KMS-Plugin: Improvements

SHARED EXTERNALLY

anish.ramasekar@gmail.com, rita.z.zhang@gmail.com

10 August 2021

Reference doc: KMS-Plugin: Areas for improvement

This document summarises some of the KMS-Plugin improvements that we can target for the upcoming Kubernetes releases. The three areas of focus in this document are:

1. Performance
2. Observability
3. Recovery

Given the scope of changes in some of the categories, I recommend we handle these in phases.

Phase 1: Observability and Recovery (Incremental improvements to existing design)
Phase 2: Performance (Changes to existing implementation with backward compatibility)

Performance

Improve the readiness time for clusters with a large number of secrets.

Cache warm-up

This is when the kube-apiserver is restarted in a cluster and then a LIST secret call is made. As the cache is empty, the kube-apiserver calls the kms plugin for all the DEKs that have been generated so far. This serial call can cause the kms plugin to hit the external kms rate limit.

This is not in the scope for this enhancement proposal.

Optimise DEK generation

Currently a DEK is generated for each object and is then encrypted using a KEK. This 1:1 mapping means if there is a burst of secret creation, then the kms plugin will hit the external kms rate limit for encrypt operations.

The encryption configuration has a cacheSize field that can be configured. This cacheSize is used by the API server to initialise a lru cache of the given size with the encrypted cipher used as index. Having a higher value for the cacheSize will prevent calls to the provider for decryption operations. However this does not solve the issue with the number of calls to KMS plugin when encryption traffic is bursty

Reduce the number of trips to KMS by caching DEKs: Attempts to reuse DEKs for a configured TTL period.

- This behaviour is similar to how KEKs are configured today and rotated over time. The DEK TTL period will be shorter (in the magnitude of seconds or few hours).
- Outstanding issue with the approach is how to inform the API server to rotate the DEKs when a KEK has been rotated?
- Today, the health check from kube-apiserver -> kms plugin is an Encrypt operation followed by Decrypt operation. Instead if we provide a separate HealthCheck RPC call with HealthCheckRequest and HealthCheckResponse, the kms plugin can indicate the change in KEK version as part of HealthCheckResponse. This could be an indication that the KEK is now rotated and re-encrypt of existing DEKs is required.

Add an option to use hierarchy in KMS plugin

1. No changes to API server, keep 1:1 DEK mapping
2. KMS plugin generates its own local KEK in-memory
3. Real KMS is used to encrypt the local KEK
4. Local KEK is used for encryption of DEKs sent by API server
5. Local KEK is used for encryption based on policy (N events, X time, etc)

Since key hierarchy is implemented at the kms provider level, it should be seamless for the kube-apiserver. So whether the provider is using a local KEK or not, the kube-apiserver should behave the same.

What is required of the kube-apiserver is to be able to tell the kms provider which KEK it should use to decrypt the incoming DEK. To do so, the KMS provider could provide a UUID identifying the local KEK in the EncryptionResponse. The kube-apiserver would then store it in etcd next to the DEK and provide it when calling Decrypt. In case no UUID is provided, then the kms provider would query the kms directly.

In terms of storage, we could make the following change:

- **From**: k8s:enc:kms:v1:[pluginName]:v1:[2 bytes DEK][aescbc encrypted data]
- **To**: k8s:enc:kms:v1:[pluginName]:v1:[16 bytes local KEK UUID]:[2 bytes DEK][aescbc encrypted data]
- We need to store local KEK in etcd and once decrypted it can be stored in memory to be used for encryption and decryption of DEK.
    - We may want to allow KMS plugins to return extra metadata in more structured way, i.e. for the purposes of debugging, data recovery, etc

Hot reload of EncryptionConfig

Currently, any change to the EncryptionConfiguration requires a kube-apiserver restart for the changes to take effect. For a single kube-apiserver configuration this can lead to a brief period when the kube-apiserver is unavailable.

On a high level, to support hot reload of the EncryptionConfig:

1. Watch on the EncryptionConfig
2. When changes are detected, process the EncryptionConfig and add new transformers/update existing ones atomically.
3. If there is an issue with creating/updating any of the transformers, we should retain the current configuration in the kube-apiserver and generate an error in logs to indicate error.

https://github.com/kubernetes/enhancements/tree/0e4d5df19d396511fe41ed0860b0ab9b96f46a2d/keps/sig-api-machinery/1965-kube-apiserver-identity -> see if there is a way to get per API server observed config

See if the storage version migrator can accurately detect that migration successfully occured for a particular resource after time T when a config was observed

Rotation in regards to latest key being used

- Would be nice if users do not have to think about it
- But, hard to track what key is being
    - Could make rollback difficult
- The idea is to express configuration that could be interpreted dynamically at runtime
    - A minimal implementation would be to allow the KMS plugin to hot reload what key is the current write key

Option to choose encryption protocol

Provide users with the option to choose the encryption protocol (AES-CBC, AES-GCM), which could also improve performance and security.

- Should we default to AES-GCM?
    - Be explicit about the mode, do not hard code it
    - Do not need to add config for it now

Observability

1. Currently, it's not possible to correlate in the logs the sequence of calls that are part of the envelope encryption operation: kube-apiserver -> kms plugin -> external KMS.

a. Including an AuditID in the encrypt/decrypt request. The kms plugin can propagate this to the external KMS.
b. The AuditID can also be used to log an entry in the kms plugin that can be used to correlate encrypt/decrypt operations.

```
type EncryptRequest struct {
   // Version of the KMS plugin API.
   Version string `protobuf:"bytes,1,opt,name=version,proto3"
json:"version,omitempty"`
   // The data to be encrypted.
   Plain []byte `protobuf:"bytes,2,opt,name=plain,proto3" json:"plain,omitempty"`
   // The auditID to be associated with this request.
   AuditID              string    `protobuf:"bytes,3,opt,name=audit_id,proto3"
json:"audit_id,omitempty"`
   XXX_NoUnkeyedLiteral struct{} `json:"-"`
   XXX_unrecognized     []byte   `json:"-"`
   XXX_sizecache        int32    `json:"-"`
}
```

2. Extending the EncryptResponse to include the KEK version used by the kms plugin for encrypting the data.

```
type EncryptResponse struct {
   // The encrypted data.
   Cipher []byte `protobuf:"bytes,1,opt,name=cipher,proto3"
json:"cipher,omitempty"`
   // The KEK version used to encrypt the data.
   KekVersion           string   `protobuf:"bytes,2,opt,name=kek_version,proto3"
json:"kek_version,omitempty"`
   XXX_NoUnkeyedLiteral struct{} `json:"-"`
   XXX_unrecognized     []byte   `json:"-"`
   XXX_sizecache        int32    `json:"-"`
}
```

- Kube-apiserver can log an entry to capture the encrypt operation using the AuditID, etcd key (*optional*), KEK version used for encryption.
- This KEK version should be stored in etcd with the rest of the payload. The API server can retrieve the KEK version and pass it as part of the DecryptRequest. The KMS plugin will now have more information on the KEK to use for decrypting. If the KEK is no longer found, the kms plugin can skip cloud operations and return an error to the API server. This can also be used to indicate unrecoverable errors.

```
type DecryptRequest struct {
```

```
   // Version of the KMS plugin API.
   Version string `protobuf:"bytes,1,opt,name=version,proto3"
json:"version,omitempty"`
   // The data to be decrypted.
   Cipher []byte `protobuf:"bytes,2,opt,name=cipher,proto3"
json:"cipher,omitempty"`
   // The KEK version that was used to encrypt the ciphertext.
   KekVersion string `protobuf:"bytes,3,opt,name=kek_version,proto3"
json:"kek_version,omitempty"`
   // The auditID to be associated with this request.
   AuditID              string    `protobuf:"bytes,3,opt,name=auditID,proto3"
json:"auditID,omitempty"`
   XXX_NoUnkeyedLiteral struct{} `json:"-"`
   XXX_unrecognized     []byte   `json:"-"`
   XXX_sizecache        int32    `json:"-"`
}
```

- This change could also provide an option for the kms plugin to be configured with multiple KEK's for encrypt/decrypt operations.

Recovery

If the KEK used for encryption is deleted, the objects in etcd become unrecoverable. This prevents secret list operations from succeeding. The only way to resolve the issue is by removing the key from etcd.

```
➜ kubectl delete secret secrets-store-creds
Error from server (InternalError): Internal error occurred: rpc error: code =
Unknown desc = failed to decrypt, error: keyvault.BaseClient#Decrypt: Failure
responding to request: StatusCode=403 -- Original Error: autorest/azure: Service
returned an error. Status=403 Code="Forbidden" Message="Operation decrypt is not
allowed on a disabled key." InnerError={"code":"KeyDisabled"}
```

1. In the decrypt response, we could include an error from kms-plugin which indicates the KEK is not recoverable. At this point, the kube-apiserver can force delete the encrypted object in etcd to allow other secret operations to succeed.
    a. Issue with this approach is for intermittent errors this could result in the etcd key being deleted.
2. A more reliable approach is to have a force delete flag as part of "kubectl delete secret" command that can remove the secret from etcd. This is a user initiated operation.

```
kubectl delete secret secrets-store-creds --force-delete
```

References

1. Reduce the number of trips to KMS by caching DEKs
2. Transform from storage concurrently during List operation
3. Improve readiness times for clusters that use envelope encryption and create large numbers of secrets