Key Management Service

Best Practices

Product Documentation

Contents

Best Practices
Symmetrical Encryption and Decryption
Encrypting/Decrypting Sensitive Data
Overview

Last updated : 2019-11-28 18:55:43

Sensitive information encryption is a core capability of KMS, which is mainly used to protect small pieces of sensitive data (less than 4 KB) such as keys, certificates, and configuration files. A CMK is used to encrypt sensitive data instead of storing it in plaintext. During decryption, the data ciphertext is decrypted to the memory, so that the plaintext does not get stored in the disk. HTTPS requests are used in the entire interaction and transfer process, ensuring the security of sensitive data.

If you need to use KMS for high-performance encryption/decryption of massive amounts of data, please see Envelope Encryption scenario.

Examples of sensitive information

-	Key/Certificate	Backend Configuration File
Usage	Encrypts business data, communication channels, and digital signatures.	Stores system architecture and other business information, such as database IP and password.
Risk of data loss	Confidential information is stolen; encrypted tunnels are monitored; signatures are faked.	Business data is breached and used to attack other systems.

Schematic diagram

In this scenario, sensitive data is encrypted/decrypted through a CMK, which is protected by a third-party certified hardware security module (HSM). The CMK performs encryption/decryption inside the HSM, and any unauthorized party, including Tencent Cloud, has no access to the CMK in plaintext.

Features

- Permission control: Fully integrated with CAM, KMS can control which accounts have access to your CMK through identity and policy management.
- Built-in audit: KMS is integrated with CloudAudit to record all API requests for detailed statistics of key management activities and key usage, ensuring that all data operations can be traced and audited.
- Integrated key management: KMS enables centralized management of keys from various applications.
- Security and compliance: KMS leverages a State Cryptography Administration of China or FIPS-140-2 certified hardware security module (HSM) to generate and protect keys, thereby ensuring their confidentiality, integrity, and availability.
- Sensitive data encryption: KMS supports encryption/decryption of small pieces of sensitive data (less than 4 KB), such as keys, certificates, and configuration files.

Precautions

- Secure storage of `SecretId` and `SecretKey` :
  - Tencent Cloud API authentication mainly relies on `SecretId` and `SecretKey` , which are your unique credentials. Tencent Cloud's service systems need such credentials to call Tencent Cloud APIs.
- Permission control over `SecretId` and `SecretKey` :
  - It is recommended to use a sub-account and manage risks by means of API authorization as needed.
- Plaintext data storage:
  - Data has already encrypted through sensitive data encryption. To ensure data security, please make sure that the original plaintext data is deleted.

Operation Guide

Last updated : 2020-04-13 16:04:01

This operation guide takes Python as an example. Operations in other programming languages can be performed in a similar way.

Preparations

- Dependent environment of the sample code: Python 2.7.
- Activate KMS: you can do so in the Tencent Cloud Console.
- Activate TencentCloud API key service: get the `SecretID` , `SecretKey` , and endpoint. The general format of the endpoint is `*.tencentcloudapi.com` . For example, the endpoint of KMS is `kms.tencentcloudapi.com` . For more information, please see the documentation of the specified product.
- Install the SDK: run the following command. For more information, please see the tencentcloud-sdk-python project on GitHub.

```
pip install tencentcloud-sdk-python
```

Process

You can follow the four steps below to encrypt sensitive data.

1. Create a customer master key (CMK) in the console or through the `CreateKey` API.
2. Call the `Encrypt` API of KMS to encrypt your sensitive data and get the ciphertext.
3. Store the ciphertext data based on your business needs.
4. When reading data, call the `Decrypt` API of KMS to decrypt the ciphertext into plaintext.

Directions

Step 1. Create a CMK

For more information on how to create a CMK, please see Creating a Key.

Step 2. Encrypt the sensitive data

Prerequisite: the CMK created in step 1 is enabled.

In the console

The online tools are suitable for one-time or non-batch encryption and decryption operations, such as the initial generation of key ciphertext. With the online tools, you can focus on your core business without developing tools for non-batch encryption and decryption. For more information, please see Encryption and Decryption.

In the SDK for Python

The `Encrypt` API is used to encrypt up to 4 KB of data, such as database passwords, RSA keys, or other sensitive information. This document describes how to encrypt data through the SDK for Python. You can also use other supported programming languages.

The `KeyId` and `Plaintext` parameters are required for this API. For more information, please see the Encrypt API document.

Encryption in the SDK for Python

The sample code below demonstrates how to use the specified CMK for data encryption.

Python sample code

```
# -*- coding: utf-8 -*-
import base64

from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloudSDKException
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.kms.v20190118 import kms_client, models

def KmsInit(region="ap-guangzhou", secretId="", secretKey=""):
try:
credProfile = credential.Credential(secretId, secretKey)
client = kms_client.KmsClient(credProfile, region)
return client
except TencentCloudSDKException as err:
print(err)
return None

def Encrypt(client, keyId="", plaintext=""):
try:
req = models.EncryptRequest()
req.KeyId = keyId
req.Plaintext = base64.b64encode(plaintext)
rsp = client.Encrypt(req) # Call the `Encrypt` API
```

```
return rsp
except TencentCloudSDKException as err:
print(err)
return None

if __name__ == '__main__':
# User-defined parameters
secretId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
secretKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
region = "ap-guangzhou"
keyId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
plaintext = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

client = KmsInit(region, secretId, secretKey)
rsp = Encrypt(client, keyId, plaintext)
print "plaintext=", plaintext, ", cipher=", rsp.CiphertextBlob
```

Step 3. Store the encrypted data

Store the ciphertext according to the application scenarios of your business.

Step 4. Decrypt the sensitive data

In the console

For more information, please see Encryption and Decryption.

In the SDK for Python

The `Decrypt` API is used to decrypt data.

The `CiphertextBlob` parameter is required for this API. For more information, please see the Decrypt API document.

Python sample code

```
# -*- coding: utf-8 -*-
import base64
from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloudSDKException
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.kms.v20190118 import kms_client, models

def KmsInit(region="ap-guangzhou", secretId="", secretKey=""):
try:
credProfile = credential.Credential(secretId, secretKey)
client = kms_client.KmsClient(credProfile, region)
```

```
return client
except TencentCloudSDKException as err:
print(err)
return None

def Decrypt(client, keyId="", ciphertextBlob=""):
try:
req = models.DecryptRequest()
req.CiphertextBlob = ciphertextBlob
rsp = client.Decrypt(req) # Call the `Decrypt` API
return rsp
except TencentCloudSDKException as err:
print(err)
return None

if __name__ == '__main__':
# User-defined parameters
secretId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
secretKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
region = "ap-guangzhou"
keyId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
ciphertextBlob = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

client = KmsInit(region, secretId, secretKey)
rsp = Decrypt(client, keyId, ciphertextBlob)
print "cipher=", ciphertextBlob, ", base64 decoded plaintext=", base64.b64decode(rsp.Plaintext)
```

Envelope Encryption/Decryption Overview

Last updated : 2019-11-28 18:56:04

Envelope encryption is a high-performance encryption/decryption solution for massive amounts of data. For encryption of large files or performance-sensitive data, use the GenerateDataKey API to generate a data encryption key (DEK). Only the DEK need to be transferred to the KMS server (which are encrypted/decrypted with a CMK), and all data are processed with efficient local symmetric encryption which has little impact on user access.
In actual business scenarios where massive amounts of data needs to be encrypted with high encryption performance needed, a DEK can be generated to encrypt/decrypt local data, which not only meets the requirements for encryption performance, but also enables KMS to keep DEKs random and secure.

Comparison of KMS encryption schemes

Item	Sensitive Data Encryption	Envelope Encryption
Related key	CMK	CMK, DEK
Performance	Symmetric encryption, remote call	Remote symmetric encryption for small amounts of data, and local symmetric encryption for massive amounts of data.
Key scenarios	Keys, certificates, and small data entries; suitable for scenarios with low call frequency	Massive amounts of data; suitable for scenarios with high requirements for encryption performance

Schematic diagram

In this scenario, a CMK generated in KMS, as an important resource, is used to generate and get the DEK plaintext and ciphertext. Based on your actual business needs, you can first encrypt local data through the DEK plaintext in the memory and store the DEK ciphertext and ciphertext data in the disk, then decrypt the DEK ciphertext using KMS when necessary, and finally decrypt the data in the memory using the decrypted DEK plaintext.

Features

- High efficiency: All business data is encrypted using highly efficient local symmetric encryption, which has little impact on the access experience in your business. As for the overhead of DEK creation and encryption/decryption, except in extreme cases, you need to use a "one key at a time" scheme. In most scenarios, the plaintext and ciphertext of one DEK can be reused for a certain period of time, so the overhead is generally small.
- Security and ease of use: The security of envelope encryption is protected with the key security feature of KMS. As DEKs protect business data, and KMS protects DEKs and provides increased availability, your CMK is mainly used to generate DEKs. Only authorized objects can operate on the CMK.

Precautions

- Secure storage of `SecretId` and `SecretKey` :
  - Tencent Cloud API authentication mainly relies on `SecretId` and `SecretKey` , which are your unique credentials. Tencent Cloud's service systems need such credentials to call Tencent Cloud APIs.
- Permission control over `SecretId` and `SecretKey` :
  - It is recommended to use a sub-account and manage risks by means of API authorization as needed.
- Plaintext key processing by the business system:
  - Envelope encryption uses symmetric encryption, so plaintext keys should not be stored in the disk and need to be used in the memory during business processes.
- DEK processing by the backend system:
  - Envelope encryption uses symmetric encryption. You can reuse the same DEK as needed by your business, or use different DEKs for different users and at different times.

Operation Guide

Last updated : 2020-04-13 16:04:01

This operation guide takes Python as an example. Operations in other programming languages can be performed in a similar way.

Preparations

- Dependent environment of the sample code: Python 2.7.
- Activate KMS: you can do so in the Tencent Cloud Console.
- Activate TencentCloud API key service: get the `SecretID`, `SecretKey`, and endpoint. The endpoint of KMS is `kms.tencentcloudapi.com`. For more information, please see the documentation of the specified product.
- Install the SDK: run the following command. For more information, please see the open-source tencentcloud-sdk-python project on GitHub.

```
pip install tencentcloud-sdk-python
```

Process

You can follow the three steps below to complete envelope encryption.

1. Create a CMK.
2. Encrypt data through envelope encryption. Your application calls the KMS `GenerateDataKey` API to generate a DEK, and the system encrypts data with the plaintext key and stores the ciphertext key and ciphertext in the disk.
3. Decrypt data. The system reads the ciphertext key and ciphertext, decrypts the ciphertext key through the `Decrypt` API of KMS, returns the plaintext key, and finally decrypts the ciphertext data with the plaintext key.

Steps

Step 1. Create a CMK

For more information on how to create a CMK, please see Creating a Key.

Step 2. Encrypt data through envelope encryption

If a new DEK is needed (e.g., data needs to be encrypted for new users or the reuse of a DEK exceeds the specified period of time), you can call a KMS API to create a new DEK, then encrypt data with the plaintext key in the memory, and store the ciphertext and ciphertext key in the disk.

Generating a DEK and encrypting your data

The `GenerateDataKey` API is used to generate a DEK, which is a second-level key generated based on a CMK and used for encrypting and decrypting local data. KMS does not store or manage DEKs, which need to be stored by yourself instead.

The examples below are implemented in the Tencent Cloud SDK for Python, which can also be implemented in other supported programming languages.

The `KeyId` parameter is required for this API. For more information, please see the GenerateDataKey API document.

Example in the SDK for Python

```
# -*- coding: utf-8 -*-
import base64
from Crypto.Cipher import AES
from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloudSDKException
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.kms.v20190118 import kms_client, models

def KmsInit(region="ap-guangzhou", secretId="", secretKey=""):
try:
credProfile = credential.Credential(secretId, secretKey)
client = kms_client.KmsClient(credProfile, region)
return client
except TencentCloudSDKException as err:
print(err)
return None

def GenerateDatakey(client, keyId, keyspec='AES_128'):
try:
req = models.GenerateDataKeyRequest()
req.KeyId = keyId
req.KeySpec = keyspec
# Call the `GenerateDataKey` API
generatedatakeyResp = client.GenerateDataKey(req)
# The plaintext key needs to be used in the memory, while the ciphertext key is used for persiste
nt storage
```

```
print "DEK cipher=", generatedatakeyResp.CiphertextBlob
return generatedatakeyResp
except TencentCloudSDKException as err:
print(err)

def AddTo16(value):
while len(value) % 16 != 0:
value += '¥0'
return str.encode(value)

# User-defined logic. The example here is for reference only
def LocalEncrypt(dataKey="", plaintext=""):
aes = AES.new(base64.b64decode(dataKey), AES.MODE_ECB)
encryptedData = aes.encrypt(AddTo16(plaintext))
ciphertext = base64.b64encode(encryptedData)
print "plaintext=", plaintext, ", cipher=", ciphertext

if __name__ == '__main__':
# User-defined parameters
secretId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
secretKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
region = "ap-guangzhou"
keyId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
keySpec = "AES_256"
plaintext = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

client = KmsInit(region, secretId, secretKey)
rsp = GenerateDatakey(client, keyId, keySpec)

LocalEncrypt(rsp.Plaintext, plaintext)
```

Step 3. Decrypt data

Read the ciphertext key stored in the disk, call the `Decrypt` API to decrypt the ciphertext key, and then decrypt data through the decrypted plaintext key.

Decrypting (in KMS SDK for Python)

The `Decrypt` API is used to decrypt data.

The examples below are called with the Tencent Cloud SDK for Python, which can also be called with any supported programming languages.

The `CiphertextBlob` parameter is required for this API. For more information, please see the Decrypt API document.

Example in the SDK for Python

Decrypt the DEK ciphertext key by calling the KMS `Decrypt` API, and then use the obtained DEK plaintext to decrypt the ciphertext data.

```
# -*- coding: utf-8 -*-
import base64
from Crypto.Cipher import AES
from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloudSDKException
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.kms.v20190118 import kms_client, models

def KmsInit(region="ap-guangzhou", secretId="", secretKey=""):
try:
credProfile = credential.Credential(secretId, secretKey)
client = kms_client.KmsClient(credProfile, region)
return client
except TencentCloudSDKException as err:
print(err)
return None

def DecryptDataKey(client, ciphertextBlob):
try:
req = models.DecryptRequest()
req.CiphertextBlob = ciphertextBlob
rsp = client.Decrypt(req) # Call the `Decrypt` API to decrypt the DEK
return rsp
except TencentCloudSDKException as err:
print(err)

# User-defined logic. The example here is for reference only
def LocalDecrypt(dataKey="", ciphertext=""):
aes = AES.new(base64.b64decode(dataKey), AES.MODE_ECB)
decryptedData = aes.decrypt(base64.b64decode(ciphertext))
plaintext = str(decryptedData)
print "plaintext=", plaintext, ", cipher=", ciphertext

if __name__ == '__main__':
# User-defined parameters
secretId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
secretKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
region = "ap-guangzhou"
dekCipherBlob="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
ciphertext="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

client = KmsInit(region, secretId, secretKey)
rsp = DecryptDataKey(client, dekCipherBlob)
```

```
LocalDecrypt(rsp.Plaintext, ciphertext)
```

Asymmetric Encryption and Decryption Overview

Last updated : 2020-04-13 10:55:06

Asymmetric encryption provides a pair of keys: a public key and a private key. Content encrypted with a public key cannot be decrypted with it but the corresponding private key. In data communication scenarios, asymmetric encryption is securer than symmetric encryption, and it is usually used between systems with different trust levels. For example, in scenarios where sensitive data in small size need to be transferred, you can consider using asymmetric keys for data encryption.

Asymmetric Key Types

KMS supports the following types of asymmetric encryption algorithms:

RSA

Currently, KMS supports RSA keys with a modulus of 2,048 bits (KeyUsage = ASYMMETRIC_DECRYPT_RSA_2048).

SM2

SM2 is a public-key algorithm that meets the standards issued by the State Cryptography Administration (SCA) of China. It is used to replace the RSA algorithm in China's commercial cryptography system. You can consider using this type of keys for applications with requirements for compliance with SCA standards (KeyUsage = ASYMMETRIC_DECRYPT_SM2).

Typical Scenarios of Asymmetric Encryption

Asymmetric encryption is widely used in secret communication scenarios and has three major participants: information sender, information recipient, and transmission medium. The encryption process mainly includes the following steps:

1. The information recipient creates a public key-private key pair and sends the public key to one or multiple information senders.
2. The information sender uses the public key to encrypt the sensitive information and sends the encrypted ciphertext to the information recipient through a transmission medium.

3. After getting the data from the transmission medium, the information recipient uses the private key to decrypt the data and restore the original information.

Ciphertext can be decrypted only with a confidential private key, therefore, even if information leakage occurs due to low security of the transmission medium, those who get the ciphertext still cannot decrypt it, which ensures the security of sensitive information. This encrypted data transfer method is usually used in key exchange scenarios.

Because of the characteristics of use cases of the public key-private key pair, KMS does not support the automatic rotation of asymmetric CMKs. If you need to update the used keys regularly or from time to time, you can create new asymmetric keys.

Operations Guide

Last updated : 2020-04-13 10:55:06

Process

If you need to encrypt sensitive information before transferring it (in scenarios such as key exchange), you can use the asymmetric key-based encryption and decryption scheme. As an information recipient, you need to perform the following operations:

1. Create an asymmetric key on KMS. For more information.
2. Get the public key on KMS. For more information, please see the GetPublicKey API documentation.
3. The information recipient distributes the public key to the information sender.
4. The information sender uses the obtained public key to encrypt the sensitive information locally and sends the ciphertext to the information recipient.
5. The information recipient calls the KMS decryption API to decrypt the ciphertext after getting it. For more information on the API, please see AsymmetricSm2Decrypt and AsymmetricRsaDecrypt. For more information on the TCCLI method, please see Asymmetric Key Decryption.

Ciphertext is transferred throughout the entire sensitive data transfer process, and the only key that can decrypt the ciphertext is managed and protected by KMS, which cannot be obtained by other people including Tencent Cloud. This scheme greatly improves the security of encrypted sensitive data transfer.

Directions

RSA sample

1. Create an asymmetric key

```
tccli kms CreateKey --Alias test --KeyUsage ASYMMETRIC_DECRYPT_RSA_2048
```

Returned result:

```
{
"Response": {
"KeyId": "22d79428-61d9-11ea-a3c8-525400******",
"Alias": "test",
"CreateTime": 1583739580,
"Description": "",
"KeyState": "Enabled",
```



```
"KeyUsage": "ASYMMETRIC_DECRYPT_RSA_2048",
"RequestId": "0e3c62db-a408-406a-af27-dd5ced******"
}
}
```

2. Download the public key

Request:

```
tccli kms GetPublicKey --KeyId 22d79428-61d9-11ea-a3c8-525400******
```

Returned result:


```
{
"Response": {
"RequestId": "408fa858-cd6d-4011-b8a0-653805******",
"KeyId": "22d79428-61d9-11ea-a3c8-525400******",
"PublicKey": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeUc5a09TfiDplI04WovBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFiTx3087wdKWcF2vHL9Ja+95VuCmKYeK1uhPyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCxDhwj08b24T+/FYB9a4icuqQypCvjY1X9j8ivAsPEdHZoc9Di7JXBTZdVeZC1igCVgl6mwzdHTJCRydE2976zyjC7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6RbxebLqtmNypNERIR7jTCt9L+fgYOX7anmuF5v7z0GfFsen9Tqb1LsZuQR0vgqCau0j************",
"PublicKeyPem": "-----BEGIN PUBLIC KEY-----¥nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeU¥nc5a09TfiDplI04WovBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFi¥nTx3087wdKWcF2vHL9Ja+95VuCmKYeK1uhPyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCx¥nDhwj08b24T+/FYB9a4icuqQypCvjY1X9j8ivAsPEdHZoc9Di7JXBTZdVeZC1igCV¥ngl6mwzdHTJCRydE2976zyjC7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6RbxebLq¥ntmNypNERIR7jTCt9L+fgYOX7anmuF5v7z0GfFsen9Tqb1LsZuQR0************¥n1QIDAQAB¥n-----END PUBLIC KEY-----¥n"
}
}
```


3. Use the public key for encryption

   i. Store the public key `PublicKey` in the file `public_key.base64` and Base64-decode it.

   Store it in the file:

```
echo "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeUc5a09TfiDplI04WovBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFiTx3087wdKWcF2vHL9Ja+95VuCmKYeK1uhPyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCxDhwj08b24T+/FYB9a4icuqQypCvjY1X9j8ivAsPEdHZoc9Di7JXBTZdVeZC1igCVgl6mwzdHTJCRydE2976zyjC7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6RbxebLqtmNypNERIR7jTCt9L+fgYOX7anmuF5v7z0GfFsen9Tqb1LsZuQR0vgqCau0j************" > public_key.base64
```

   Base64-decode the public key to get its content:

```
openssl enc -d -base64 -A -in public_key.base64 -out public_key.bin
```

   ii. Create a testing plaintext file:

```
echo "test" > test_rsa.txt
```

iii. Use OpenSSL to encrypt the file `test_rsa.txt` with the public key.

```
openssl pkeyutl -in test_rsa.txt -out encrypted.bin -inkey public_key.bin -keyform DER -pubin -encrypt -pkeyopt rsa_padding_mode:oaep -pkeyopt rsa_oaep_md:sha256
```

iv. Base64-encode the data encrypted with the public key for transfer.

```
openssl enc -e -base64 -A -in encrypted.bin -out encrypted.base64
```

4. Use the private key on KMS for decryption

   Use the above-mentioned Base64-encoded ciphertext `encrypted.base64` as the `Ciphertext` parameter for `AsymmetricRsaDecrypt` to decrypt the ciphertext with the private key.

   Request:

```
tccli kms AsymmetricRsaDecrypt --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm RSAES_OAEP_SHA_256 --Ciphertext "DEb/JBmuhVkYS34r0pR7Gv1WTc4khkxqf7S1WIr7/GXsAs/tfP/v/2+1SwsIG7BqW7kUZqr38/FGkaIEqYeewot37t3+Jx0t5w7/yXkUnyUfyfPpXlHXf94g3wF0jijEWWsjWWzaXTkTr8uW0fRBenq+bcaY783FIy03XjJW/Y0wKWjD3tULvKndCJ0/3bkb65kn1Fbsfm20xrUUwqV/p2DVLXBdG1ymr0DjsbG7R0tb3ytc2LmH33YPAQE32eP27ciKzSml+w2tdUM3dw3nEZcTGMs1wFDGk001WB052jZ7TitUD9zCftFv2dKlZD3LRx1+vHqpNVgPhLmL******=="
```

   Returned result:

```
{
"Response": {
"RequestId": "6758cbf5-5e21-4c37-a2cf-8d47f5******",
"KeyId": "22d79428-61d9-11ea-a3c8-525400******",
"Plaintext": "dGVzdAo="
}
}
```

> The process of using SM2 asymmetric keys for encryption and decryption is similar to this example. For more information on the private key-based decryption API, please see AsymmetricSm2Decrypt.

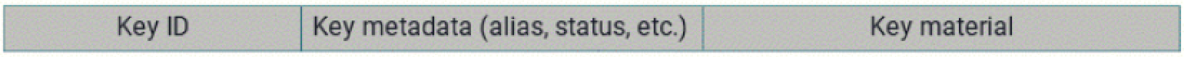
Importing External Key

Overview

Last updated : 2020-04-27 14:37:06

A customer master key (CMK) is a basic element of the KMS service. The CMK contains key ID, key metadata (alias, description, status, etc.), and key material used to encrypt and decrypt data.

By default, the underlying encryptor of KMS creates secure key material for a CMK when the CMK is created in KMS. If you want to use your own key material, i.e., implementing a Bring Your Own Key (BYOK) solution, you can use KMS to generate a CMK with the key material left empty, and then import your own key material into the CMK to form an external CMK. The external CMK can be distributed and managed by KMS.



CMK schematic diagram

Features

- KMS allows you to use your own key material to encrypt and decrypt sensitive data by implementing a Bring Your Own Key (BYOK) solution in Tencent Cloud.
- KMS gives you full control over the key services used in Tencent Cloud, including importing and deleting key material as needed.
- You can back up your key material in local key management infrastructure as an additional disaster recovery measure for KMS.
- You can use your own key material for encryption and decryption operations in the cloud to meet your industry-specific compliance requirements.

Notes

- You need to ensure the security of the key material:
  - When using the key importing feature, you need to ensure that the random material generation source is secure and reliable. Currently, the SM-CRYPTO edition of KMS only supports importing

128-bit symmetric keys, while the FIPS-compliant edition only supports importing 256-bit symmetric keys.

- You need to ensure the availability of the key material:
  - KMS provides high availability of its own services and the capability for restoring from backups, but the availability of your key material is your responsibility. It is strongly recommended that you keep the original backup of the key material in a safe and reliable way, so that if the key material is deleted accidentally or expired, the backup can be imported into KMS timely.
- You need to ensure the correctness of the key importing operations:
  - Once the key material is imported into an external CMK, the two will be associated permanently, i.e., other key materials cannot be imported into this CMK. If this CMK is used for data encryption, the encrypted data can only be decrypted with the CMK used for encryption (i.e., the CMK metadata and key material should match those of the imported key); otherwise, decryption would fail. Please be cautious when deleting key materials and CMKs.
- You need to pay attention to the key importing status:
  Keys in "Pending Import" status are actually enabled keys and incur fees.

Operation Guide

Last updated : 2019-12-12 15:18:20

Process

You can follow the four steps below to create an external CMK.

1. Create a CMK whose source is "external" in the console or through the API, i.e., creating an external CMK.
2. Call an API to get the parameters of the material to be imported into a CMK, including a public key used to encrypt the key material and an import token.
3. Use an encryptor or other secure encryption measures to encrypt your key material locally with the public key obtained in step 2.
4. Call an API to import the encrypted key material and the import token obtained in step 2 into the external CMK.

Directions

Step 1. Create an external CMK

You can create an external CMK in the console or through the API.

- **Via the console**
  (1). Log in to the KMS Console.
  (2). Select the region where you want to create a key and click **Create**.
  (3). In the "Create Key" window, enter the key name and select "External" for key material source, read the document on the methods of importing external key materials and the precautions, and check the box.
  (4). Click **OK** to create the external CMK. You can view the created CMK in the console, where the "Key Source" is displayed as "External".

- **Via the API**
  Below is an example using Tencent Cloud TCCLI, which can be called with any supported programming language.
  When requesting the CreateKey API, set the `Type` parameter to `2` by running the following command:


```
tccli kms CreateKey --Alias <alias> --Type 2
```

Sample source code of the `CreateKey` API:

```
def create_external_key(client, alias):
"""
Generate a BYOK key,
:param Type = 2
"""
try:
req = models.CreateKeyRequest()
req.Alias = alias
req.Type = 2
rsp = client.CreateKey(req)
return rsp, None
except TencentCloudSDKException as err:
return None, err
```

Step 2. Get the parameters of the material to be imported into a CMK

To ensure the security of your key material, you need to encrypt your key material before importing it. You can get its parameters through an API, including a public key used to encrypt the key material and an import token.

Run the following command on TCCLI:

```
tccli kms GetParametersForImport --KeyId <keyid> --WrappingAlgorithm RSAES_PKCS1_V1_5 --WrappingKeySpec RSA_2048
```

Sample source code of the `GetParametersForImport` function:

```
def get_parameters_for_import(client, keyid):
"""
Get the parameters of the material to be imported into a CMK,
of which the returned `Token` is a parameter that executes the `ImportKeyMaterial` function,
and the returned `PublicKey` is used to encrypt the key material.
The `Token` and `PublicKey` will expire in 24 hours. After that, you need to call the API again to get new `Token` and `PublicKey`.
`WrappingAlgorithm ` is used to specify the algorithm for key material encryption. Currently, `RSAES_PKCS1_V1_5`, `RSAES_OAEP_SHA_1`, and `RSAES_OAEP_SHA_256` are supported.
`WrappingKeySpec` is used to specify the type of key material encryption. Currently, only `RSA_2048` is supported.
"""
try:
```

```
req = models.GetParametersForImportRequest()
req.KeyId = keyid
req.WrappingAlgorithm = 'RSAES_PKCS1_V1_5' # RSAES_PKCS1_V1_5 | RSAES_OAEP_SHA_1 | RSAES_OAEP_SHA
_256
req.WrappingKeySpec = 'RSA_2048' # RSA_2048
rsp = self.client.GetParametersForImport(req)
return rsp, None
except TencentCloudSDKException as err:
return None, err
```

Step 3. Encrypt your key material locally

Use the encryption public key obtained in step 2 to encrypt your key material locally. The encryption public key is a 2,048-bit RSA public key, and the encryption algorithm used should be the same as specified for getting the parameters of the key material. As the encryption public key returned by the API is Based64-encoded, you need to Base64-decode it before using it. Currently, algorithms supported by KMS include `RSAES_OAEP_SHA_1` , `RSAES_OAEP_SHA_256` , and `RSAES_PKCS1_V1_5` .

Below is an example of encrypting the key material using OpenSSL. In actual use, it is recommended to encrypt your key material using an encryptor or other secure encryption measures.

(1). Call the GetParametersForImport API to get the `Token` and `PublicKey` , and write the `PublicKey` into the `public_key.base64` file.
(2). Generate a random number using OpenSSL.

```
openssl rand -out raw_material.bin 16
```

You can also use the GenerateRandom API to generate a random number for Base64-decoding.

> The length of a SM-CRYPTO key material must be 128 bits, while that of a FIPS-compliant one must be 256 bits.

(3). Decode the public key.

```
openssl enc -d -base64 -A -in public_key.base64 -out public_key.bin
```

(4). Use the public key to encrypt the key material.

```
# The command line corresponding to `RSAES_OAEP_SHA_1` is as follows:
openssl pkeyutl -in raw_material.bin -out encrypted_key_material.bin -inkey public_key.bin -keyfo
rm DER -pubin -encrypt -pkeyopt rsa_padding_mode:oaep -pkeyopt rsa_oaep_md:sha1
```

```
# The command line corresponding to `RSAES_PKCS1_V1_5` is as follows:
openssl pkeyutl -in raw_material.bin -out encrypted_key_material.bin -inkey public_key.bin -keyfo
rm DER -pubin -encrypt -pkeyopt rsa_padding_mode:pkcs1

# The command line corresponding to `RSAES_OAEP_SHA_256` is as follows:
openssl pkeyutl -in raw_material.bin -out encrypted_key_material.bin -inkey public_key.bin -keyfo
rm DER -pubin -encrypt -pkeyopt rsa_padding_mode:oaep -pkeyopt rsa_oaep_md:sha256
```

(5). Import the encoded ciphertext into KMS as a parameter.

```
openssl enc -e -base64 -A -in encrypted_key_material.bin -out encrypted_material.base64
```

Import the final output `encrypted_material.base64` into KMS as `EncryptedKeyMaterial` .

Step 4. Import the key material

Call an API to import the encrypted key material and the import token obtained in step 2 into the external CMK created in step 1.

- The import token and the public key for key material encryption are bound, and a token can only be used to import key material for the CMK specified when it was generated. The import token is valid for 24 hours and can be reused within its validity period. If it expires, you need to get a new token and encryption public key.
- If the GetParametersForImport API is called multiple times to get the key material, only the token and publicKey obtained from the last call will be valid, while those returned from previous calls will expire automatically.
- You can import key material into an external key where no key materials have ever been imported, reimport key material that has expired or been deleted, or reset the expiration time of key material.

Make a request to import key material through the ImportKeyMaterial API. Below is a sample command:

```
tccli kms ImportKeyMaterial --EncryptedKeyMaterial <material> --ImportToken <token> --KeyId <keyi
d>
```

Sample source code of the `ImportKeyMaterial` function:

```
def import_key_material(client, material, token, keyid):
try:
req = models.ImportKeyMaterialRequest()
req.EncryptedKeyMaterial = material
req.ImportToken = token
req.KeyId = keyid
```

```
rsp = client.ImportKeyMaterial(req)
return rsp, None
except TencentCloudSDKException as err:
return None, err
```

At this point, the external CMK has been imported. You can use it just like an ordinary key.

More Operations

Deleting an external CMK

Deleting an external CMK involves two kinds of operations: deleting the CMK at the scheduled time, and deleting the key material, which will lead to different results.

Deleting a CMK at the scheduled time

The schedule deletion feature can be used to delete an external CMK and has a mandatory waiting period of 7-30 days, after which the external key will be deleted. Please note that once deleted, the CMK cannot be recovered, and the data encrypted with it cannot be decrypted.

Deleting key material

You can delete key material in two ways. If the key material expires or is deleted, the external CMK can no longer be used, and the data encrypted with the CMK can no longer be decrypted, unless you import the same key material into the CMK again.

- You can call the DeleteImportedKeyMaterial API to delete the key material. After the key material is deleted, the key status will become `PendingImport` .
- In an ImportKeyMaterial API call, set the expiration time using the `ValidTo` input parameter, and KMS will automatically delete the key material upon expiration.

> Waiting for the key material to become invalid upon expiration and deleting it manually have the same effect.

Delete the key material by running the following command:

```
tccli DeleteImportedKeyMaterial --KeyId <keyid>
```

Sample source code of the `DeleteImportedKeyMaterial` function:


```
def delete_key_material(client, keyid):
try:
req = models.DeleteImportedKeyMaterialRequest()
req.KeyId = keyid
rsp = client.DeleteImportedKeyMaterial(req)
return rsp, None
except TencentCloudSDKException as err:
return None, err
```

- Once the key material is imported into an external CMK, the two will be associated permanently, i.e., other key materials cannot be imported into this CMK. In other words, after the key material is deleted, if you need to import key material into the CMK again, you need to make sure that the key material to be imported is exactly the same as the deleted one; otherwise, the import will fail.
- If a CMK is used for data encryption, the encrypted data can only be decrypted with the CMK used for encryption (i.e., the CMK metadata and key material should match the imported key material); otherwise, decryption would fail. Please be cautious when deleting key materials and CMKs.