→ repository require secure access of other resources → secrets

Secrets Management with GitOps

GitOps projects have git repositories in the center that are considered a source of truth for managing both infrastructure and application. This infrastructure and application will require secured access to other resources of the system through secrets. Committing clear-text secrets into git repositories is unacceptable even if the repositories are private to your team and organization. Teams need a secure way to handle secrets when using GitOps.

There are many ways to manage secrets with GitOps and at high level can be categorized into:

1. Encrypted secrets in git repositories
2. Reference to secrets stored in the external key vault

} clear text secrets into git repository is unacceptable.

> **TLDR**: Referencing secrets in an external key vault is the recommended approach. It is easier to orchestrate secret rotation and more scalable with multiple clusters and/or teams.

1 Encrypted Secrets in Git Repositories

In this approach, Developers manually encrypt secrets using a public key, and the key can only be decrypted by the custom Kubernetes controller running in the target cluster. Some popular tools for his approach are Bitnami Sealed Secrets, Mozilla SOPS

All the secret encryption tools share the following:

- Secret changes are managed by making changes within the GitOps repository which provides great traceability
- All secrets can be rotated by making changes in GitOps, without accessing the cluster
- They support fully disconnected gitops scenarios
- Secrets are stored encrypted in the gitops repository, if the private encryption key is leaked and the attacker has access to the repo, all secrets can be decrypted

Bitnami Sealed Secrets

Sealed Secrets use asymmetric encryption to encrypt secrets. A Kubernetes controller generates a key-pair (private-public) and stores the private key in the cluster's `etcd` database as a Kubernetes secret. Developers use Kubeseal CLI to seal secrets before committing to the git repo.

Some of the key points of using Sealed Secrets are:

- Support automatic key rotation for the private key and can be used to enforce re-encryption of secrets
  - Due to automatic renewal of the sealing key, the key needs to be prefetched from the cluster or cluster set up to store the sealing key on renewal in a secondary location
- Multi-tenancy support at the namespace level can be enforced by the controller
- When sealing secrets developers need a connection to the cluster control plane to fetch the public key or the public key has to be explicitly shared with the developer
- If the private key in the cluster is lost for some reason all secrets need to be re-encrypted followed by a new key-pair generation
- Does not scale with multi-cluster, because every cluster will require a controller having its own key pair
- Can only encrypt `secret` resource type
- The Flux documentation has inconsistences in the Azure Key Vault examples

Mozilla SOPS

SOPS: Secrets OPerationS is an encryption tool that supports YAML, JSON, ENV, INI, and BINARY formats and encrypts with AWS KMS, GCP KMS, Azure Key Vault, age, and PGP and is not just limited to Kubernetes. It supports integration with some common key management systems including Azure Key Vault, where one or more key management system is used to store the encryption key for encrypting secrets and not the actual secrets.

Some of the key points of using SOPS are:

- Flux has native support for SOPS with cluster-side decryption

- Provides an added layer of security as the private key used for decryption is protected in an external key vault
- To use the Helm CLI for encryption the (Helm Secrets) plugin is needed
- Needs (KSOPS)(kustomize-sopssecretgenerator) plugin to work with Kustomization
- Does not scale with larger teams as each developer has to encrypt the secrets
- The public key is sufficient for creating brand new files. The secret key is required for decrypting and editing existing files because SOPS computes a MAC on all values. When using the public key solely to add or remove a field, the whole file should be deleted and recreated
- Supports several types of keys that can be used in both connected and disconnected state. A secret can have a list of keys and will try do decrypt with

2 Reference to Secrets Stored in an External Key Vault (Recommended)

This approach relies on a key management system like Azure Key Vault to hold the secrets and the git manifest in the repositories has reference to the key vault secrets. Developers do not perform any cryptographic operations with files in repositories. Kubernetes operators running in the target cluster are responsible for pulling the secrets from the key vault and making them available either as Kubernetes secrets or secrets volume mounted to the pod.

All the below tools share the following:

- Secrets are not stored in the repository
- Supports Prometheus metrics for observability
- Supports sync with Kubernetes Secrets
- Supports Linux and Windows containers
- Provides enterprise-grade external secret management
- Easily scalable with multi-cluster and larger teams
- Both solutions support either Azure Active Directory (Azure AD) service principal or managed identity for authentication with the Key Vault.

For secret rotation ideas, see Secrets Rotation on Environment Variables and Mounted Secrets

For how to authenticate private container registries with a service principal see: Authenticated Private Container Registry

Azure Key Vault Provider for Secrets Store CSI Driver

Azure Key Vault Provider (AKVP) for Kubernetes secret store CSI Driver allows you to get secret contents stored in an Azure Key Vault instance and use the Secrets Store CSI driver interface to mount them into Kubernetes pods. Mounts secrets/keys/certs to pod using a CSI Inline volume.

Azure Key Vault Provider for Secrets Store CSI Driver install guide.

CSI driver will need access to Azure Key Vault either through a service principal or managed identity (recommended). To make this access secure you can leverage Azure AD Workload Identity(recommended) or AAD Pod Identity. Please note AAD pod identity will soon be replaced by workload identity.

Product Group Links provided for AKVP with SSCSID:

1. Differences between ESO / SSCSID (GitHub Issue) 2. Secrets Management on K8S talk here (Native Secrets, Vault.io, and ESO vs. SSCSID)

Advantages:

- Supports pod portability with the SecretProviderClass CRD
- Supports auto rotation of secrets with customizable sync intervals per **cluster**.
- Seems to be the MSFT choice (Secrets Store CSI driver is heavily contributed by MSFT and Kubernetes-SIG)

Disadvantages:

- Missing disconnected scenario support: When the node is offline the SSCSID fails to fetch the secret and thus mounting the volume fails, making scaling and restarting pods not possible while being offline
- AKVP can only access Key Vault from a non-Azure environment using a service principal
- The Kubernetes Secret containing the service principal credentials need to be created as a secret in the same namespace as the application pod. If pods in multiple namespaces need to use the same SP to access Key Vault, this Kubernetes Secret needs to be created in each namespace.
- The GitOps repo must contain the name of the Key Vault within the SecretProviderClass
- Must mount secrets as volumes to allow syncing into Kubernetes Secrets

- Uses more resources (4 pods; CSI Storage driver and provider) and is a daemonset - not test on RPS / resource usage

External Secrets Operator with Azure Key Vault

The External Secrets Operator (ESO) is an open-sourced Kubernetes operator that can read secrets from external secret stores (e.g., Azure Key Vault) and sync those into Kubernetes Secrets. In contrast to the CSI Driver, the ESO controller creates the secrets on the cluster as K8s secrets, instead of mounting them as volumes to pods.

Docs on using ESO Azure Key vault provider here.

ESO will need access to Azure Key Vault either through the use of a service principal or managed identity (via Azure AD Workload Identity(recommended) or AAD Pod Identity).

Advantages:

- Supports auto rotation of secrets with customizable sync intervals per **secret**.
- Components are split into different CRDs for namespace (ExternalSecret, SecretStore) and cluster-wide (ClusterSecretStore, ClusterExternalSecret) making syncing more manageable i.r.t. different deployments/pods etc.
- Service Principal secret for the (Cluster)SecretStores could placed in a namespaced that only the ESO can access (see Shared ClusterSecretStore).
- Resource efficient (single pod) - not test on RPS / resource usage.
- Open source and high contributions, (GitHub)
- Mounting Secrets as volumes is supported via K8S's APIs (see here)
- Partial disconnected scenario support: As ESO is using native K8s secrets the cluster can be offline, and it does not have any implications towards restarting and scaling pods while being offline

Disadvantages:

- The GitOps repo must contain the name of the Key Vault within the SecretStore / ClusterSecretStore or a ConfigMap linking to it
- Must create secrets as K8s secrets

Resources

- Sealed Secrets with Flux v2
- Mozilla SOPS with Flux v2
- Secret Management with Argo CD
- Secret management Workflow

Appendix

Authenticated Private Container Registry

An option on how to authenticate private container registries (e.g., ACR):

1. Use a `dockerconfigjson` Kubernetes Secret on Pod-Level with `ImagePullSecret` (This can be also defined on namespace-level)

Last update: August 26, 2024