Secret management with Cloud KMS

Applications often require access to small pieces of sensitive data at build or run time. These pieces of data are often referred to as *secrets*. Secrets are similar in concept to configuration files, but are generally more sensitive, as they may grant access to additional data, such as user data.

This topic describes some of the main concepts of secret management. It also provides guidance on how you can use Cloud Key Management Service for secret management.

Note: Cloud KMS does not directly store secrets. It can encrypt secrets (https://cloud.google.com/kms/docs/encrypt-decrypt) that you store elsewhere. Secret Manager (https://cloud.google.com/secret-manager/) (currently in beta) is the dedicated secret management solution on Google Cloud.

Overview

Several options exist for managing secrets. Some common ways of storing secrets include using:

- Code or binaries
- A deployment manager
- A secret volume in a container
- Metadata of a VM
- A storage system

Note: To manage secrets on Google Cloud, you may also want to consider Secret Manager (https://cloud.google.com/secret-manager/), which is currently in beta. Secret Manager provides a straightforward and convenient way to securely store credentials and other sensitive data.

Choosing from among these options generally requires striking a balance between security and functionality. Typical security concerns include:

- **Authorization**: Access management for secrets or where they are stored, including tight authorization scopes.
- **Verification of usage**: The ability to audit at a low level of granularity (for example, on a per-secret basis), of access to and use of secrets.
- **Encryption at rest**: The encryption of secrets in case of data theft or loss.
- **Rotation**: The ability to rotate or refresh secrets, either regularly or as needed in reaction to an incident.
- **Isolation**: Separation between where secrets are managed versus where they are used. Isolation also applies to separation of duties between users who have the ability to manage secrets versus use secrets.

Typical functionality concerns include:

- **Consistency**: Synchronization of secrets across multiple locations and in multiple applications.
- **Version management**: An understanding of when and how keys are updated, to support secret rotation.

Choosing a secret management solution

Choosing the best secret management solution depends on your unique existing environment, secrets, and security needs.

Some common approaches include:

1. **Storing secrets in code, encrypted with a key from Cloud KMS**. This solution is typically implemented by encrypting secrets at the application layer. Using this solution helps provide an additional layer of protection against insider threats by limiting the scope of access to the secret. Access to the secret is restricted from all developers with access to the code to only those who have both access to the code and the corresponding key. Even in cases where all developers have access to both the code and the key, implementing this option has benefits as it provides the ability to audit access to the secret, which might not be possible in a code repository.

2. **Storing secrets in a storage bucket in Cloud Storage, encrypted at rest**. This solution has similar benefits as the previous solution, in that it limits access to secrets to a smaller set of developers and provides the ability to audit that access. In addition, by storing secrets in a separate location, you can more easily rotate secrets when necessary; for example, if a security breach is detected. Also, this solution allows for separation of systems; if the code repository using the secrets is breached, the secrets themselves might still be protected.
3. **Using a third-party secret management solution**. A dedicated secret management tool builds on the first two options in this list. In addition, these tools might allow you to not only rotate secrets more easily, but, in some cases, either perform that rotation on your behalf, or simplify regular rotation.

Warning: Another common option is to store your secrets directly in code. This option is not recommended. Although this is the simplest solution to implement, it potentially allows anyone with access to your code to also have access to your secrets, leaving you vulnerable to attacks from inside and outside your organization. This vulnerability could, at best, lead to abuse of your data and accounts elsewhere; at worst, it could potentially expose even more data to the attacker.

Changing secrets

Another important consideration when considering a secret management solution is how easy it is to change secrets. For example, hardcoding a secret is often a tempting solution, but this solution makes changing secrets at a later time-consuming and difficult.

When looking at a solution for secret management, consider the following design requirements and how relevant they are to your application:

- **Rotating secrets**. You may want to rotate secrets regularly, especially for security. Ideally, you can store multiple versions of each secret, and have your code try them one at time. By storing many versions of a secret, and rotating to newer secrets as needed, you can better maintain consistency with an external system that may need that secret. This also allows you to roll back to earlier secrets when needed. This solution can be quite complicated to implement, but considering these needs in advance can make it easier to manage secrets over time.
- **Cache secrets locally**. Depending on where you store your secrets with respect to your application, you might need to cache secrets locally. These secrets can then be refreshed frequently, such as several times per hour. The advantage to this solution is that, the more

frequently you refresh, the faster you can respond to an outage. However, the disadvantage is that, if the secret becomes misconfigured somehow, a faster refresh allows that error to spread faster throughout your fleet.

- **Using a separate solution or platform.** When it comes to secret management, you might want to avoid lock-in by taking your secrets to a platform-agnostic secret management solution. This way, you have options should a more flexible solution becomes available.

Secret management using Google Cloud

The Google Cloud offers several ways to help you manage your secrets. This section describes an alternate approach that uses Cloud Storage for secret storage, Cloud KMS for encryption keys, Cloud Identity and Access Management for access control and Cloud Audit Logs for auditing.

Note: This section describes one of the many possible implementations of Google Cloud's infrastructure to protect secrets. It is recommended that you consider the security and functionality needs of your organization, application, and use cases before settling on a specific implementation.

Here's one way to implement using secret management on the Google Cloud. For more information, see How to store secrets encrypted with Cloud KMS (https://cloud.google.com/kms/docs/store-secrets).

- Create two projects. The first project uses Cloud Storage to store secrets. The second project uses Cloud KMS to manage encryption keys.
- Assign `storage.objectAdmin` and `cloudkms.cryptoKeyEncrypterDecrypter` roles to any user that needs to access secrets. Alternatively, you can use a service account that accesses Cloud Storage on a user's behalf. Ensure that users that do not need access to secrets have management, but not access, permissions.
- In Cloud Storage, store each secret as an encrypted object, and group those secrets into buckets as needed. In general, you could group secrets if they share the same usage, access, and protection needs.
- Protect each bucket, by using a unique key in Cloud KMS at the application layer. Another option is to rely on Google's default encryption.

Note: Mapping buckets to the keys used to encrypt them can be difficult. One solution is to ensure that the buckets and keys follow the same naming convention.

- Rotate secrets regularly whenever possible.
- Monitor activity using Cloud Audit Logs. By default, administrative activity logs, such as key rotation or Cloud IAM permission changes, are recorded by default. An additional option to consider is to enable logging for data access logs on Cloud Storage objects. These data access logs are helpful when monitoring secrets that are particularly critical.

This solution addresses the majority of the secret management requirements described in Choosing a secret management solution (#choosing_a_secret_management_solution). One item that is not addressed is version management. That's because addressing version management varies from application to application.

Before you implement a solution like this, you should also consider what encryption option (#encryption_using_cloud_kms) best fits your needs and how you want to manage user access (#managing_access_to_secrets) to secrets.

Encryption options

On the Google Cloud, you have two options for encrypting secrets:

- **Use application layer encryption using a key in Cloud KMS**. With this option, you implement encryption on objects or buckets in Cloud Storage on top of existing Google encryption, using a key stored in Cloud KMS. This is the recommended option.
- **Use the default encryption built into the Cloud Storage bucket**. Google Cloud encrypts customer content stored at rest, using one or more encryption mechanisms. As the name implies, this encryption is available by default and requires no additional action on your part.

To learn about these and other encryption options, see Encryption at Rest (https://cloud.google.com/security/encryption-at-rest/).

Application layer encryption using a key in Cloud KMS

The recommended way to store secrets is to use application layer encryption using a key in Cloud KMS. This method is particularly useful if you are looking for an additional layer of

control, or have a compliance requirement to manage your own keys.

To implement this type of encryption, you send the secrets to be encrypted to Cloud KMS using an `Encrypt` request. Cloud KMS then returns the encrypted secrets, which you can then write to storage.

Note: Try this implementation out using the Getting Started with Cloud KMS (https://codelabs.developers.google.com/codelabs/cloud-encrypt-with-kms/) codelab.

Cloud KMS can handle secrets up to 64 KiB in size. If you need to encrypt larger secrets, it is recommended that you use a key hierarchy, with a locally-generated data encryption key (DEK) to encrypt the secret, and a key encryption key (KEK) in Cloud KMS to encrypt the DEK. To learn more about DEKs, see Envelope Encryption (https://cloud.google.com/kms/docs/data-encryption-keys).

Note: Try a simple file encryption example using the Cloud KMS quickstart (https://cloud.google.com/kms/docs/quickstart).

Default encryption

If using application layer encryption is not an option for your application, another common solution is to use Cloud Storage's default encryption. This option is often used when you are primarily looking for a cloud solution to protect secrets in storage.

This encryption is enabled automatically and requires no additional effort on your part to implement.

Managing access to secrets

There are two main options to restrict and enforce access:

- Access controls on the bucket in which the secret is stored. This option can support multiple secrets (objects) per bucket, or only a single secret per bucket. This is the recommended option.
- Access controls on the key which encrypts the bucket in which the secret is stored. This option supports multiple secrets per key, or only a single secret per key.

You should segregate secrets only when it improves security and ease of use. A common best practice is to limit the amount of data that any one encryption key protects, for cryptographic isolation; or that any one access control list protects. This practice allows you to have greater granular control over secret access, helps prevent accidental permissions, and supports more granular auditing. When you group secrets together, do so when it logically makes sense. For example, you might group some secrets together to simplify control, such as when a single application needs access to a particular collection of secrets at runtime.

When deciding how to store your secrets, it is recommended you use these guidelines:

- Each secret as its own object
- Secrets in a single bucket where they have multiple traits in common
- A single encryption key is used to encrypt each bucket, which includes these logically grouped secrets.

Some scenarios in which you might want to group secrets together include:

- The same application requires secret access
- The same human administrators manage secret versions and access
- The same environment, such as production, development, or test
- The same use time, such as build time or deploy time
- The same desired level of protection

Key rotation

A good practice is to regularly rotate keys to encrypt secrets. Rotating keys helps to limit the amount of data encrypted with a single key, and helps limit the key's lifecycle in the event it becomes compromised. In addition to automatic key rotation, you can also manually rotate a key. For example, you might manually rotate a key when a new version of the secret is updated. To learn more, see Key rotation (https://cloud.google.com/kms/docs/key-rotation).

Secret rotation

In addition to rotating keys (#key_rotation), you can also rotate secrets. A secret is often rotated (or updated) when a new version is created. For example, generating a new password for a database credential. You might also want to rotate keys regularly to limit the secret’s lifecycle.

Secret rotation results in a secret having multiple versions that you must manage. There might be a single version of a secret that is valid at any one time, or multiple versions of secrets. It is recommended that you consider preserving older versions of a secret for a period of time, because they may be needed if the application is rolled back to an earlier version.

One way to manage multiple versions of secrets is to create an object for each version, and store those objects in the same bucket associated with that specific secret. You can then employ a naming convention that you can use to track which version is in use. In addition, you can use a central set of variables to help you determine which secret should be in use at any given time.

Permission management using Cloud Identity and Access Management

As part of secret management on Google Cloud, it is recommended that you use Cloud Identity and Access Management to create and manage permissions for Google Cloud resources. Cloud IAM unifies access control for Google Cloud services into a single system and presents a consistent set of operations. To learn more about Cloud IAM, see the Cloud IAM Documentation (https://cloud.google.com/iam/docs/).

A critical concept in role and access management is the separation of duties. You want to avoid having a single individual who is able to both encrypt and decrypt data, as well as manage or create new keys. For more information, see Separation of Duties (https://cloud.google.com/kms/docs/separation-of-duties).

The two ways of managing permissions are:

- Without a service account. This is the recommended option.
- With a service account

Permission management without a service account

For secret management using Cloud KMS, the ideal configuration should minimize unnecessary access and enforce separation of duties. This type of configuration requires several users:

- An organizational-level administrator. This is a user with the `resourcemanager.organizationAdmin` role. The organizational-level administrator is

typically the business owner of the account. Note that if you used the `resourcemanager.projectCreator` role instead, that user is granted `owner` access on those projects. This level of access is usually unnecessary. It is recommended that you do not use this role for secret management.

- A second user that has the `storage.objectAdmin` role. This user is responsible for managing secrets. This user also uses a service account that has the `storage.admin` role for the project that contains the Cloud Storage bucket. This service account limits this user's ability to edit the bucket metadata, or delete the bucket.
- A third user with the `cloudkms.admin` role. This user manages the keys used to encrypt secrets.
- A fourth user that has both the `storage.objectAdmin` and `cloudkms.cryptoKeyEncrypterDecrypter` roles. This is the end user that needs to access secrets.

Note: These users do not need to be separate individuals. In fact, they could all be the same person. These roles are proposed to allow for a separation of duties between the user who accesses secrets, and the user who manages the keys used to protect those secrets.

Permission management with a service account

An alternative implementation requires that the end user only have permissions on the storage bucket, and have the storage service use a service account to access the key on the user's behalf. This configuration is similar to the one described in Permission management without a service account (#permission_management_without_a_service_account), with the following changes:

- The end user accessing the secrets, has the`storage.objectAdmin` role.
- A fifth user, which is for the Cloud Storage service account, has the `cloudkms.cryptoKeyEncrypterDecrypter` role.

This configuration allows for a situation where no human has access to encrypt and decrypt with a key, which can be preferable in some cases. However, this configuration allows Cloud Storage to encrypt and decrypt with the key on its own authority.

Auditing using Cloud Audit Logs

A final consideration when managing secrets on Google Cloud is using Cloud Audit Logs (https://cloud.google.com/logging/docs/audit/). This service consists of two log streams, Admin Activity and Data Access, which are generated by Google Cloud services. These streams help you answer the question of "who did what, where, and when?" within your Google Cloud projects.

Admin Activity logs contain log entries for API calls or administrative actions that modify the configuration or metadata of a service or project. This log is always enabled and is visible by all project members.

Data Access logs contain log entries for API calls that create, modify, or read user-provided data managed by a service, such as data stored in a database service. Data Access logs are visible only by project owners and users with the Private Logs Viewer role.

In both Cloud Storage and Cloud KMS, Admin Activity logs are on by default; these include actions like creating a new bucket, or rotating a key. Admin activity logs are on by default, and require no action from the user to turn on.

Neither Cloud Storage nor Cloud KMS have data access logs turned on by default, as these could potentially be a significant volume of data. These logs track actions that occur frequently; especially when implementing the solution described here. Examples of these actions include reading the bucket, or encrypting or decrypting with a key. In addition, any secret access requires the use of both Cloud Storage and Cloud KMS, so interactions with secrets could potentially be recorded in either location (although doing both would be redundant).

If you opt to use logging, it is recommended you turn on Data Access logs on objects in Cloud Storage, rather than for keys in Cloud KMS. These logs provide data that is more granular and easier to audit than the data access logs on keys in Cloud KMS. You might also want to turn on Data Access logs on Cloud Storage objects for secrets that are particularly critical.

Last updated January 16, 2020.