KPMG

Kubernetes secrets and Azure Key Vault as secret storage.

Agenda

- Introduction
- Kubernetes secrets overview
- How Kubernetes secrets works
- Introduction to Azure Key Vault
  - Overview
  - Features
  - Integration with AKS
  - Demo
- Best practise for Kubernetes secrets

kubernetes

Introduction

Secrets are already a fundamental building block of the modern Software Development Lifecycle (SDLC). Applications, CI/CD systems, API access, Databases, etc., all require some form of secrets/tokens/credentials in one way or another: secrets are literally involved in every single stage of the SDLC.

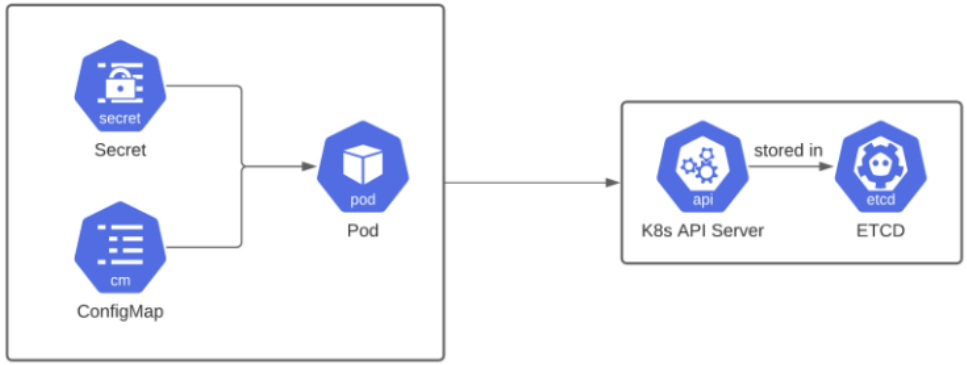
- Password
- Token
- API keys
- Any sensitive data that need to be hidden from others

Kubernetes secrets overview

Kubernetes Secrets is special Kubernetes document/object which aimed to store sensitive data.

There are two major ways in which Kubernetes secrets can be consumed by a container in a Pod in a Kubernetes environment:

- mounted as data volumes
- exposed as environment variables



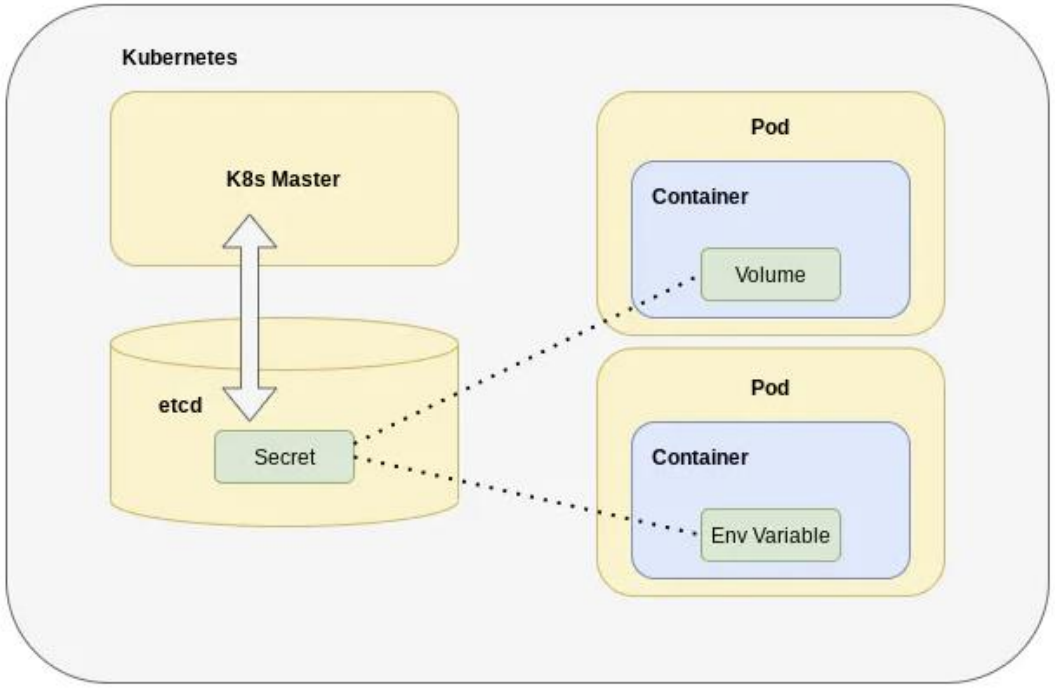
Built-in Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	ServiceAccount token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

How Kubernetes secrets works

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key.

Secrets are limited to 1MiB in size.

Secrets stored in ETCD component of control plane in plain text.



How Kubernetes secrets works

There are two ways to create secrets:

- From the command line

```
kubectl create secret generic prod-db-secret --from-literal=username=produser --from-literal=password=Y4nys7f11
```

- From command line using manifest file

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

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

How Kubernetes secrets works

Once secrets created we can use them in POD definition

Mount secret as volume

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
  labels:
    name: secret-test
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: ssh-key-secret
  containers:
  - name: ssh-test-container
    image: mySshImage
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

Mount as environment variable

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-single-secret
spec:
  containers:
  - name: envars-test-container
    image: nginx
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: backend-user
          key: backend-username
```

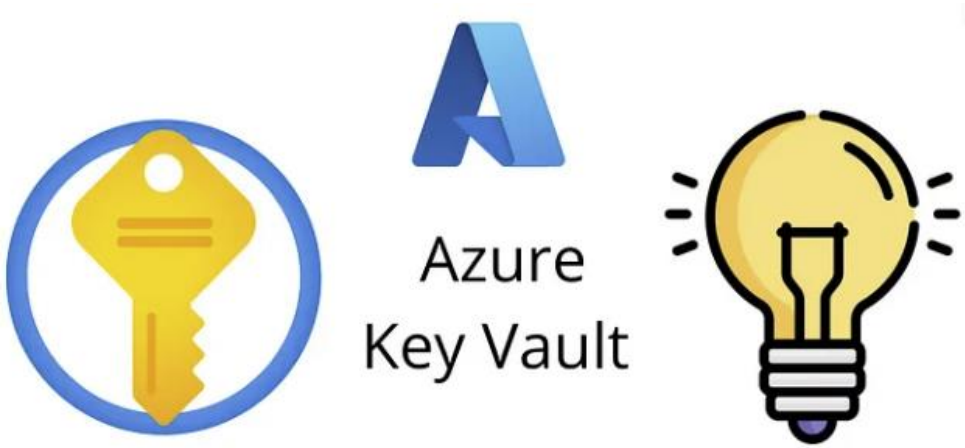
Challenges

An applications configuration will likely vary between deploys (staging, production, developer environments, etc.). The use of secrets files has its weaknesses and brings challenges, such as:

- Should we check in the secrets in files in your version control system?
- If so, how to encrypt them (since they are secrets)?
- If not, where to store these files?
- What if secrets files are scattered in different repos?
- Where is the single source of truth?
- What if you deployed new applications and old one is deprecated? Will you remember to delete old secrets from Kubernetes?

Azure Key Vault

- Secrets Management
- Key Management
- Certificate Management



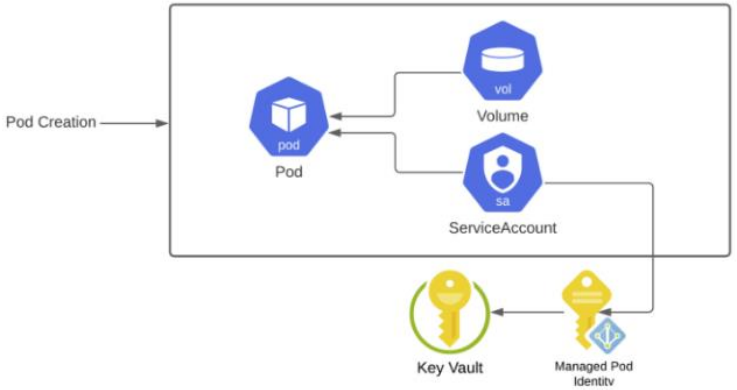
Benefits

- Centralized application secrets
- Securely store secrets and keys
- Integrate with other Azure services (like Azure Kubernetes Services)
- Access monitoring and control
- High Availability
- Scalability
- Simplified administration
- Encryption of data in transit

Azure Key Vault and Azure Kubernetes Service

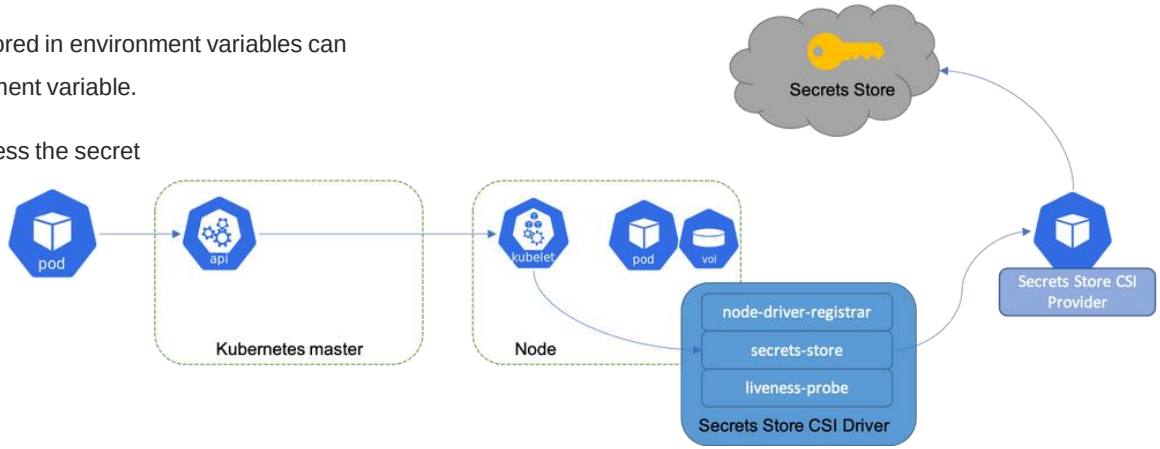
Secrets are stored at rest in Key Vault, in a secure encrypted store.

- Secrets are only stored in the AKS cluster when a pod is running with the secret mounted. As soon as those pods are removed, the secret is removed from the cluster, rather than with Kubernetes secrets which will be retained after a pod is removed.
- Key Vault remains the source of truth for the secrets, so you can update the secret in a single place and roll it out to your clusters.
- Auto certificate renewal or key rotation



Azure Key Vault Store CSI driver helps to:

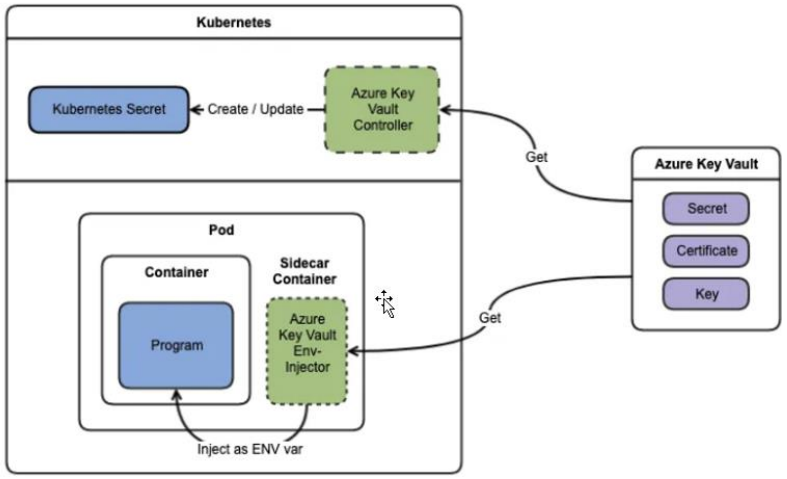
- Applications that require Kubernetes Secrets can access these secrets without issue.
- Applications that require values to be stored in environment variables can have the secret mounted as an environment variable.
- Other, non-pod-based services can access the secret



Azure Key Vault and Azure Kubernetes Service

- For a secrets to be available through environment variables, it first must be mounted by at least one pod
- Secrets will be synchronized after you start a pod to mount them. When you delete the pods that consume the secrets, your Kubernetes secret will also be deleted.

```yaml
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: azure-kvname
  namespace: myNameSpace
spec:
  provider: azure
  parameters:
    usePodIdentity: "true"
    keyvaultName: "<key Vault Name>"
    objects:  |
      array:
        - |
          objectName: secret1
          objectType: secret
        - |
          objectName: key1
          objectType: key
    tenantId: "<tenant ID which the Key Vault sits under"
  secretObjects:
  - secretName: appsecrets
    data:
    - key: secret1
      objectName: secret1
    type: Opaque
```



- «objects» is a array of secrets we are providing to AKS
- «secretObjects» array of secrets that AKS will sync to create Kubernetes secret objects (mounting as variable environment)

Azure Key Vault and Azure Kubernetes Service

TLS certificates as secret

1. Declaring SecretProviderClass

```yaml
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: azure-tls
spec:
  provider: azure
  secretObjects:
  - secretName: ingress-tls-csi
    type: kubernetes.io/tls
    data:
    - objectName: $CERT_NAME
      key: tls.key
    - objectName: $CERT_NAME
      key: tls.crt
  parameters:
    usePodIdentity: "false"
    useVMManagedIdentity: "true"
    userAssignedIdentityID: <client id>
    keyvaultName: $AKV_NAME
    objects: |
      array:
        - |
          objectName: $CERT_NAME
          objectType: secret
    tenantId: $TENANT_ID
```

- Use ***objectType=secret***, which is the only way to obtain the private key and the certificate from Azure Key Vault
- Set ***kubernetes.io/tls*** as the type in your *secretObjects* section

2. Mounting tls secret as volume

```yaml
  volumeMounts:
  - name: secrets-store-inline
    mountPath: "/mnt/secrets-store"
    readOnly: true
volumes:
- name: secrets-store-inline
  csi:
    driver: secrets-store.csi.k8s.io
    readOnly: true
    volumeAttributes:
      secretProviderClass: "azure-tls"
```

3. Verifying the Kubernetes secret has been created

```
NAME              TYPE                DATA   AGE
ingress-tls-csi   kubernetes.io/tls   2      1m34s
```

Azure Key Vault and Azure Kubernetes Service

How Secrets Store CSI authenticate against Azure Key Vault?

The Secrets Store CSI Driver allows for the following methods to auth against Azure key vault:

1. An **Azure Active Directory pod identity** (in deprecation).
2. **Azure Active Directory Workload Identity** is the newer version of Azure AD Pod Identity
3. A **User-assigned** or **System-assigned** managed identity

Azure Key Vault authorization for identities need to be set either through RBAC or access policies

Demo and Best practise

Best practices for Kubernetes Secrets:

1. Enable encryption at rest
2. Configure RBAC rules
3. Use a centralized Secrets store for easy management