

Encrypting Secrets in Kubernetes Clusters using KMS

Chirag Kyal
Software Engineer



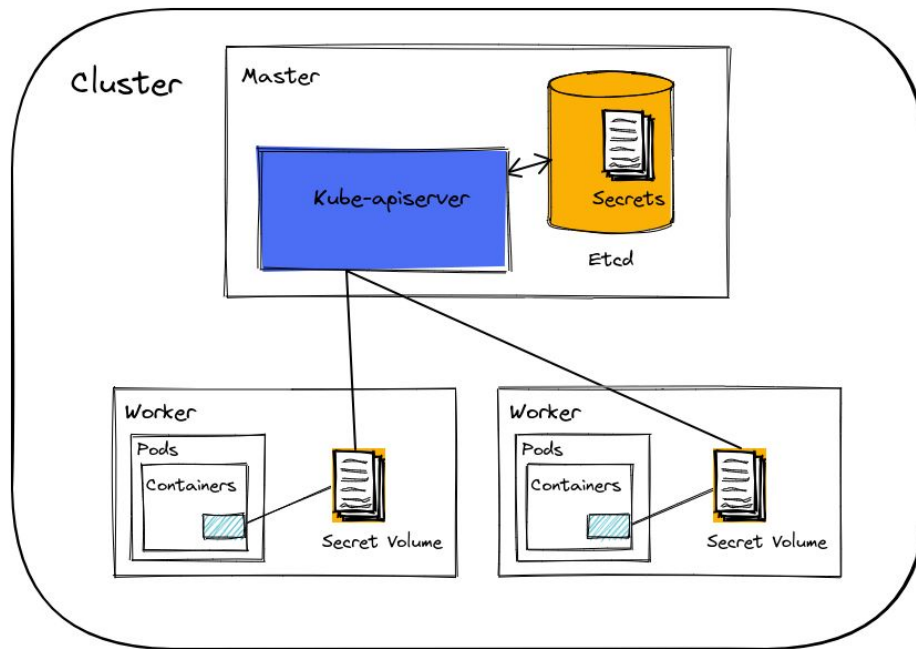
Red Hat

Swarup Ghosh
Software Engineer



Red Hat

Kubernetes Secrets



- Secret is an object that contains **sensitive data such as a password, a token, or a key**.
- Using a Secret means that you don't need to include confidential data in your application code.

[Alert] Security Problem

- Secrets are, by default stored **unencrypted** in etcd
- Attackers gain access to one of the etcd databases or one of the backups or disks with etcd data
- **The secrets can be leaked!**

```
$ etcdctl get  
"/registry/secrets/default/top-secret"  
/registry/secrets/default/top-secret  
  
v1Secret❖❖  
❖  
builder-token-66g5k❖❖default"*$054232f0-3206-4fdf  
-a72f-c32cd7a312712❖❖йbA  
❖❖kubernetes.io/created-by%openshift.io/create-do  
ckercfg-secretsb-  
"kubernetes.io/service-account.namebuilderbI  
!kubernetes.io/service-account.uid$bb6ab80a-30a6  
-46a0-a38e-c998de7a5b42❖❖  
...
```

so what's the

Solution?

encrypt secrets in etcd/datastore layer

```
00000000 2f 72 65 67 69 73 74 72 79 2f 73 65 63 72 65 74 |/registry/secret|
00000010 73 2f 64 65 66 61 75 6c 74 2f 73 65 63 72 65 74 |s/default/secret|
00000020 31 0a 6b 38 73 3a 65 6e 63 3a 61 65 73 63 62 63 |1.k8s:enc:aescbc|
00000030 3a 76 31 3a 6b 65 79 31 3a c7 6c e7 d3 09 bc 06 |:v1:key1:.1.....|
00000040 25 51 91 e4 e0 6c e5 b1 4d 7a 8b 3d b9 c2 7c 6e |%Q...1..Mz.=..|n|
00000050 b4 79 df 05 28 ae 0d 8e 5f 35 13 2c c0 18 99 3e |.y..(..._5.,...>|
[...]
00000110 23 3a 0d fc 28 ca 48 2d 6b 2d 46 cc 72 0b 70 4c |#:...(H-k-F.r.pL|
00000120 a5 fc 35 43 12 4e 60 ef bf 6f fe cf df 0b ad 1f |..5C.N`.o.....|
00000130 82 c4 88 53 02 da 3e 66 ff 0a |...S...>f..|
```

Kubernetes Encryption Providers

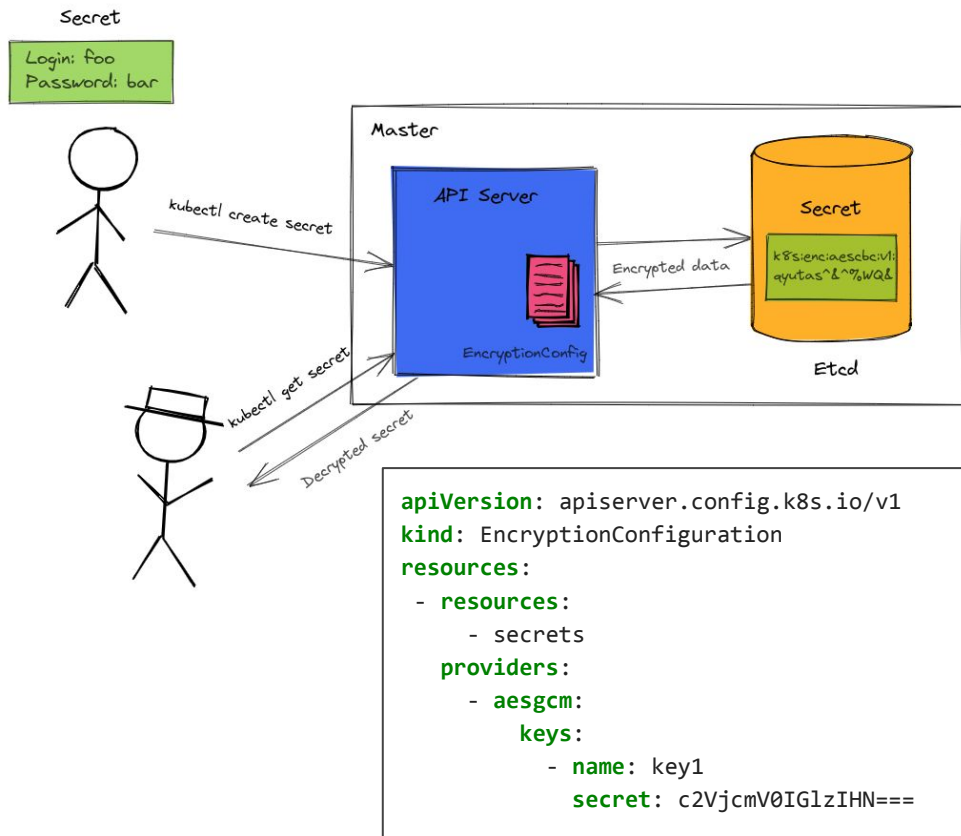
| Name | Encryption | Strength | Speed | Key length |
|---------------|--|--------------------------------------|---|--------------------|
| identity | None | N/A | N/A | N/A |
| | Resources written as-is without encryption. When set as the first provider, the resource will be decrypted as new values are written. Existing encrypted resources are not automatically overwritten with the plaintext data. The identity provider is the default if you do not specify otherwise. | | | |
| aescbc | AES-CBC with PKCS#7 padding | Weak | Fast | 32-byte |
| | Not recommended due to CBC's vulnerability to padding oracle attacks. Key material accessible from control plane host. | | | |
| aesgcm | AES-GCM with random nonce | Must be rotated every 200,000 writes | Fastest | 16, 24, or 32-byte |
| | Not recommended for use except when an automated key rotation scheme is implemented. Key material accessible from control plane host. | | | |
| kms v1 | Uses envelope encryption scheme with DEK per resource. | Strongest | Slow (<i>compared to kms version 2</i>) | 32-bytes |
| | Data is encrypted by data encryption keys (DEKs) using AES-GCM; DEKs are encrypted by key encryption keys (KEKs) according to configuration in Key Management Service (KMS). Simple key rotation, with a new DEK generated for each encryption, and KEK rotation controlled by the user. Read how to configure the KMS V1 provider . | | | |
| kms v2 (beta) | Uses envelope encryption scheme with DEK per API server. | Strongest | Fast | 32-bytes |
| | Data is encrypted by data encryption keys (DEKs) using AES-GCM; DEKs are encrypted by key encryption keys (KEKs) according to configuration in Key Management Service (KMS). A new DEK is generated at API server startup, and is then reused for encryption. The DEK is rotated whenever the KEK is rotated. A good choice if using a third party tool for key management. Available in beta from Kubernetes v1.27. Read how to configure the KMS V2 provider . | | | |
| secretbox | XSalsa20 and Poly1305 | Strong | Faster | 32-byte |
| | Uses relatively new encryption technologies that may not be considered acceptable in environments that require high levels of review. Key material accessible from control plane host. | | | |

Local Encryption Provider

- **Encryption Keys are stored in the EncryptionConfig** on the API Server itself
- Secrets encrypted prior to being stored in etcd and decrypted in API Server prior to use

Providers:

- AESGCM
- AESCBC
- Secretbox



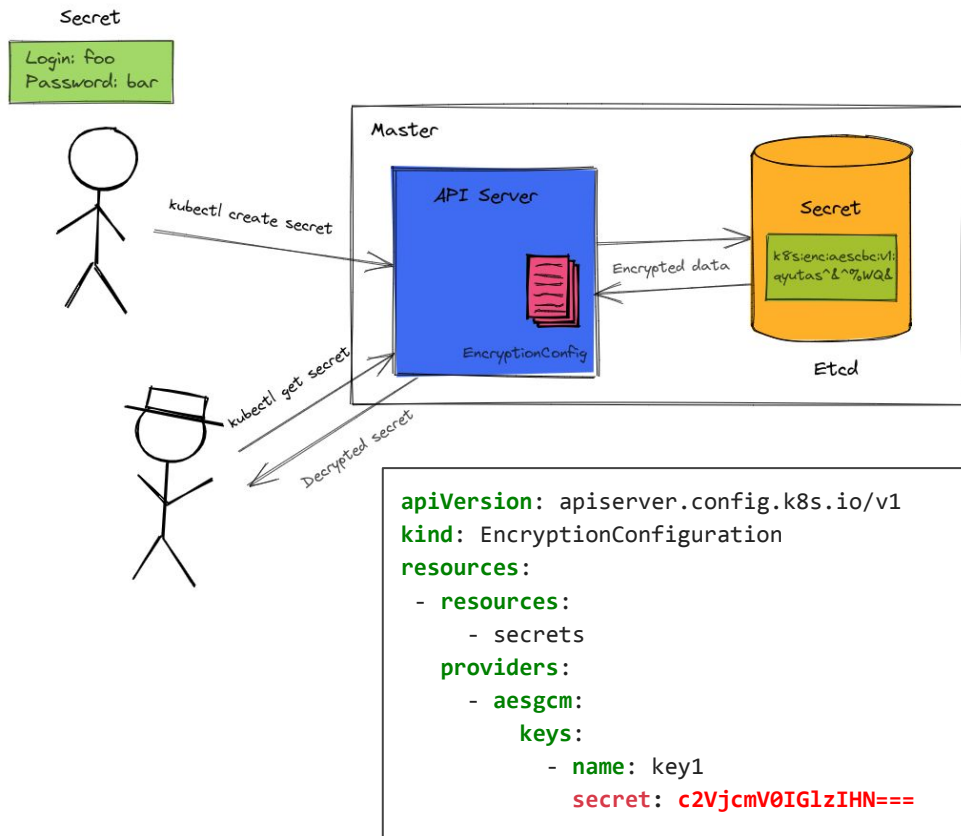
Encrypted data is stored in etcd in the following format:

`k8s:enc:aescbc:v1:[keyName]:[InitializationVector:8Bytes][EncryptedData]`

Local Encryption Provider

Threat Model:

- Mitigate: attacker accessing etcd database
- **Doesn't mitigate:** attacker accessing the host



Encrypted data is stored in etcd in the following format:

`k8s:enc:aescbc:v1:[keyName]:[InitializationVector:8Bytes][EncryptedData]`

Still, another problem...

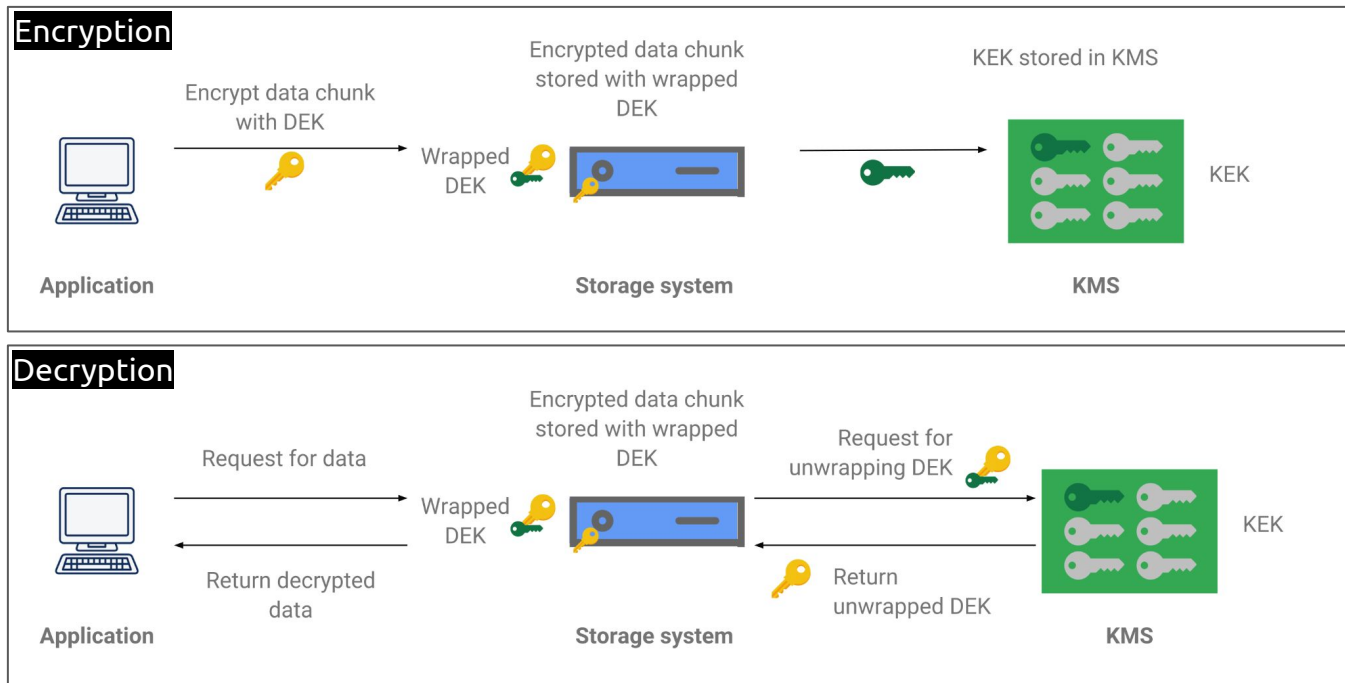
- Local encryption doesn't protect against **host compromise**
- If attacker gets access to the disk of the node or host itself, they will have access to the local encryption keys
 - They can use it to decrypt the encrypted etcd data



KMS Envelope Encryption

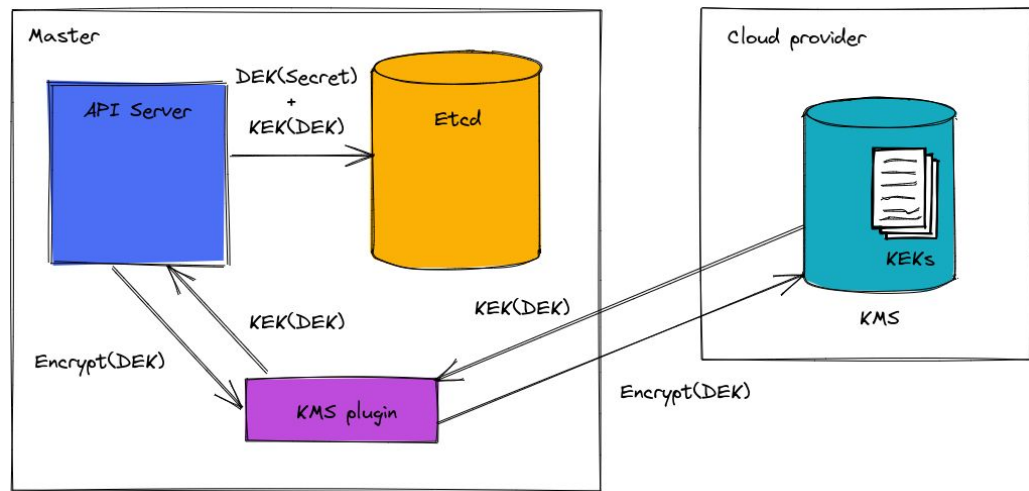
/kɪˈenɪəs/

A system for the management of cryptographic keys and their metadata including generation, distribution, storage, backup, archive, recovery, use, revocation, and destruction.



KMS Encryption Provider

- **Encryption Keys** are stored in a KMS external to the API server
- Secrets encrypted with a data encryption key (DEK)
- DEKs are wrapped with key encryption key (KEK)
- KEKs stored and managed in remote KMS



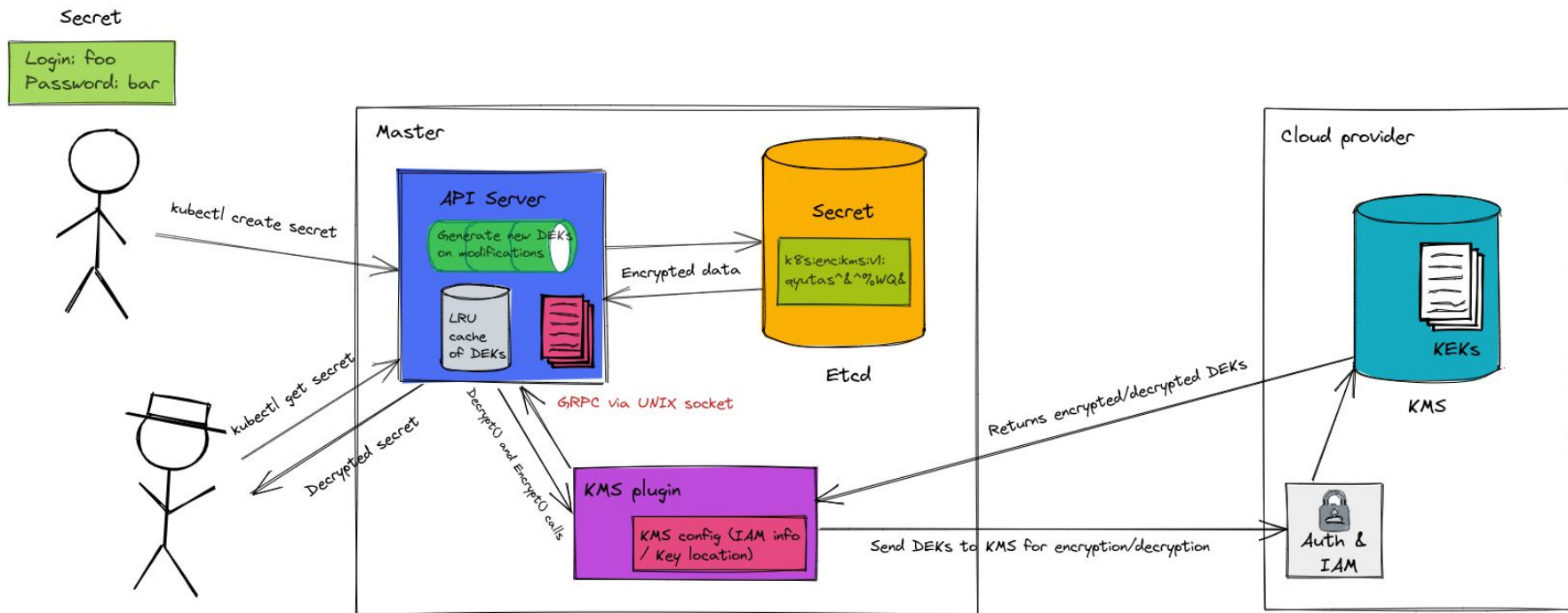
Provider:

- **KMS**

Encrypted data is stored in etcd in the following format:

`k8s:enc:kms:v1:[pluginName]:[InitializationVector:8Bytes][EncryptedData]`

KMS Encryption Provider



Threat Model:

- **Mitigates:** etcd compromise
- **Mitigates:** host compromise

[KMS v1] What happens when you create a Secret?

1. The Kubernetes API server generates a unique DEK for the secret using a random number generator.
2. The Kubernetes API server uses the DEK locally to encrypt the secret.
3. The KMS plugin sends the DEK to KMS for encryption. KMS encrypts the DEK using the remote KEK, and sends it back to the KMS plugin.
4. The Kubernetes API server saves the encrypted secret and the encrypted DEK. The plaintext DEK is not saved to disk.
5. The Kubernetes API server creates a cache entry mapping the encrypted DEK to the plaintext DEK which lets API server to decrypt the secret without KMS.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- resources:
  - secrets
providers:
- kms:
  name: myKmsPlugin
  endpoint:
    unix:///var/kms/plugin.sock
  cachesize: 100
  timeout: 3s
```

[KMS v1] What happens when you read a Secret?

1. The Kubernetes API server retrieves the encrypted secret and the encrypted DEK.
2. The Kubernetes API server checks the cache for an existing mapping entry of encrypted DEK and plaintext DEK.
3. If a cache entry is not found, the KMS plugin sends the DEK to KMS for decryption using the remote KEK.
4. The decrypted DEK is then used to decrypt the secret.
5. The Kubernetes API server returns the decrypted secret to the client.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- resources:
  - secrets
providers:
- kms:
  name: myKmsPlugin
  endpoint:
    unix:///var/kms/plugin.sock
  cachesize: 100
  timeout: 3s
```

[KMS v2] What happens on API Server startup?

- The Kubernetes API server generates a new DEK at startup and caches it.
- The API server also makes a call to the KMS plugin to encrypt the DEK using the remote KEK.
- This is a one-time call at startup however this can be repeated occasionally when KMS KEK is rotated.
 - When key ID is changed, the remote KEK would have been rotated.
 - Periodically, API server checks for the KEK key ID.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
  providers:
    - kms:
        apiVersion: v2
        name: myKmsPlugin
        endpoint:
          unix:///var/kms/plugin.sock
        timeout: 3s
```

[KMS v2] What happens when you create a Secret?

1. The API server encrypts the secret using the DEK from the local cache.
2. The Kubernetes API server saves the encrypted Secret, encrypted DEK and key ID. The plaintext DEK is not saved to disk.

```
// EncryptedObject is the representation of data
// stored in etcd after envelope encryption.
type EncryptedObject struct {
    // EncryptedData is the encrypted data.
    EncryptedData []byte
    // KeyID is the KMS key ID used for
    // encryption operations.
    KeyID string
    // EncryptedDEK is the encrypted DEK.
    EncryptedDEK []byte
    // Annotations is additional metadata
    // provided by KMS plugin.
    Annotations map[string][]byte
    // ...
}
```

Encrypted data is stored in etcd in the following format:
`k8s:enc:kms:v2:[pluginName]:[EncodedProtocolBuffer]`





[KMS v2] What happens when you read a Secret?

1. The Kubernetes API server retrieves the encrypted secret and the encrypted DEK.
2. The Kubernetes API server checks the cache for the encrypted DEK and gets the corresponding plaintext DEK.
3. If the key was not found in cache, the KMS plugin sends the DEK to Cloud KMS for decryption. The decrypted DEK is obtained.
4. The DEK is used to decrypt the encrypted secret.

```
// EncryptedObject is the representation of data
// stored in etcd after envelope encryption.
type EncryptedObject struct {
    // EncryptedData is the encrypted data.
    EncryptedData []byte
    // KeyID is the KMS key ID used for
    // encryption operations.
    KeyID string
    // EncryptedDEK is the encrypted DEK.
    EncryptedDEK []byte
    // Annotations is additional metadata
    // provided by KMS plugin.
    Annotations map[string][]byte
    // ...
}
```

Encrypted data is stored in etcd in the following format:
`k8s:enc:kms:v2:[pluginName]:[EncodedProtocolBuffer]`

KMS Plugins

| Plugin Provider | Cloud KMS Provider | Managed Kube Offering | KMS v1 | KMS v2 |
|--|--|---|--------|--------|
|  | Azure KeyVault | Azure Kubernetes Service (AKS) | Yes | Yes |
|  | AWS KMS | Amazon Elastic Kubernetes Service (EKS) | Yes | No |
|  Google Cloud | Cloud KMS | Google Kubernetes Engine (GKE) | Yes | No |
|  Trousseau | HashiCorp Vault, AWS KMS, Azure KeyVault | - | Yes | No |

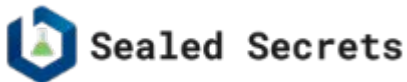
Alternatives



The Vault Agent Sidecar Injector is a Kubernetes admission webhook that adds Vault Agent containers to pods for consuming Vault secrets.



External Secrets Operator reads information from external secret management systems and injects the values into Kubernetes secrets.



Sealed Secrets Operator stores the secrets in the form of SealedSecrets such that the secrets can be managed through GitOps. Only decryption controller can decrypt the contents.

Thank you!

Chirag Kyal



Swarup Ghosh

