = 2048

  key_opts = [
    "decrypt",
    "encrypt",
    "sign",
    "unwrapKey",
    "verify",
    "wrapKey",
  ]
}
```

5.4. Use HSM for Storing Cloud Key

The use of HSM for storing the Cloud Key requires **Hashicorp Vault Enterprise**. The open source version of Hashicorp Vault cannot be used with HSM.

https://www.vaultproject.io/docs/enterprise/hsm/index.html

5.5. Install Hashicorp Vault with Azure Key Vault Options

We use Terraform & Helm Provider to install Consul backend and Vault.

5.5.1. Install Consul Server

Nothing changes regarding Consul installation.

```
resource "random_string" "consul-gossip-password" {
  length = 16
  special = true
}

#Install Consul
resource "helm_release" "consul" {

  name      = "consul"
  chart     = "stable/consul"
  namespace = "${kubernetes_namespace.hashicorp.metadata.0.name}"

  timeout   = 300

  set {
    name  = "image.pullPolicy"
    value = "Always"
  }
  set {
    name  = "replicaCount"
    value = 1
  }
  set {
    name = "GossipKey"
    value = "${random_string.consul-gossip-password.result}"
  }

  depends_on=["local_file.kube_config"]
}
```

5.5.2. Install Vault Server

We add the following options to instruct vault to use azure key vault auto-unseal feature:

Option	Description
vault.config.seal.azurekeyvault.tenant_id	Azure Subscription Tenant ID
vault.config.seal.azurekeyvault.client_id	Azure App ID (SP ID)
vault.config.seal.azurekeyvault.client_secret	Azure App Secret (SP Secret)
vault.config.seal.azurekeyvault.vault_name	Azure Key Vault Name
vault.config.seal.azurekeyvault.key_name	Azure Key (Cloud Key) to be used to encrypt Vault Master Key

→ how vault stores azure service principal credentials?

```
resource "helm_release" "vault" {
  name       = "vault"
  chart      = "incubator/vault"
  repository = "${data.helm_repository.incubator.metadata.0.name}"
  namespace = "${kubernetes_namespace.hashicorp.metadata.0.name}"
  version = "0.18.14"
  timeout    = 300

  set {
    name  = "image.pullPolicy"
    value = "Always"
  }
  set {
    name  = "replicaCount"
    value = 1
  }
  set {
    name  = "vault.dev"
    value = false
  }
  set {
    name  = "vault.config.storage.consul.address"
    #value = "consul:8500"
    value = "127.0.0.1:8500"
  }
  set {
    name  = "vault.config.storage.consul.path"
    value = "vault"
  }
  set {
    name  = "consulAgent.join"
    value = "consul"
  }
  set {
    name = "consulAgent.gossipKeySecretName"
    value = "consul-gossip-key"
  }
  set {
    name = "consulAgent.tag"
    value = "1.5.2"
  }
  set {
    name  = "vault.config.storage.consul.scheme"
    value = "http"
  }

  ## Enable auto-unseal using Azure Key Vault
  set {
    name = "vault.config.seal.azurekeyvault.tenant_id"
    value = "${var.tenant_id}"
  }
  set {
    name = "vault.config.seal.azurekeyvault.client_id"
    value = "${var.client_id}"
  }
  set {
    name = "vault.config.seal.azurekeyvault.client_secret"
    value = "${var.client_secret}"
  }
  set {
    name = "vault.config.seal.azurekeyvault.vault_name"
    value = "${azurerm_key_vault.vault.name}"
  }
  set {
    name = "vault.config.seal.azurekeyvault.key_name"
    value = "${var.az_vault_key_name}"
  }
   # Enable Vault UI in production
  set {
    name = "vault.config.ui"
    value = true
  }
  set {
    name = "vault.config.listener.tcp.tls_disable"
    value = 1
  }

  depends_on=["helm_release.vault"]
```

```
}
```

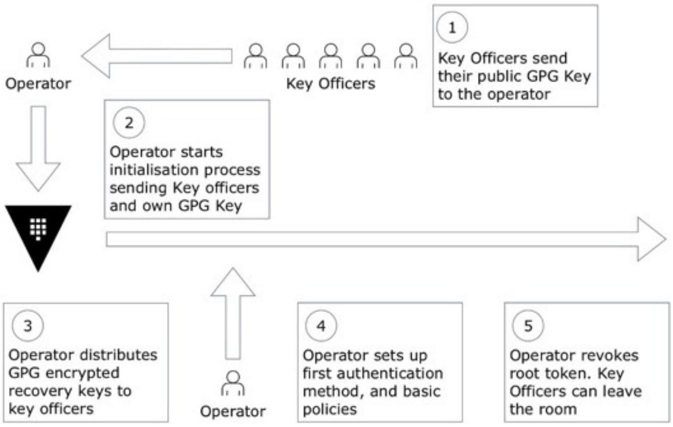
5.6. Initialize Auto-Unsealed Vault

5.6.1. Initialization Ceremony

Refer to the following documentation:
https://www.hashicorp.com/resources/adopting-hashicorp-vault

While the auto-unseal process is automated using the provider's (azure) KMS system (key vault and cloud key), the initialization of Vault is **a manual process and requires the presence of key holders**. This process must be executed as follows:

1. The Operator starts the Vault daemon and the initialization process, providing the public GPG keys from the Keyholders, and the Operators own public GPG key for the root token.
2. Vault will return the GPG encrypted recovery keys, which should be distributed among the keyholders.
3. The operator uses the root token for loading the initial policy and configuring the first authentication backend, traditionally Active Directory or LDAP, as well as the Audit backend.
4. The operator validates they can log into Vault with their directory account, and can add further policy.
5. The operator revokes the root token. The Keyholders can now leave the room with the assurance that no one person has full and unaudited access to Vault.


Operator
Key Officers
1
Key Officers send their public GPG Key to the operator
2
Operator starts initialisation process sending Key officers and own GPG Key
3
Operator distributes GPG encrypted recovery keys to key officers
Operator
4
Operator sets up first authentication method, and basic policies
5
Operator revokes root token. Key Officers can leave the room

5.6.2. Initialize Auto-Unsealed Vault using Command Line

Vault requires a first initialization operation that returns the following:

- Recovery Key Shares
- Initial Root Token

Hereafter the steps required in order to initialize Vault:

Create port forwarding from your local machine to vault pod:

```
$ kubectl -n hashicorp port-forward $(kubectl -n hashicorp get pod -l app=vault -o
jsonpath='{.items[0].metadata.name}') 8200:8200
```

Export required environment variables:

```
$ export VAULT_ADDR=http://localhost:8200
$ export VAULT_SKIP_VERIFY=true
```

Run the vault status command to get the status of vault server. The command would return the following output (note the recovery seal type **azurekeyvault**)

```
$ vault status

Key                      Value
---                      -----
Recovery Seal Type       azurekeyvault
Initialized              false
Sealed                   true
Total Recovery Shares    0
Threshold                0
Unseal Progress          0/0
Unseal Nonce             n/a
Version                  n/a
HA Enabled               true
```

Run the init command with number of key shares and threshold (we choose 5 key shares which means the master key will be split in 5 shares, and we choose 3 as threshold which means 3 shares are enough to recover the master key).

```
$ vault operator init -key-shares=5 -key-threshold=3

Recovery Key 1: 1Q6JWdYAbrch5rxf1QTP61092irVbDEOXqYP88sju0MV
Recovery Key 2: N9ly3LiP5n+adQNYNt2jh/utz4u/RFHRLWo79kWxkMGj
Recovery Key 3: yzadGVdmMPldF5cObkq/V4e58ObZtEY2nqkIiplgFkO4
Recovery Key 4: mHMUlh8aIBeheYK/e22EqVITvJyTM6C1HoCx/jmkVStD
Recovery Key 5: stiXLIc6vrJHhDCII3rZZyoMP0JpjllzqNQN2mEnCnHy

Initial Root Token: s.S2A1q9R9N4f3JAXLrpwAteDo

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.
```

The Initial Root Token can then be used to provision the ACLs as in paragraph 4.2.

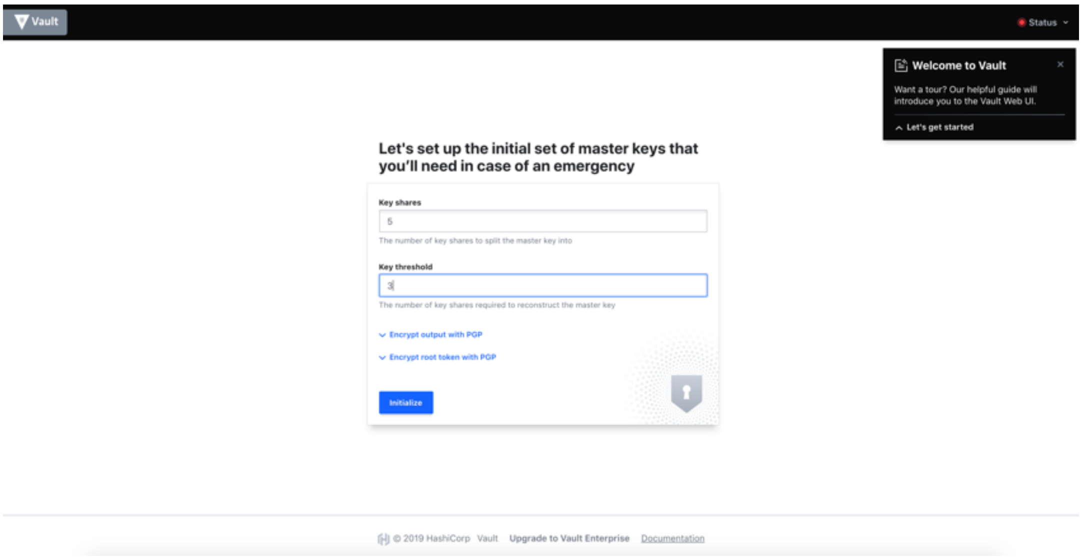
→ Refer to documentation for more initialization options and encryption of recovery keys and root token using PGP:
https://www.vaultproject.io/docs/commands/operator/init.html

5.6.3. Initialize & Configure Auto-Unsealed Vault using Web UI

Enable port forwarding to vault's pods using **kubectl port-forward** command and define required environment variables:

```
$ kubectl -n hashicorp port-forward $(kubectl -n hashicorp get pod -l app=vault -
o jsonpath='{.items[0].metadata.name}') 8200:8200

$ export VAULT_ADDR=http://localhost:8200
$ export VAULT_SKIP_VERIFY=true
```

Open the following URL in browser: http://localhost:8200/ui/
Notice that vault status is red (uninitialized) and you are invited to initialize vault.


Vault
Status
Welcome to Vault
Want a tour? Our helpful guide will introduce you to the Vault Web UI.
Let's get started

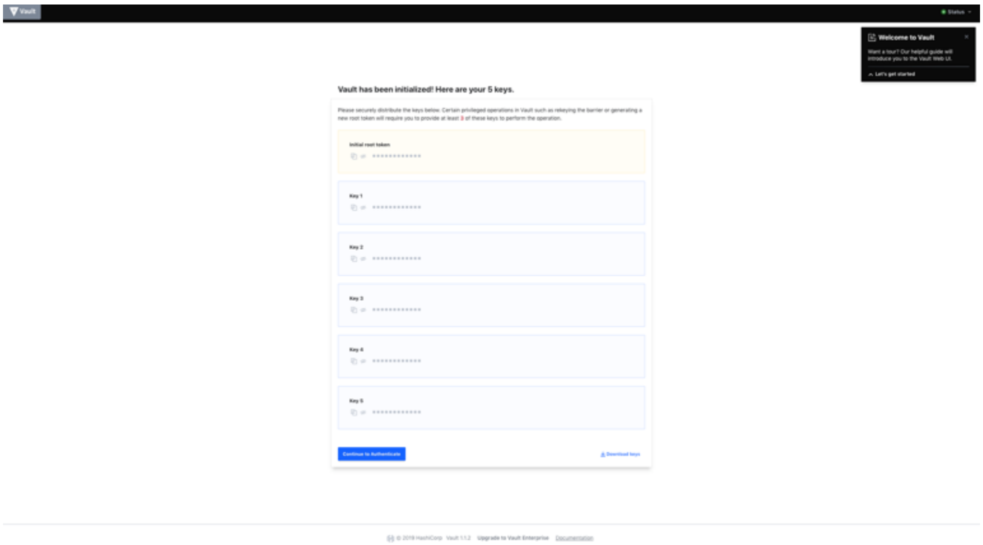
Let's set up the initial set of master keys that you'll need in case of an emergency

Key shares
5
The number of key shares to split the master key into

Key threshold
3
The number of key shares required to reconstruct the master key

Encrypt output with PGP
Encrypt root token with PGP
Initialize
© 2019 HashiCorp Vault Upgrade to Vault Enterprise Documentation

Specify the number of key shares and threshold as before (5 shares and threshold 3) and press the initialize button.


Vault
Status
Welcome to Vault
Want a tour? Our helpful guide will introduce you to the Vault Web UI.
Let's get started

Vault has been initialized! Here are your 5 keys.

Please securely distribute the keys below. Certain privileged operations in Vault such as rekeying the barrier or generating a new root token will require you to provide at least 3 of these keys to perform the operation.

Initial root token
****

Key 1
****

Key 2
****

Key 3
****

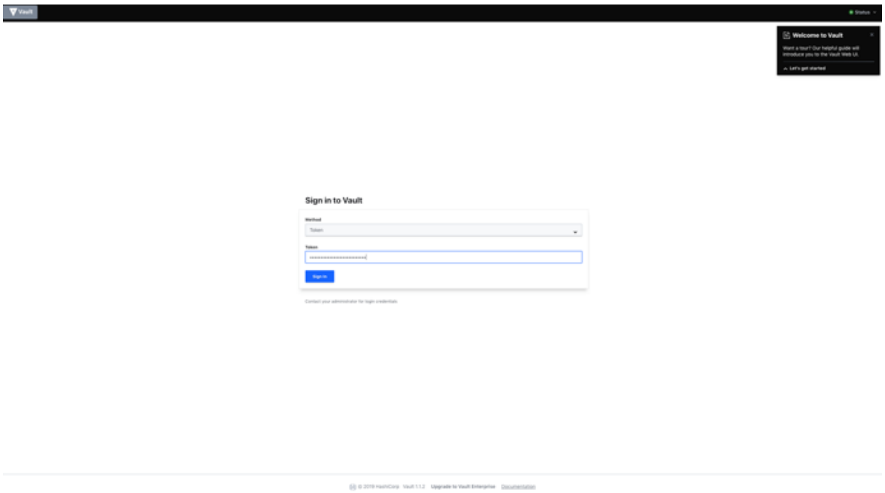
Key 4
****

Key 5
****

Continue to Authenticate
Download keys
© 2019 HashiCorp Vault 1.1.2 Upgrade to Vault Enterprise Documentation

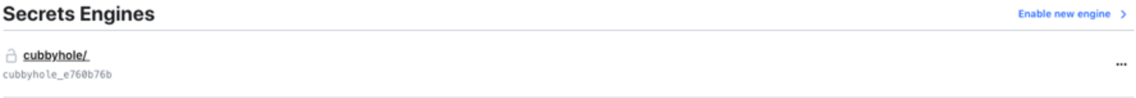
Note that vault is now initialized, and status is in green (picture below)

- Download the recovery key shares and save them to a secure place
- Make the initial root token visible and use it to authenticate the first time and configure vault

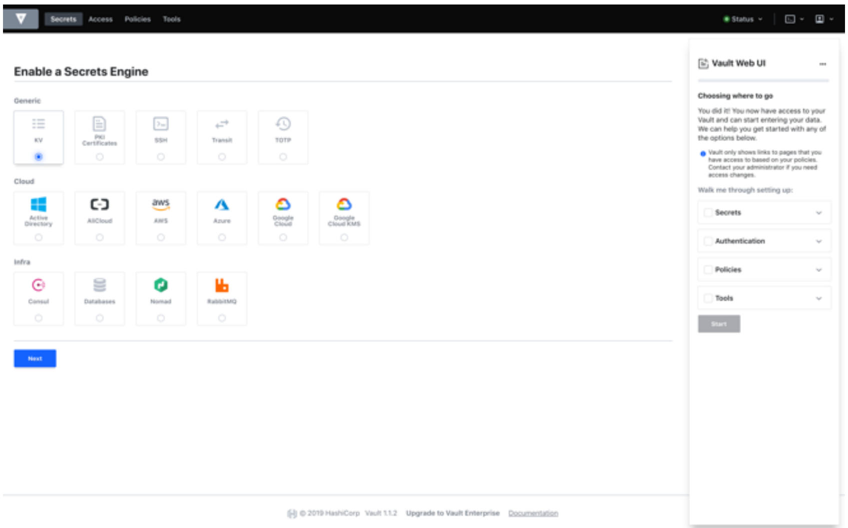


Vault
Status
Welcome to Vault
Want a tour? Our helpful guide will introduce you to the Vault Web UI.
Let's get started
Sign in to Vault
Method
Token
Token
Sign In
Contact your administrator for login credentials
© 2019 HashiCorp Vault 1.1.2 Upgrade to Vault Enterprise Documentation

Once we are authenticated, we are presented with the enabled secret engines. Press **Enable new engine** link to enable the KV secret engine that we are going to use to store application secrets.



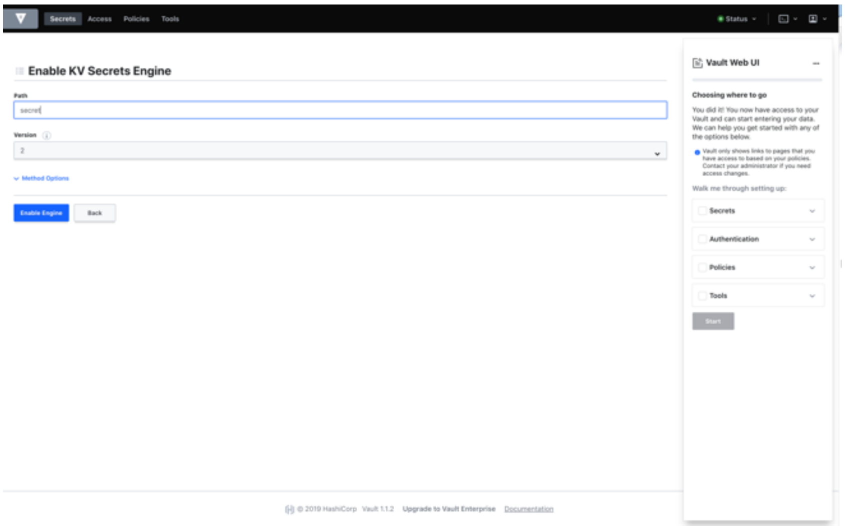
Select the KV engine and then press Next



You did it! You now have access to your Vault and can start entering your data. We can help you get started with any of the options below.

Vault only shows links to pages that you have access to based on your policies. Contact your administrator if you need access changes.

Name the path of the KV engine **“secret”** as we did in the command line and then press **Enable Engine**



Enable KV Secrets Engine

Path: secret

Version: 2

Method Options

Vault Web UI

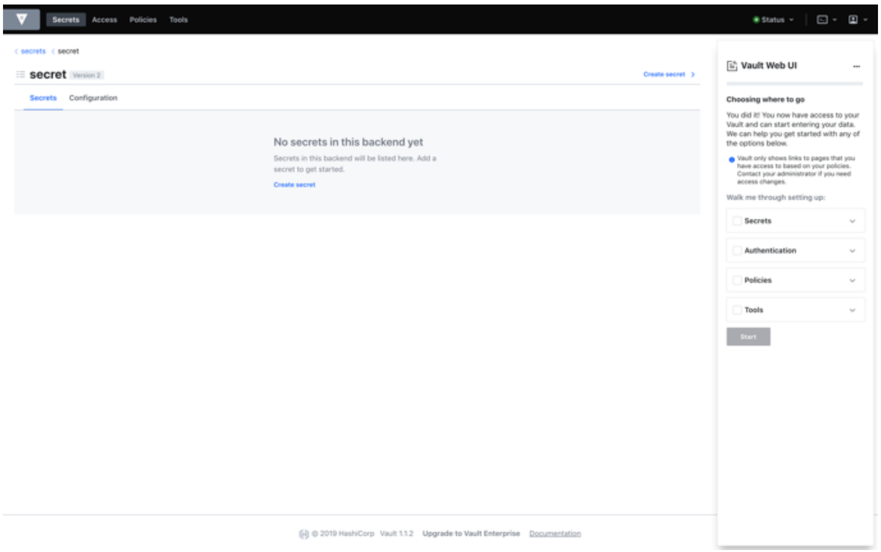
Choosing where to go

You did it! You now have access to your Vault and can start entering your data. We can help you get started with any of the options below.

Vault only shows links to pages that you have access to based on your policies. Contact your administrator if you need access changes.

Walk me through setting up:

The secret KV engine is now created and we can create secrets for applications



No secrets in this backend yet

Secrets in this backend will be listed here. Add a secret to get started.

Vault Web UI

Choosing where to go

You did it! You now have access to your Vault and can start entering your data. We can help you get started with any of the options below.

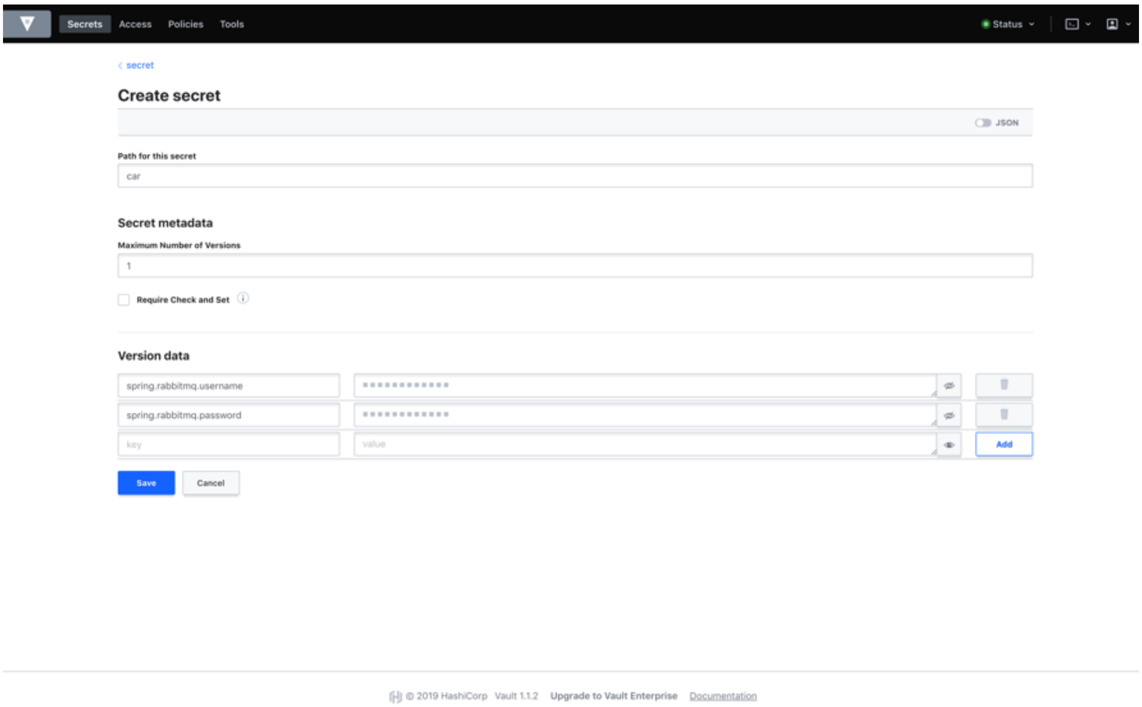
Vault only shows links to pages that you have access to based on your policies. Contact your administrator if you need access changes.

Walk me through setting up:

Create a secret named **car** for our **car** application and create 2 properties in this secret:

- spring.rabbitmq.username
- spring.rabbitmq.password

Set the maximum number of versions to 1 to have only one version of this secret, and then hit the **Save** button to save the new Secret.


Secrets
Access
Policies
Tools
Status
secret
JSON
Save
Cancel
Add
© 2019 HashiCorp Vault 1.1.2 Upgrade to Vault Enterprise Documentation

Create secret

Path for this secret: car

Secret metadata

Maximum Number of Versions: 1

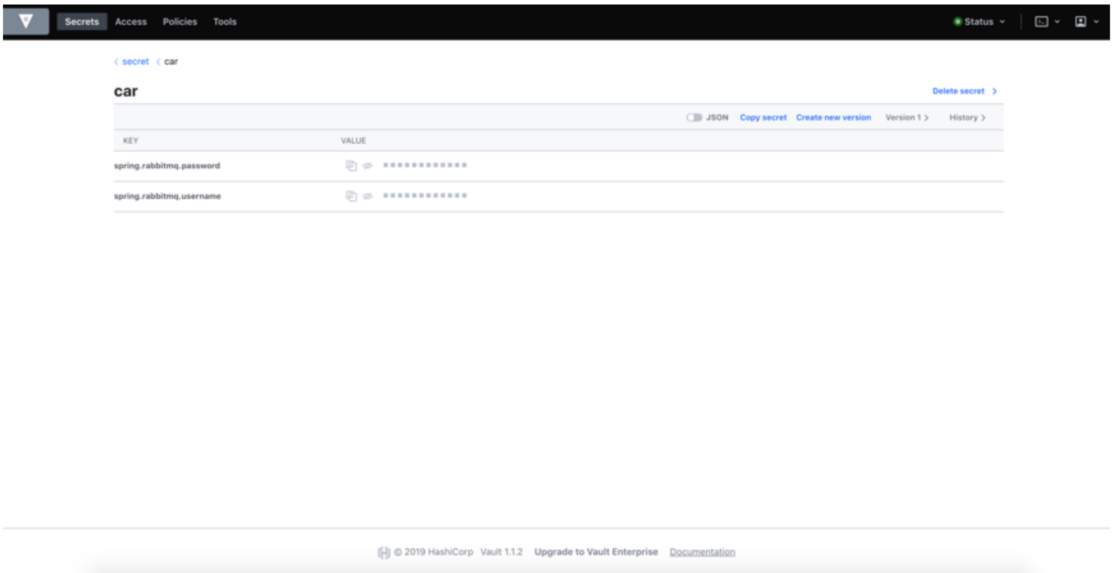
□ Require Check and Set

Version data

spring.rabbitmq.username ************

spring.rabbitmq.password ************

The secret with path **secret/car** is now created



car

KEY	VALUE
spring.rabbitmq.password	************
spring.rabbitmq.username	************

5.7. Vault Root Token Recovery

Refer to this link:
https://www.vaultproject.io/docs/commands/operator/generate-root.html

6. Vault Production Hardening

Hashicorp Vault production hardening recommendations can be found here:
https://www.vaultproject.io/guides/operations/production

7. Vault Spring Boot Integration

Spring has Spring Cloud Vault framework that facilitates Vault integration into Spring Boot applications.

Vault API Reference can be found at this link:
https://cloud.spring.io/spring-cloud-vault/reference/html/

7.1. Application Dependencies

Update the **pom.xml** file with the following dependency in order to enable Spring Cloud Vault Secret Management.

```
<!-- Spring Cloud Vault -->
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-vault-config</artifactId>
	<version>2.1.2.RELEASE</version>
</dependency>
```

7.2. Spring Cloud Vault Configuration

Spring Cloud Vault must be enabled / configured in **bootstrap.properties** file so that Spring Boot can use it to load its configuration. This file must provide Spring Boot app with the following information:

- Vault URI
- Vault Authentication Method
- Scheme (http / https)
- Information required to authenticate with Vault

This is an example **bootstrap.properties** configuration file that uses Vault Token Authentication method.

```
# Application Name
spring.application.name=car

# Vault Parameters
spring.cloud.vault.enabled = true
spring.cloud.vault.uri=http://localhost:8200
spring.cloud.vault.token=s.RUV2kChBNXwXwcwV61nNdnXM
spring.cloud.vault.scheme=http
spring.cloud.vault.kv.enabled=true
```

7.3. Application K/V Secrets Store in Vault

Spring Cloud Vault allows Spring Boot Application to load its secrets from a K/V store that has the same name of the application.

For example, my **car** application will load its secrets from **secret/car** secret store.

The application name must be specified using **spring.application.name** property in **application.properties** file.

7.4. Access Vault K/V Secrets from Spring Boot App

Secrets are stored in Vault as Key / Value pairs. These K/V secrets can be retrieved like any other application setting stored in **application.properties**. As an example, load AMQP connection credentials from Vault:

```
@Value("${spring.rabbitmq.username}")
private String username;

@Value("${spring.rabbitmq.password}")
private String password;

@Bean
public ConnectionFactory connectionFactory() {
    CachingConnectionFactory connectionFactory = new CachingConnectionFactory(hostname);
    connectionFactory.setUsername(username);
    connectionFactory.setPassword(password);
    connectionFactory.setPublisherConfirms(true);
    return connectionFactory;
}
```

8. Kubernetes Authentication Method

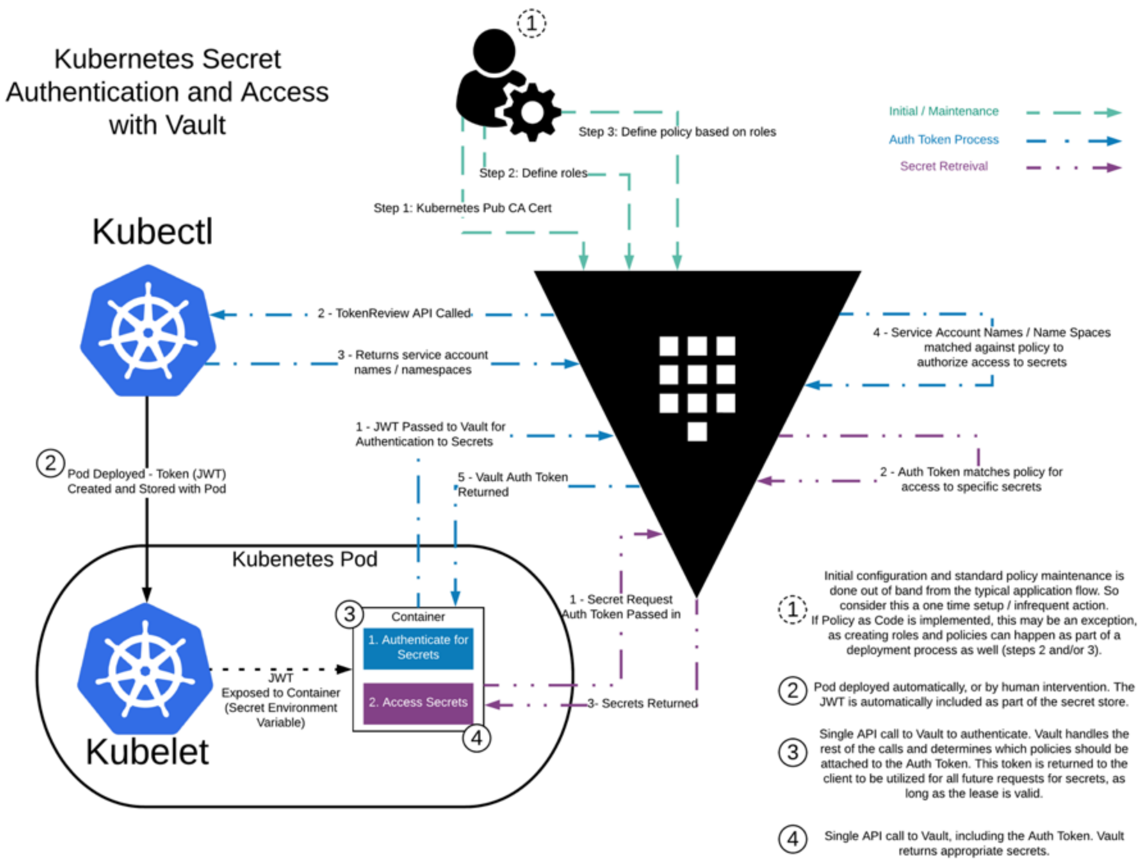
Kubernetes authentication mechanism (since Vault 0.8.3) allows to authenticate with Vault using a Kubernetes Service Account Token. The authentication is role based and the role is bound to a service account name and a namespace.

Refer to the following Spring / Pivotal Documentation:
https://cloud.spring.io/spring-cloud-vault/reference/html/#vault.config.authentication.kubernetes

Refer also to Hashicorp documentation to understand how to configure Vault to use Kubernetes as an authentication method.
https://learn.hashicorp.com/vault/identity-access-management/vault-agent-k8s

8.1. Kubernetes Authentication Principle

The following diagram explains how Kubernetes-based authentication works with Hashicorp Vault.


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
3
Container
1. Authenticate for Secrets
2. Access Secrets
4
JWT Exposed to Container (Secret Environment Variable)
Kubelet
1 - Secret Request Auth Token Passed in
3- Secrets Returned
1
2
3
4

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).

Pod deployed automatically, or by human intervention. The JWT is automatically included as part of the secret store.

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.

Single API call to Vault, including the Auth Token. Vault returns appropriate secrets.

- Admin defines roles and policies in Vault
- Each Kubernetes application has its own service account (to be defined in its helm chart)
- Kubernetes auto-mounts the service account when it starts an application Pod and generate a JWT token and give it to the pod

- The pod gives its JWT token to Vault to get an authentication token. Vault review this request and returns the authentication token matching the defined policies
- The pod uses the authentication token to get access to its secrets

→ Using this method, we do not need to set Vault token in **bootstrap.properties.** The pod will use Kubernetes service account token to retrieve a Vault Token.

8.2. Cluster Role Binding for Vault Service Account

Vault must be granted access to the token review api for Kubernetes.
This can be done by:

- Creating a service account for Vault (name it for example **vault-auth**)
- Creating a cluster role binding for this service account to give it access to the token review api. Below is the yaml configuration file named vault-auth-service-account.yaml to be created in order to create the role binding:

```
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
```

- This binding can be applied using the command:

```
$ kubectl apply -f vault-auth-service-account.yaml
```

8.3. Enable Kubernetes Auth Method in Vault

8.3.1. Activation in CLI

Kubernetes auth method can be enabled in Vault using this command:

```
$ vault auth enable kubernetes
```

In order to configure Vault with Kubernetes information, we need first to export this information as environment variables. Source the bash script below:

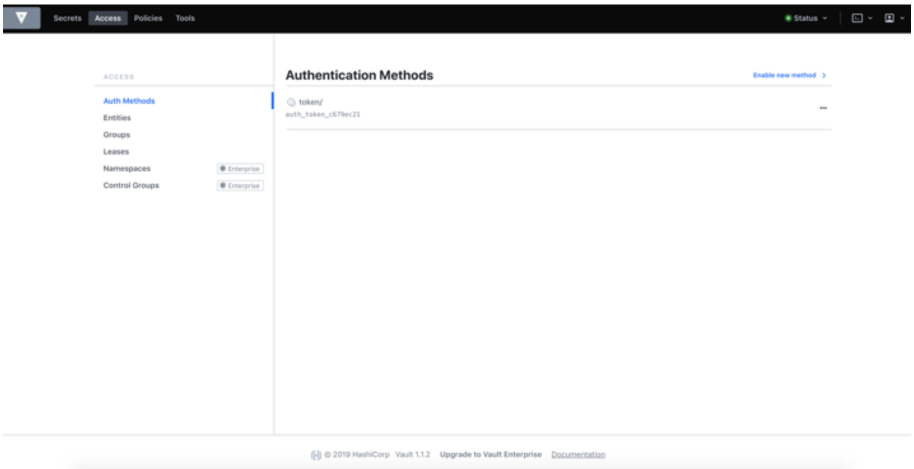
```
# Set VAULT_SA_NAME to vault service account name
export VAULT_SA_NAME=$(kubectl get sa vault-auth -o jsonpath="{.secrets[*]['name']}")

# Set SA_JWT_TOKEN value to the service account JWT used to access the TokenReview API
export SA_JWT_TOKEN=$(kubectl get secret $VAULT_SA_NAME -o jsonpath="{.data.token}" | base64
--decode; echo)

# Set SA_CA_CRT to the PEM encoded CA cert used to talk to Kubernetes API
export SA_CA_CRT=$(kubectl get secret $VAULT_SA_NAME -o jsonpath="{.data['ca\.crt']}" |
base64 --decode; echo)

# Set K8S_HOST to minikube IP address
export K8S_HOST=$(terraform output host)
```

Then run the following command to apply the configuration to Vault:

```
$ vault write auth/kubernetes/config \
        token_reviewer_jwt="$SA_JWT_TOKEN" \
        kubernetes_host="$K8S_HOST" \
        kubernetes_ca_cert="$SA_CA_CRT"
```

8.3.2. Web Activation

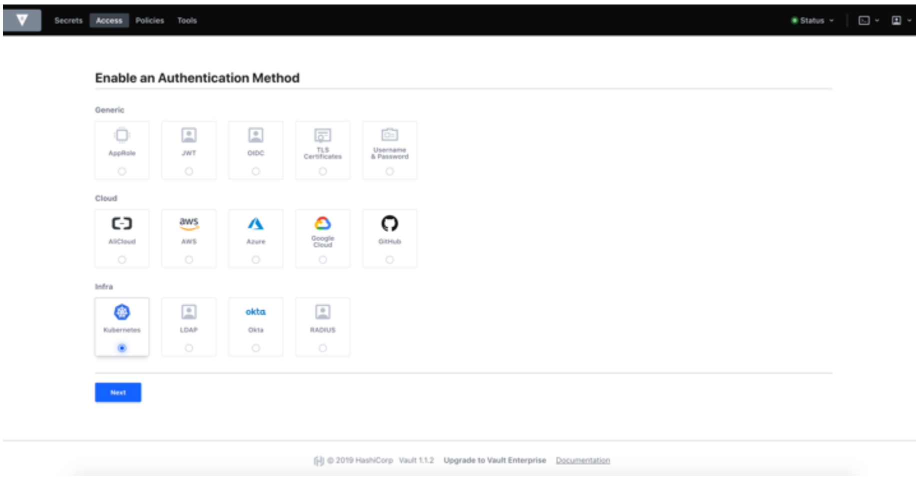
- The Kubernetes authentication method can also be enabled using the web ui
  - Go to Access > Auth Methods



Authentication Methods

token/
auth_token_c679ec21

  - Enable the Kubernetes method



Enable an Authentication Method

Generic

AppRole ○ JWT ○ OIDC ○ TLS Certificates ○ Username & Password ○

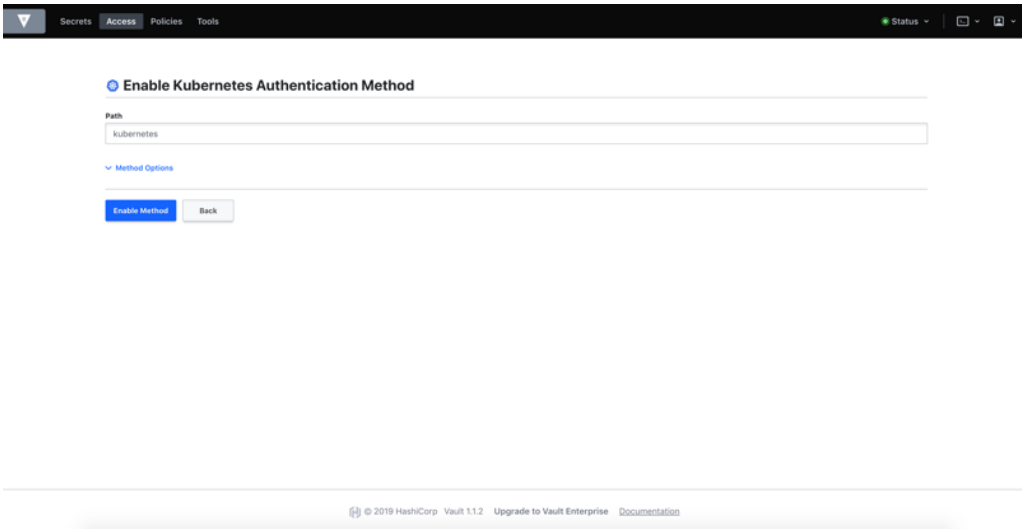
Cloud

AliCloud ○ AWS ○ Azure ○ Google Cloud ○ GitHub ○

Infra

Kubernetes ● LDAP ○ Okta ○ RADIUS ○

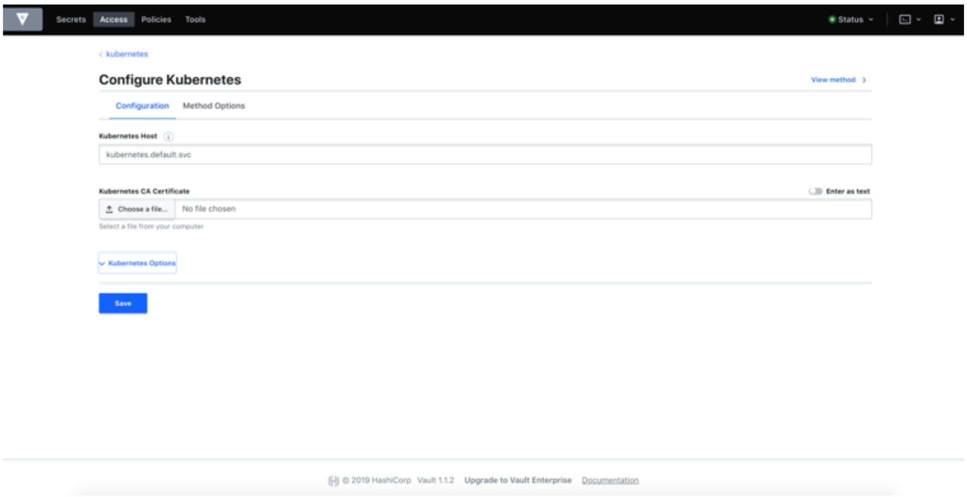
  - Keep the default path **Kubernetes** and press **Enable Method**


Secrets
Access
Policies
Tools
Status
Method Options
Enable Method
Back
© 2019 HashiCorp Vault 1.1.2 Upgrade to Vault Enterprise Documentation

Enable Kubernetes Authentication Method

Path: kubernetes

- ○ Provide Kubernetes api server host to vault. The api server can be addressed internally through the following host name: **kubernetes.default.svc**
  Refer to link: https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#accessing-the-api-from-a-pod



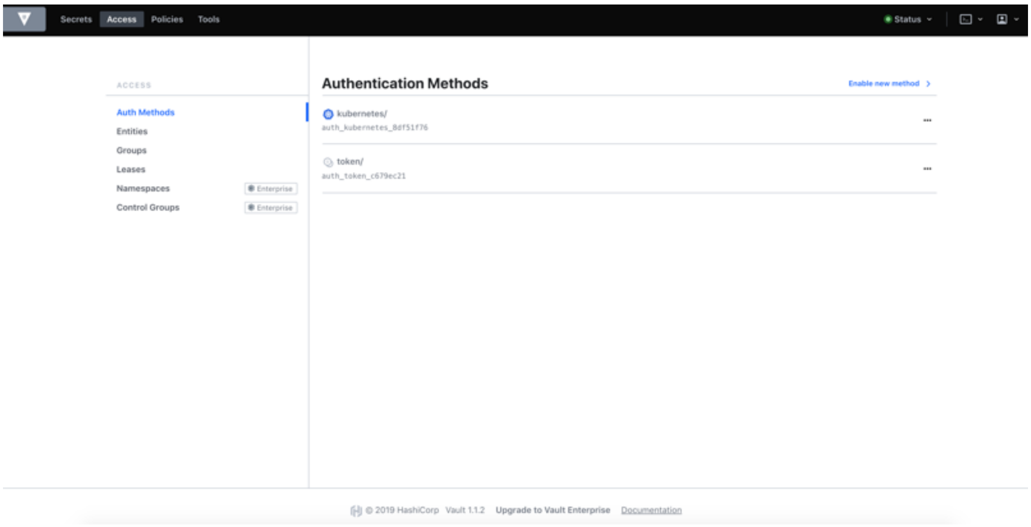
Configure Kubernetes

Kubernetes Host: kubernetes.default.svc

Kubernetes CA Certificate: No file chosen

Select a file from your computer

- ○ The Kubernetes auth method is now enabled in Vault



Authentication Methods

kubernetes/
auth_kubernetes_8df51f76

token/
auth_token_c679ec21

8.4. Configure Application

Each application pod must have:

- A Kubernetes service account giving it a unique identity within the cluster and a Kubernetes JWT token
- A Vault Policy grant it access to secret or other type of operations in Vault
- A role that associates the application Kubernetes service account to the application Vault Policy

8.4.1.Create Service Account

Each application should have a unique service account to manage its access to Kubernetes resources and to Vault secrets. The service account can be specified in the Helm chart of the application so that it can be created during the application installation stage.

The helm chart of the application will have a file named **service-account.yaml** that contains the following:

```
{{- if .Values.serviceAccount.create }}
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ include "app.fullname" . }}
{{- end }}
```

The creation of the service account must be enabled in the **values.yaml** file of the helm chart:

```
serviceAccount:
  create: true
```

→ All applications will use the same helm chart template and will all be configured to create a specific service account having the same name as the application.

8.4.2. Service Account Role Binding

The service account created for the application in the previous paragraph needs to be explicitely granted access to resources the application needs to work properly. This is necessary because of Kubernetes RBAC merchanism.

This is accomplished by creating a role for the application allowing it to access configmaps and other resources its needs, and by binding this role to the application service account.

We add therefore an rbac.yaml file in the application helm chart to create the role and the rolebinding.

```
{{- if .Values.rbac.create }}
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: {{ include "app.fullname" . }}
rules:
- apiGroups:
  - ""
  resources:
  - services
  - pods
  - endpoints
  - namespaces
  - configmaps
  - secrets
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: {{ include "app.fullname" . }}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: {{ include "app.name" . }}
subjects:
- kind: ServiceAccount
  name: {{ include "app.fullname" . }}
  namespace: {{ .Release.Namespace }}
{{- end }}
```

8.4.3. View Application Service Account in Kubernetes

Use the following command to get the service accounts in a given namespace (here **microservices**):

```
$ kubectl -n microservices get serviceAccounts

NAME      SECRETS   AGE
car       1         3m46s
default   1         7h33m
```

Use this command to get the metadata of a given service account:

```
$ kubectl -n microservices get serviceAccounts/car -o yaml

apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2019-08-09T16:41:51Z
  name: car
  namespace: microservices
  resourceVersion: "42387"
  selfLink: /api/v1/namespaces/microservices/serviceaccounts/car
  uid: 96dc7151-bac4-11e9-8d8b-3a0487af82f4
secrets:
- name: car-token-hw6k6
```

8.4.4. Create Application Vault Policy

Create a Vault policy **car-policy.hcl** file for **car** application for example. The policy file would have the following content:

```
path "secret/car" {
capabilities = ["read", "list"]
}
```

This policy will give car applications all access rights over its secret path **secret/car/***. Create the policy named **car** using the following command:

```
$ vault policy write car car-policy.hcl
```

8.4.5. Create Application Role in Vault

After the policy is created, we must bind it with the Kubernetes service account of the application. This binding can be achieved using the command below. We create a role named **“car”** for the application to name the binding.

```
$ vault write auth/kubernetes/role/car \
bound_service_account_names=car \
bound_service_account_namespaces=microservices \
policies=car
```

8.4.6. Test Authentication

To test and validate application authentication using the service account we created for the pod, we will do the following:

- Get the jwt token of the application service account
- Login using the jwt token and the application vault role

```
$ TOKEN_NAME=$(kubectl get sa vault-auth -o jsonpath="{.secrets[*]['name']}")
$ JWT=$(kubectl get secret $TOKEN_NAME -o jsonpath="{.data.token}" | base64 --
decode; echo)
bound_service_account_namespaces=microservices \
policies=car
```

8.4.7. Update Spring Boot Configuration

Now, that Kubernetes auth method is configured in Vault, and the role binding between the app service account and vault policy created, we can updated application **bootstrap.properties** file accordingly so it can use Kubernetes / Vault authentication:

```
# Application Name
spring.application.name=car

# Vault Kubernetes Parameters
spring.cloud.vault.enabled = ${VAULT_ENABLED}
spring.cloud.vault.uri=${VAULT_ADDR}
spring.cloud.vault.authentication = KUBERNETES
spring.cloud.vault.kubernetes.role = car
spring.cloud.vault.kubernetes.kubernetes-path = kubernetes
spring.cloud.vault.kubernetes.service-account-token-file =
/var/run/secrets/kubernetes.io/serviceaccount/token
spring.cloud.vault.kv.enabled=false
#spring.cloud.vault.kv.backend=secret
spring.cloud.vault.generic.enabled=true
spring.cloud.vault.generic.backend=secret
spring.cloud.vault.generic.default-context=car
```

8.4.8. Running Spring Boot Application

When running our car Spring Boot application, we can see in the traces below that the application managed to locate all property sources (configmap, secret, vault) and load them successfully:

```
PropertySourceBootstrapConfiguration : Located property source: CompositePropertySource
{name='vault', propertySources=[LeaseAwareVaultPropertySource
{name='secret/data/car/kubernetes'}, LeaseAwareVaultPropertySource {name='secret/data/car'},
LeaseAwareVaultPropertySource {name='secret/data/application/kubernetes'},
LeaseAwareVaultPropertySource {name='secret/data/application'},
LeaseAwareVaultPropertySource {name='secret/car/kubernetes'}, LeaseAwareVaultPropertySource
{name='secret/car'}, LeaseAwareVaultPropertySource {name='secret/application/kubernetes'},
LeaseAwareVaultPropertySource {name='secret/application'}]}

PropertySourceBootstrapConfiguration : Located property source: CompositePropertySource
```

```
{name='composite-configmap', propertySources=[ConfigMapPropertySource
{name='configmap.car.microservices'}]}

PropertySourceBootstrapConfiguration : Located property source: SecretsPropertySource
{name='secrets.car.microservices'}
```

→ There is no conflict in using Vault / Consul stores with Kubernetes Configmap / Secrets.

8.4.9. Vault vs Kubernetes K/V Store

From a client / developer perspective, the difference in storing secrets in Vault compared to Kubernetes is that with Vault and Spring Cloud Vault, we don't need to expose the secrets as environment variables to the application.
For example, in Vault we can create those 2 secrets:

Key	Value
spring.rabbitmq.username	guest
spring.rabbitmq.password	guest

The application with Spring Cloud Vault, will retrieve those secrets without any additional configuration.

With Kubernetes secrets however, we must create a secret (named **car**) that contains the same 2 Key/Values as in the table above.

Besides, we must in our pod deployment create environment variables to expose those secrets to the application. Here, the environment variables are:

- RABBITMQ_USERNAME
- RABBITMQ_PASSWORD

```
env:
- name: RABBITMQ_USERNAME
    valueFrom:
    secretKeyRef:
        name: car
        key: spring.rabbitmq.username
- name: RABBITMQ_PASSWORD
    valueFrom:
    secretKeyRef:
        name: car
        key: spring.rabbitmq.password
```

Then, in the configmap, we must set our application properties from exposed environment variables:

```
spring.rabbitmq.username = ${RABBITMQ_USERNAME}
spring.rabbitmq.password = ${RABBITMQ_PASSWORD}
```

→ From client perspective, using Spring Cloud Vault to provide secrets to application is easier than using Spring Cloud Kubernetes

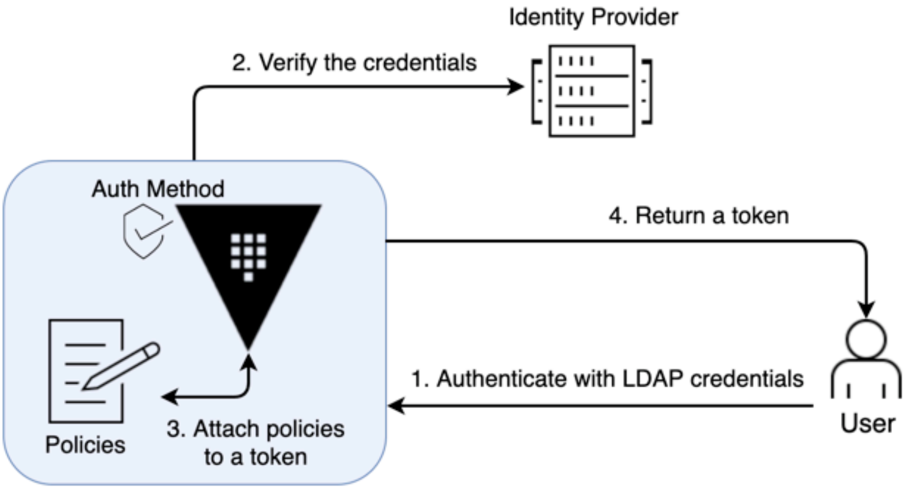
9. Single Sign-On with OIDC

Vault can be configured to use existing identity providers to authenticate users. Connection to identity providers use the Open ID Connect protocol (OIDC).

Example of identity providers tested with Vault:
https://www.vaultproject.io/docs/auth/jwt_oidc_providers.html

The following documentation describes how to enable and configured OIDC in Vault:
https://learn.hashicorp.com/vault/identity-access-management/oidc-auth

The diagram below shows the user authentication flow using OIDC:



In order to enable and configure OIDC, follow these steps:

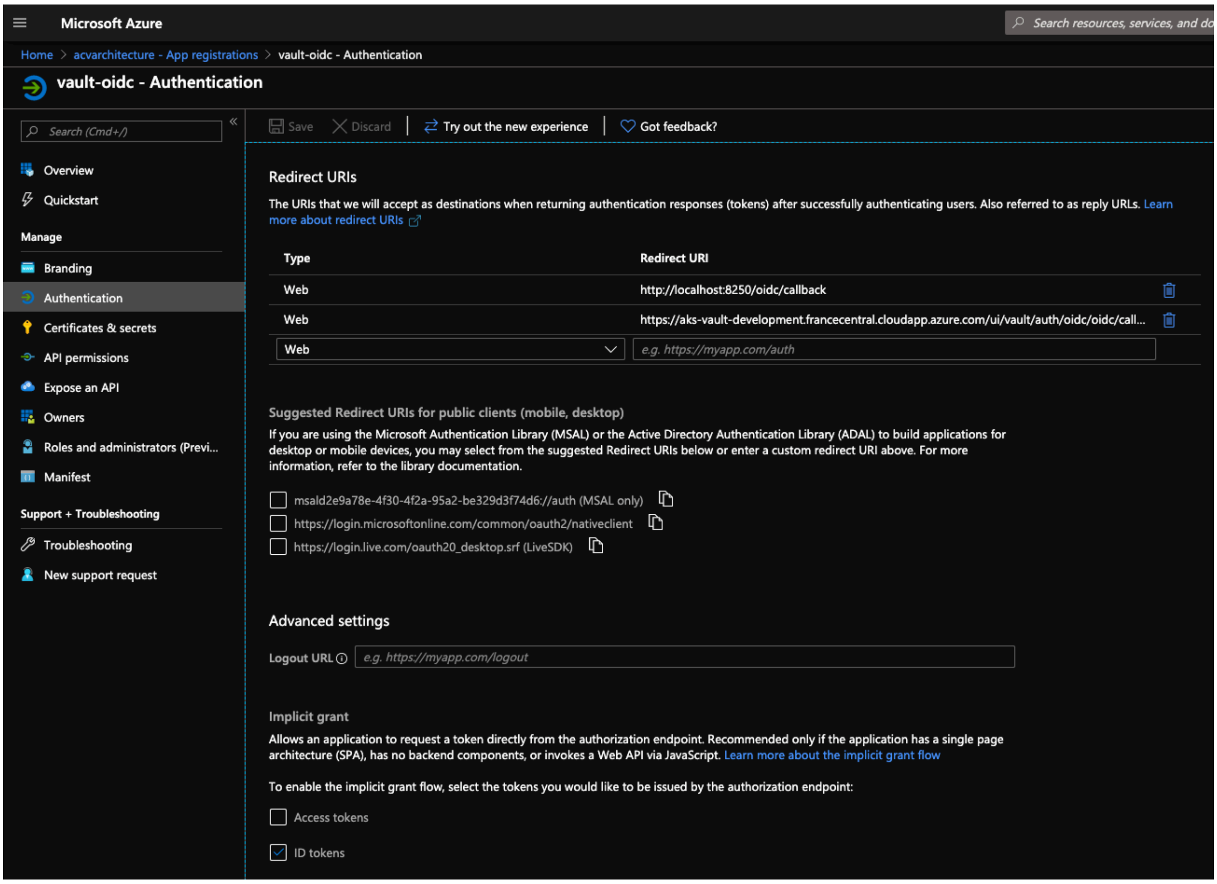
- Create an OIDC application in your OIDC provider (Azure AD, Okta, Autho, …)
- Enable the OIDC authentication method in Vault and configure it to use the OIDC application parameters (Discovery URL, Client ID, Client Secret)
- Create user roles in Vault (Admin, Provisioner, Secret Admin, Secret Reader for example). A role is bound to a vault policy that defines its access rights
- Create Internal identity groups in vault to be associated with different policies created
- Bind the internal group IDs to external group IDs (IDs created in azure ad or okta) by creating aliases

In the next paragraph, we give an example of how to create an OIDC application in Azure AD and link it to Vault.

9.1. Create Azure AD OIDC App

Azure AD OIDC Application can be created in Azure Portal in the following menu:
Azure Active Directory > App Registrations

We create an application named **vault-oidc.** Application ID is the client id to be used by vault to connect to **vault-oidc** application to perform user authentication.



vault-oidc - Authentication

Redirect URIs

The URIs that we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred to as reply URLs. Learn more about redirect URIs

Type	Redirect URI
Web	http://localhost:8250/oidc/callback
Web	https://aks-vault-development.francecentral.cloudapp.azure.com/ui/vault/auth/oidc/oidc/call...
Web	e.g. https://myapp.com/auth

Suggested Redirect URIs for public clients (mobile, desktop)

If you are using the Microsoft Authentication Library (MSAL) or the Active Directory Authentication Library (ADAL) to build applications for desktop or mobile devices, you may select from the suggested Redirect URIs below or enter a custom redirect URI above. For more information, refer to the library documentation.

□ msald2e9a78e-4f30-4f2a-95a2-be329d3f74d6://auth (MSAL only)
□ https://login.microsoftonline.com/common/oauth2/nativeclient
□ https://login.live.com/oauth20_desktop.srf (LiveSDK)

Advanced settings

Logout URL: e.g. https://myapp.com/logout

Implicit grant

Allows an application to request a token directly from the authorization endpoint. Recommended only if the application has a single page architecture (SPA), has no backend components, or invokes a Web API via JavaScript. Learn more about the implicit grant flow

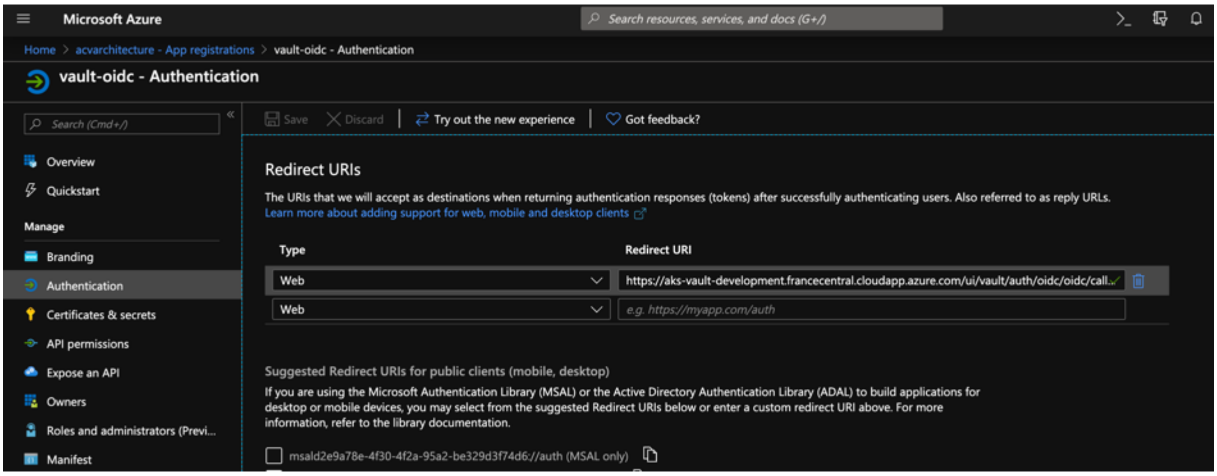
To enable the implicit grant flow, select the tokens you would like to be issued by the authorization endpoint:

□ Access tokens
☑ ID tokens

In application authentication menu, we provide the callback URL of the vault server that is called by azure ad **vault-oidc** application after user authentication.

Example of callback url:
https://aks-vault-development.francecentral.cloudapp.azure.com/ui/vault/auth/oidc/oidc/callback
http://localhost:8250/oidc/callback

→ The localhost URL is used with Vault command to perform OIDC authentication in the CLI
→ Make sure the vault server is secured with valid TLS certificate.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > acvarchitecture - App registrations > vault-oidc - Authentication
Search (Cmd+/)
Overview
Quickstart
Manage
Branding
Authentication
Certificates & secrets
API permissions
Expose an API
Owners
Roles and administrators (Previ...
Manifest
Save
Discard
Try out the new experience
Got feedback?

vault-oidc - Authentication

Redirect URIs

The URIs that we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred to as reply URLs. Learn more about adding support for web, mobile and desktop clients

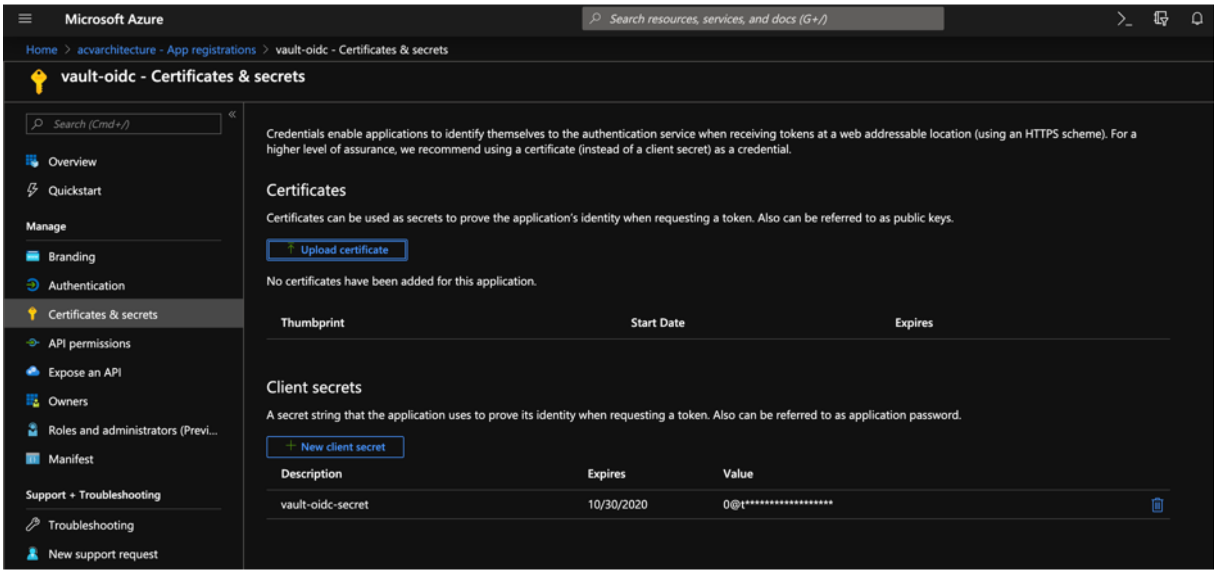
Type	Redirect URI
Web	https://aks-vault-development.francecentral.cloudapp.azure.com/ui/vault/auth/oidc/oidc/call...
Web	e.g. https://myapp.com/auth

Suggested Redirect URIs for public clients (mobile, desktop)

If you are using the Microsoft Authentication Library (MSAL) or the Active Directory Authentication Library (ADAL) to build applications for desktop or mobile devices, you may select from the suggested Redirect URIs below or enter a custom redirect URI above. For more information, refer to the library documentation.

□ msald2e9a78e-4f30-4f2a-95a2-be329d3f74d6://auth (MSAL only)

In Certificates & Secrets menu, create the client secret to be used by Vault to perform oidc authentication with **vault-oidc** azure ad application.



vault-oidc - Certificates & secrets

Credentials enable applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

Certificates

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

No certificates have been added for this application.

Thumbprint	Start Date	Expires

Client secrets

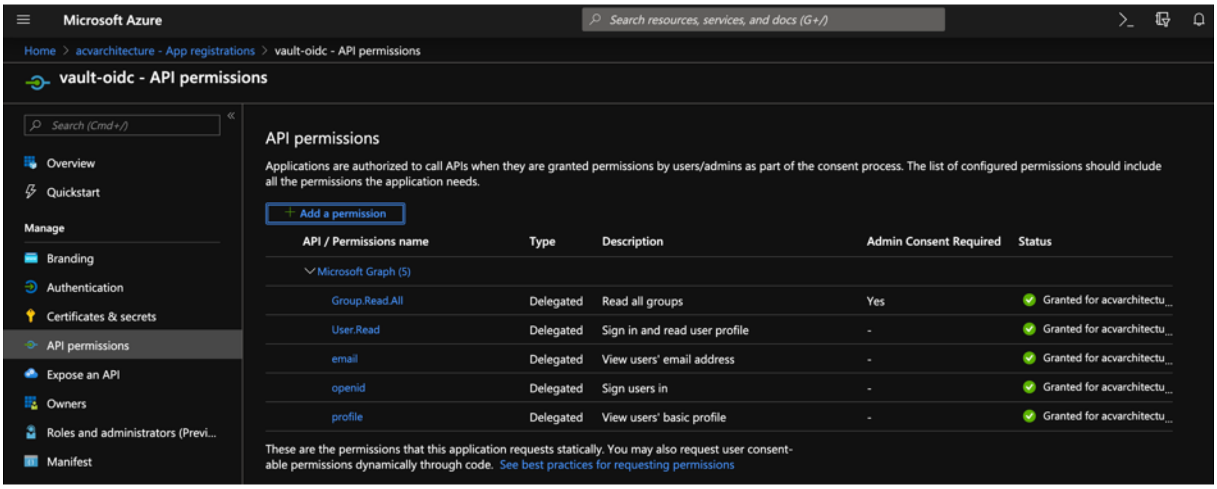
A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

Description	Expires	Value
vault-oidc-secret	10/30/2020	0@t******************

In API Permissions menu, provide **vault-oidc** application access to the following Azure AD resources:

- Group.Read.All
- User.Read
- Email
- Openid
- Profile

Azure admin consent is required for Group.Read.All permission.



vault-oidc - API permissions

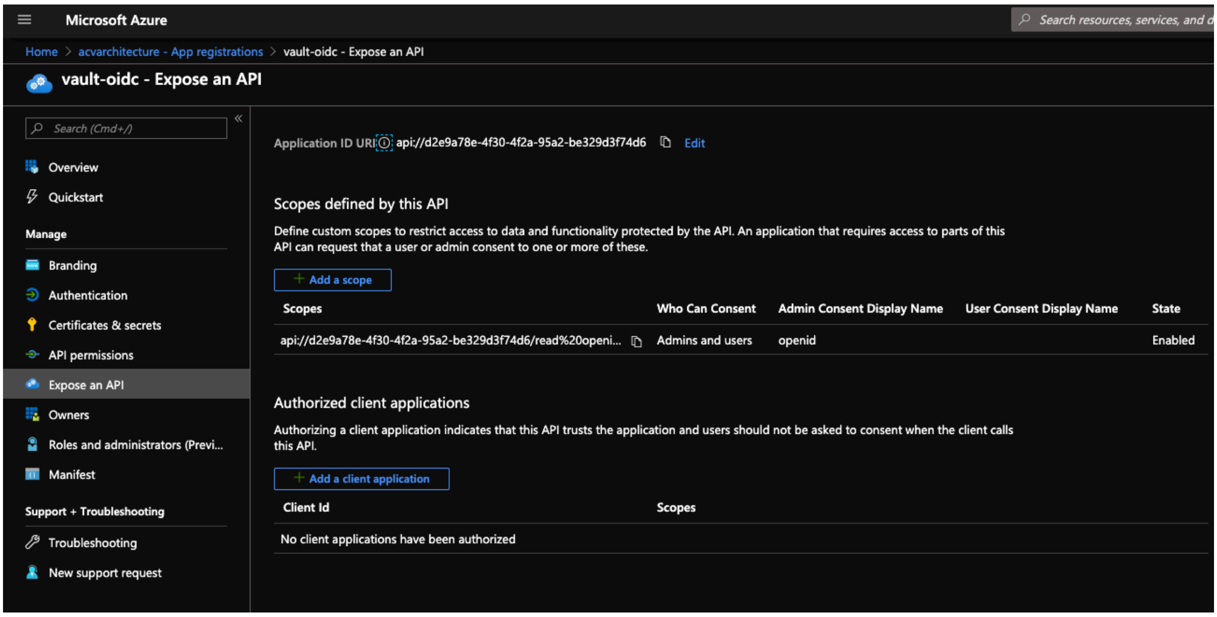
API permissions

Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs.

API / Permissions name	Type	Description	Admin Consent Required	Status
Microsoft Graph (5)				
Group.Read.All	Delegated	Read all groups	Yes	✓ Granted for acvarchitectu...
User.Read	Delegated	Sign in and read user profile	-	✓ Granted for acvarchitectu...
email	Delegated	View users' email address	-	✓ Granted for acvarchitectu...
openid	Delegated	Sign users in	-	✓ Granted for acvarchitectu...
profile	Delegated	View users' basic profile	-	✓ Granted for acvarchitectu...

These are the permissions that this application requests statically. You may also request user consent-able permissions dynamically through code. See best practices for requesting permissions

Expose an API for the resource to be exposed in Azure Resource Manager:


Microsoft Azure
Search resources, services, and d
Home > acvarchitecture - App registrations > vault-oidc - Expose an API
vault-oidc - Expose an API
Search (Cmd+/)
Overview
Quickstart
Manage
Branding
Authentication
Certificates & secrets
API permissions
Expose an API
Owners
Roles and administrators (Previ...
Manifest
Support + Troubleshooting
Troubleshooting
New support request
Edit
+ Add a scope
+ Add a client application

Application ID URI: api://d2e9a78e-4f30-4f2a-95a2-be329d3f74d6

Scopes defined by this API

Define custom scopes to restrict access to data and functionality protected by the API. An application that requires access to parts of this API can request that a user or admin consent to one or more of these.

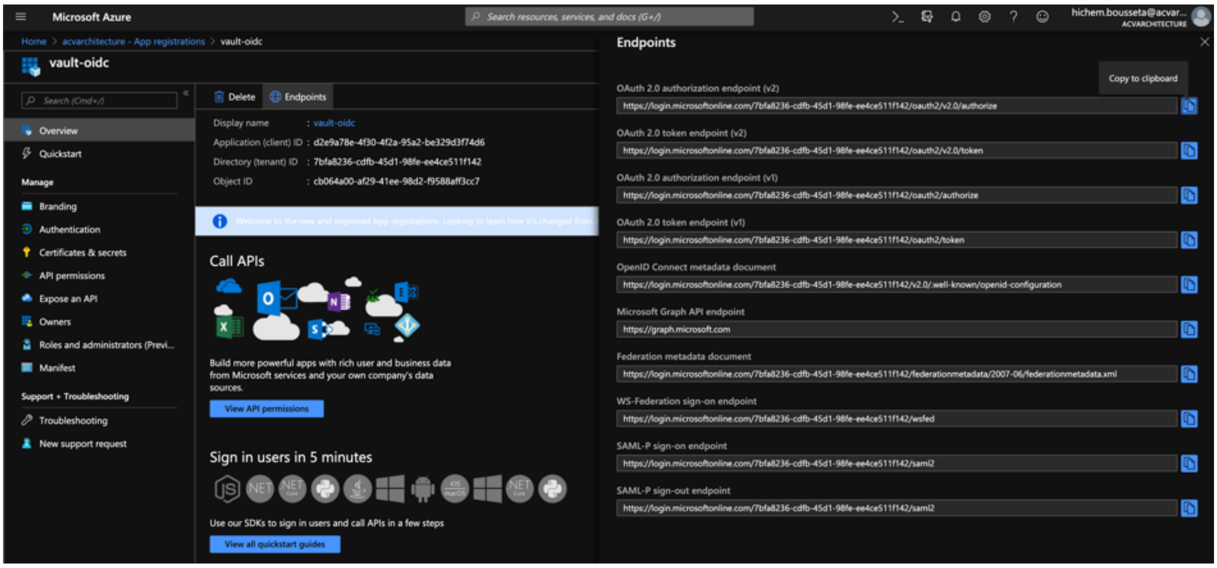
Scopes	Who Can Consent	Admin Consent Display Name	User Consent Display Name	State
api://d2e9a78e-4f30-4f2a-95a2-be329d3f74d6/read%20openi...	Admins and users	openid		Enabled

Authorized client applications

Authorizing a client application indicates that this API trusts the application and users should not be asked to consent when the client calls this API.

Client Id	Scopes
No client applications have been authorized	

In **vault-oidc** app Overview > Endpoints, get the OIDC discovery URL to be used to configure Vault with OIDC:


Microsoft Azure
Search resources, services, and docs (G+/)
hichem.bousseta@acvar...
ACVARCHITECTURE
Home > acvarchitecture - App registrations > vault-oidc
vault-oidc
Search (Cmd+/)
Overview
Quickstart
Manage
Branding
Authentication
Certificates & secrets
API permissions
Expose an API
Owners
Roles and administrators (Previ...
Manifest
Support + Troubleshooting
Troubleshooting
New support request
Delete
Endpoints
Welcome to the new and improved App registrations. Looking to learn how it's changed from
Call APIs
Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources.
View API permissions
Sign in users in 5 minutes
Use our SDKs to sign in users and call APIs in a few steps
View all quickstart guides
Copy to clipboard

Display name : vault-oidc
Application (client) ID : d2e9a78e-4f30-4f2a-95a2-be329d3f74d6
Directory (tenant) ID : 7bfa8236-cdfb-45d1-98fe-ee4ce511f142
Object ID : cb064a00-af29-41ee-98d2-f9588aff3cc7

Endpoints

OAuth 2.0 authorization endpoint (v2)
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/oauth2/v2.0/authorize

OAuth 2.0 token endpoint (v2)
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/oauth2/v2.0/token

OAuth 2.0 authorization endpoint (v1)
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/oauth2/authorize

OAuth 2.0 token endpoint (v1)
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/oauth2/token

OpenID Connect metadata document
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/v2.0/.well-known/openid-configuration

Microsoft Graph API endpoint
https://graph.microsoft.com

Federation metadata document
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/federationmetadata/2007-06/federationmetadata.xml

WS-Federation sign-on endpoint
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/wsfed

SAML-P sign-on endpoint
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/saml2

SAML-P sign-out endpoint
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/saml2

The OIDC discovery URL is:
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/v2.0/.well-known/openid-configuration

When you call this URL in browser, you get the settings of your **vault-oidc** application:

```
{
  "token_endpoint": "https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-
ee4ce511f142/oauth2/v2.0/token",
  "token_endpoint_auth_methods_supported": [
    "client_secret_post",
    "private_key_jwt",
    "client_secret_basic"
  ],
  "jwks_uri": "https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-
ee4ce511f142/discovery/v2.0/keys",
  "response_modes_supported": [
    "query",
    "fragment",
    "form_post"
  ],
  "subject_types_supported": [
    "pairwise"
  ],
  "id_token_signing_alg_values_supported": [
    "RS256"
  ],
  "response_types_supported": [
    "code",
    "id_token",
    "code id_token",
    "id_token token"
  ],
  "scopes_supported": [
    "openid",
    "profile",
    "email",
    "offline_access"
  ],
  "issuer": "https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/v2.0",
  "request_uri_parameter_supported": false,
  "userinfo_endpoint": "https://graph.microsoft.com/oidc/userinfo",
  "authorization_endpoint": "https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-
ee4ce511f142/oauth2/v2.0/authorize",
  "http_logout_supported": true,
  "frontchannel_logout_supported": true,
  "end_session_endpoint": "https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-
ee4ce511f142/oauth2/v2.0/logout",
  "claims_supported": [
    "sub",
    "iss",
    "cloud_instance_name",
    "cloud_instance_host_name",
    "cloud_graph_host_name",
    "msgraph_host",
    "aud",
    "exp",
    "iat",
    "auth_time",
    "acr",
    "nonce",
    "preferred_username",
    "name",
    "tid",
    "ver",
    "at_hash",
    "c_hash",
    "email"
  ],
  "tenant_region_scope": "EU",
  "cloud_instance_name": "microsoftonline.com",
  "cloud_graph_host_name": "graph.windows.net",
  "msgraph_host": "graph.microsoft.com",
  "rbac_url": "https://pas.windows.net"
}
```

The configuration is now done for the azure ad oidc application. We will move to Vault configuration.

9.2. Configure Vault OIDC

In order to enable user Single Sign-On and authentication to Vault using the azure ad OIDC application that we created in the previous paragraph, we must go through few steps (all theses actions are performed using the initial root token of vault):

→ Create specific roles for users. Here we will create a secret reader role, and secret manager role defined in 2 .acl files:

reader.acl (read access)

```
# Read permission on the k/v secrets
path "/secret/*" {
    capabilities = ["read", "list"]
}
```

manager.acl (read / write access)

```
# Manage k/v secrets
path "/secret/*" {
    capabilities = ["create", "read", "update", "delete", "list"]
}
```

Create the policies by running the following commands:

```
# Create manager policy
vault policy write manager aks-vault/k8s/policies/manager.hcl

# Create reader policy
vault policy write reader aks-vault/k8s/policies/reader.hcl
```

→ Enable OIDC authentication method

```
# Enable OIDC auth backend
vault auth enable oidc
```

→ Create OIDC Configuration using **vault-oidc** application parameters that we got in the previous paragraph:

- Discovery URL
- Client ID
- Client Secret

The discovery url is the same url as we had in the previous paragraph but without the **.well-known/openid-configuration** part.
https://login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce511f142/v2.0

```
# Create OIDC Config
vault write auth/oidc/config \
        oidc_discovery_url="https://login.microsoftonline.com/${TENANT_ID}/v2.0" \
        oidc_client_id="${AD_APP_ID}" \
        oidc_client_secret="${AD_APP_SECRET}" \
        default_role="reader"
```

We set the default role to reader.

→ Create Reader role

```
# Create a reader role
vault write -address="${VAULT_ADDR}"  auth/oidc/role/reader \
        bound_audiences="${AD_APP_ID}" \
        allowed_redirect_uris="${VAULT_ADDR}/ui/vault/auth/oidc/oidc/callback" \
        oidc_scopes="openid,profile,email,https://graph.microsoft.com/.default" \
        user_claim="sub" \
        groups_claim="groups" \
        policies="reader"
```

→ Create Manager role

```
# Create Manager Role
vault write -address="${VAULT_ADDR}"  auth/oidc/role/manager \
        bound_audiences="${AD_APP_ID}" \
        allowed_redirect_uris="${VAULT_ADDR}/ui/vault/auth/oidc/oidc/callback" \
        oidc_scopes="openid,profile,email,https://graph.microsoft.com/.default" \
        user_claim="sub" \
        groups_claim="groups" \
        policies="manager"
```

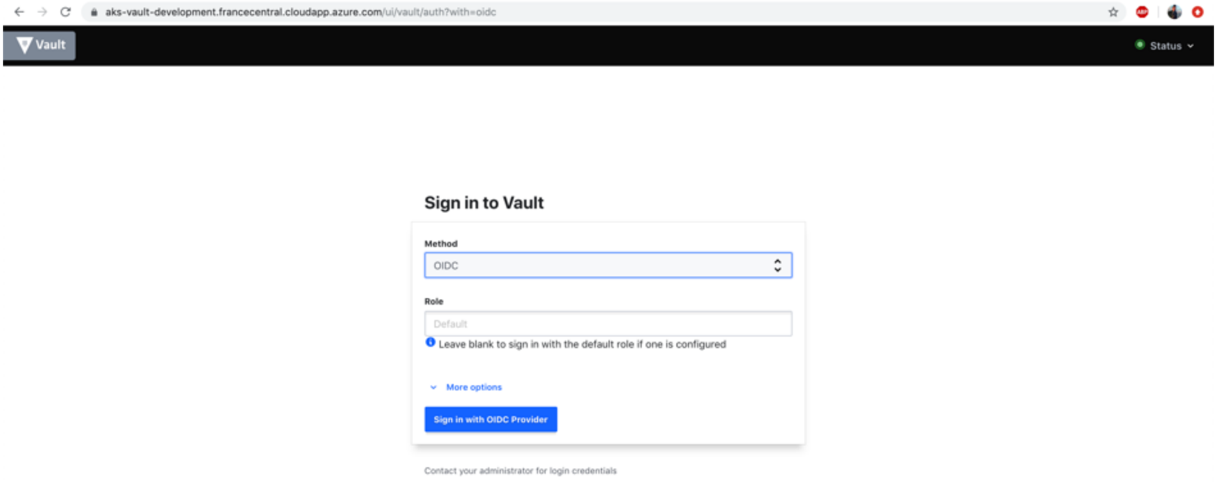
→ Create internal manager group to be bound to external azure ad amin group

```
OUT=$(vault write identity/group name="manager" type="external" policies="manager" metadata=responsibility="Manage K/V Secrets")
MANAGER_GROUP_ID=$(echo "$OUT" | grep id | awk -F ' ' '{print $2}')
OIDC_ACCESSOR=$(vault auth list | grep oidc | awk -F ' ' '{print $3}')

vault write identity/group-alias name="$AD_GROUP_ADMIN" \
        mount_accessor=${OIDC_ACCESSOR} \
        canonical_id="$MANAGER_GROUP_ID"
```

$AD_GROUP_ADMIN variable contains the ID of a group we created in azure ad
$MANAGER_GROUP_ID is the id of the manager group we created inside vault

9.3. Test OIDC Connection

Open the Vault server URL and select OIDC authentication:



aks-vault-development.francecentral.cloudapp.azure.com/ui/vault/auth?with=oidc

Sign in to Vault

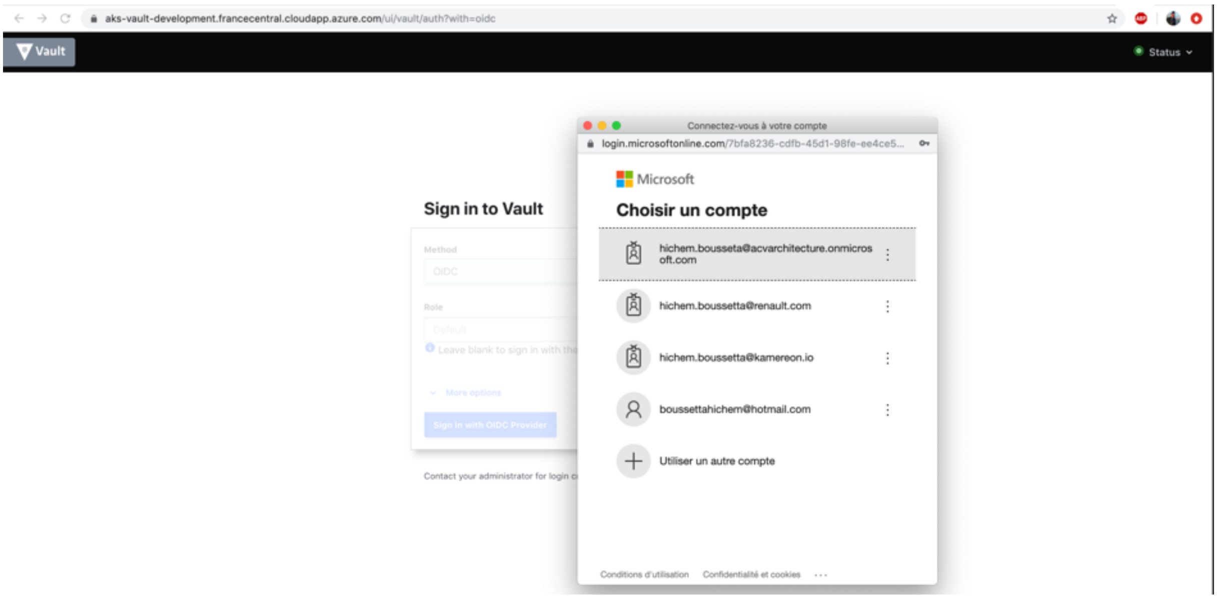
Method: OIDC

Role: Default

Leave blank to sign in with the default role if one is configured

Contact your administrator for login credentials

Leave the role to Default and click on **Sign in with OIDC Provider**. We are presented the azure login screen and invite to provide azure credentials:



aks-vault-development.francecentral.cloudapp.azure.com/ui/vault/auth?with=oidc

Connectez-vous à votre compte

login.microsoftonline.com/7bfa8236-cdfb-45d1-98fe-ee4ce5...

Microsoft

Choisir un compte

hichem.bousseta@acvarchitecture.onmicrosoft.com

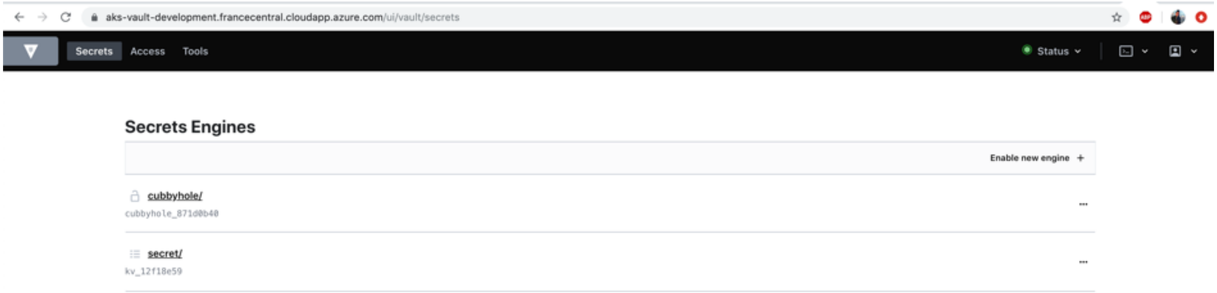
hichem.boussetta@renault.com

hichem.boussetta@kamereon.io

boussettahichem@hotmail.com

Utiliser un autre compte

Connect using your azure credentials to get access to vault:


aks-vault-development.francecentral.cloudapp.azure.com/ui/vault/secrets
Secrets
Access
Tools
Status
Enable new engine +

Secrets Engines

cubbyhole/
cubbyhole_871d0b40

secret/
kv_12f18e59