

# Key Management Service

## Practical Tutorial

### Product Documentation



## Copyright Notice

©2013–2026 Tencent Cloud. All rights reserved.

Copyright in this document is exclusively owned by Tencent Cloud. You must not reproduce, modify, copy or distribute in any way, in whole or in part, the contents of this document without Tencent Cloud's the prior written consent.

## Trademark Notice

 Tencent Cloud

All trademarks associated with Tencent Cloud and its services are owned by the Tencent corporate group, including its parent, subsidiaries and affiliated companies, as the case may be. Trademarks of third parties referred to in this document are owned by their respective proprietors.

## Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

# Contents

## Practical Tutorial

### Symmetrical Encryption and Decryption

#### Encrypting/Decrypting Sensitive Data

##### Overview

##### Operation Guide

#### Envelope Encryption/Decryption

##### Overview

##### Operation Guide

### Asymmetric Encryption and Decryption

#### Overview

#### Asymmetric Data Encryption and Decryption

#### Asymmetric Signature Verification

##### Overview

##### SM2 Signature Verification

##### RSA Signature Verification

##### ECC Signature Verification

### Post-Quantum Cryptography Practice In KMS

#### Importing External Key

##### Overview

##### Operation Guide

### Implementing Exponential Backoff to Deal with Service Frequency

### Cloud Product Integration with KMS for Transparent Encryption

# Practical Tutorial

## Symmetrical Encryption and Decryption

### Encrypting/Decrypting Sensitive Data

## Overview

Last updated: 2024-01-11 16:31:21

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: 2024-01-11 16:31:21

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
```





# Envelope Encryption/Decryption

## Overview

Last updated: 2024-01-11 16:31:21

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: 2024-01-11 16:31:21

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 persistent storage
```



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 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    secretKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    region = "ap-guangzhou"
    dekCipherBlob="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    ciphertext="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

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

    LocalDecrypt(rsp.Plaintext, ciphertext)
```

# Asymmetric Encryption and Decryption

## Overview

Last updated: 2024-01-11 16:31:22

A **public key** and a **private key** are required for asymmetric encryption and decryption. These two are a pair of bidirectional keys in cryptography, that is, the public key and private key both can be used for encryption. If one is used for encryption, the decryption can only be performed using another key. The public key can be disclosed to anyone even an unreliable party, but the private key must be kept confidential.

Compared to symmetric encryption, asymmetric encryption does not require a reliable channel for key distribution, so that it is usually applied in communications between systems with different trust levels for encrypted transfer of sensitive data and digital signature verification.

### Asymmetric Key Types

Tencent Cloud KMS currently supports the three asymmetric key algorithms below:

#### 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).

#### ECC

Elliptic Curve Cryptography (ECC) is an encryption algorithm based on mathematical elliptic curves (KeyUsage = ASYMMETRIC\_SIGN\_VERIFY\_ECC).

### Typical Scenarios of Asymmetric Encryption

There are two typical scenarios of asymmetric encryption and decryption in actual use cases, namely the **encrypted communication** and **digital signature**:

#### Encrypted communication

Encrypted communication is a typical application of asymmetric encryption algorithm, of which the process is similar to symmetric encryption with the difference being that the public key is required for encryption and the private key is required for decryption.

How the encrypted communication works:

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.

Tencent Cloud KMS offers solutions for encrypted communication. For more information, please see [Asymmetric Data Encryption and Decryption](#).

## Digital signature

Digital signature is another typical application of asymmetric encryption algorithm, which consists of signature generation and signature verification two processes. The private key is used for signature generation and the public key is used for signature verification, however, the implementation process of encrypted communication is in contrast.

How the digital signature works:

1. The information sender creates a public key–private key pair and sends the public key to one or multiple information recipients.
2. The information sender uses the Hash function to generate a message abstract from the original message, and then uses its private key to encrypt the abstract to get the digital signature of the original message.
3. The information sender transmits the original message and digital signature to the information recipient.
4. After receiving the original message and digital signature, the information recipient uses the same Hash function to generate the abstract A from the original message and uses the public key given by the information sender to decrypt the digital signature to get the abstract B, and then compares the two abstracts to check whether they are identical and the original data is tampered with.

The signature is unique as it is generated and encrypted with a confidential private key. Digital signatures can guarantee confidential data transmission, the correctness of information senders, and the non–repudiation of transactions.

Tencent Cloud KMS offers solutions for digital signatures. For more information, please see [Asymmetric Signature Verification](#).

### Note:

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.

# Asymmetric Data Encryption and Decryption

Last updated: 2024-01-11 16:31:21

## Operation 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 encryption key on KMS. For more information, please see [CreateKey](#).
2. Get the public key on KMS. For more information, please see [GetPublicKey](#).
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 data locally and sends the ciphertext to the information recipient.
5. The information recipient calls the KMS decryption API to decrypt the ciphertext. For more information on the API, please see [AsymmetricSm2Decrypt](#) and [AsymmetricRsaDecrypt](#). For operations using TCCLI, 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.

## Operation Directions

### RSA sample

1. Create an asymmetric encryption key

Request:

```
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":
    "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeUc5a09
    TfiDplIO4WovBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFiTx3O87wdKWc
    F2vHL9Ja+95VuCmKYeK1uhPyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCxDhwj08b24T+/FYB9a
    4icucQypCvjY1X9j8ivAsPEdHZoc9Di7JXBTzdVeZC1igCVg16mwzdHTJCRydE2976zyjC
    7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6RbxebLqtmNypNERIR7jTct9L+fgYOX7anmuF
    5v7z0GfFsen9Tqb1LsZuQR0vgqCauOj*****",
    "PublicKeyPem": "-----BEGIN PUBLIC KEY-----
    \nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeU\nc5
    a09TfiDplIO4WovBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFi\nTx3O87
    wdKWcF2vHL9Ja+95VuCmKYeK1uhPyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCx\nDhwj08b24T
    +/FYB9a4icucQypCvjY1X9j8ivAsPEdHZoc9Di7JXBTzdVeZC1igCV\nngl6mwzdHTJCRyd
    E2976zyjC7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6RbxebLq\ntmNypNERIR7jTct9L+
    fgYOX7anmuF5v7z0GfFsen9Tqb1LsZuQR0*****\n1QIDAQAB\n-----END
    PUBLIC KEY-----\n"
  }
}

```

## 3. Use the public key for encryption.

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

```
echo
"MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeUc
5aO9TfiDplIO4WovBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFiTx3
O87wdKWcF2vHL9Ja+95VuCmKYeK1uhPyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCxDhwj0
8b24T+/FYB9a4icucQypCvjY1X9j8ivAsPEdHZoc9Di7JXBTZdVeZC1igCVgl6mwzd
HTJCRyde2976zyjC7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6RbxebLqtmNypNERI
R7jTct9L+fgYOX7anmuF5v7z0GfFsen9Tqb1LsZuQR0vgqCauOj*****" >
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
```

- 3.2 Create a testing plaintext file.

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

- 3.3 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
```

- 3.4 Base64–encode the data encrypted with the public key for transmission.

```
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+1SwsIG7BqW7kUZ
qr38/FGkaIEqYeewot37t3+Jx0t5w7/yXkUnyUfyfPpXlHXf94g3wFOjijEWwsjWWzaXTk
Tr8uWOfRBEnq+bcaY783FIy03XjJW/Y0wKWjD3tULvKndCJO/3bkb65kn1Fbsfm20xrUUw
qV/p2DVLXBdG1ymr0DjsbG7R0tb3ytc2LmH33YPAQE32eP27ciKzSml+w2tdUM3dw3nEZc
TGMs1wFDGk001WB052jZ7TitUD9zCftFv2dKlZD3LRx1+vHqpNVgPhLmL*****=="
```

Returned result:

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

**Note:**

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](#).

# Asymmetric Signature Verification

## Overview

Last updated: 2024-01-11 16:31:21

In sensitive information transmission, the information sender can provide the identity certification through asymmetric signature verification. The operation process is as follows:

1. Create a pair of asymmetric keys in the KMS console. For more information, please see [CreateKey](#).
2. The information sender uses the created private key to generate a signature for the data to be transmitted. For more information, please see [SignByAsymmetricKey](#).
3. The information sender transmits the signature and data to the information recipient.
4. After receiving the signature and data, the information recipient verifies the signature by one of the two methods below:
  - ① Call the KMS signature verification API to verify the signature. For more information, please see [VerifyByAsymmetricKey](#).
  - ② Download the KMS public asymmetric key, and then locally verify the signature using GmSSL, OpenSSL, password library, KMS SM-CRYPTO Encryption SDK, or any other tools.

**Note:**

Asymmetric signature verification currently supports SM2, RSA, and ECC algorithms.

# SM2 Signature Verification

Last updated: 2025-08-19 11:57:44

This document describes how to use the SM2 signature verification algorithm.

## Operation Directions

### Step 1: Creating an asymmetric signature key

#### Note:

To use the signature feature, the correct key purpose `KeyUsage= ASYMMETRIC_SIGN_VERIFY_SM2` is required when calling the KMS [CreateKey](#) API to create a CMK.

Request:

```
tccli kms CreateKey --Alias test --KeyUsage ASYMMETRIC_SIGN_VERIFY_SM2
```

Returned result:

```
{
  "Response": {
    "KeyId": "22d79428-61d9-11ea-a3c8-525400*****",
    "Alias": "test",
    "CreateTime": 1583739580,
    "Description": "",
    "KeyState": "Enabled",
    "KeyUsage": "ASYMMETRIC_SIGN_VERIFY_SM2",
    "TagCode": 0,
    "TagMsg": "",
    "RequestId": "0e3c62db-a408-406a-af27-dd5ced*****"
  }
}
```

### Step 2: Downloading 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":
"MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLLge0vtct949CwtadHODzISgXJahujq+Pv
M*****bBs/f3axWbvGvHx8Jmqw==",
    "PublicKeyPem": "-----BEGIN PUBLIC KEY-----
\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLLge0vtct949CwtadHODzISgXJa\nhujq
+PvM*****bBs/f3axWbvGvHx8Jmqw==\n-----END PUBLIC KEY-----\n"
  }
}
```

Convert the public key `PublicKeyPem` into the PEM format and save it in the file `public_key.pem` :

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLLge0vtct949CwtadHODzISgXJa
hujq+PvM*****bBs/f3axWbvGvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

#### Note:

You can also log in to the [KMS console](#), click **Customer Managed CMK** on the left sidebar, click a key ID/name in the key list to view the key information, and download the public asymmetric key.

### Step 3: Creating the plaintext file

Create the testing plaintext file:

```
echo "test" > test_verify.txt
```

#### Note:

If there are any invisible characters such as line breaks in the generated content, you need to truncate the file (`truncate -s -1 test_verify.txt`) to provide a correct signature.

### Step 4: Calculating the message abstract

- If the message to be generated a signature for is not longer than 4,096 bytes, you can skip this step to [Step 5](#).
- If the message to be generated a signature for is longer than 4,096 bytes, you need to calculate a message abstract locally first.

Use GmSSL to calculate the message abstract for `test_verity.txt` :

```
gmssl sm2utl -dgst -in ./test_verify.txt -pubin -inkey
./public_key.pem -id 1234567812345678 > digest.bin
```

## Step 5: Calling the KMS signature API to generate a signature

Call the KMS [SignByAsymmetricKey](#) API to calculate the signature.

1. Base64-encode the original message or message abstract before signature calculation.

```
// Base64-encode the message abstract
gmssl enc -e -base64 -A -in digest.bin -out encoded.base64
// Base64-encode the original message
gmssl enc -e -base64 -A -in test_verify.txt -out encoded.base64
```

2. Calculate the signature.

Request:

```
// Generate the signature for the message abstract using the content
of the file `encoded.base64` as the `Message` parameter of
`SignByAsymmetricKey`.
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-
525400***** --Algorithm SM2DSA --Message
"qJQj83hSyOuU7Tn0SRReGck4yuuVWaeZ44BP*****==" --MessageType DIGEST

// Generate the signature for the Base64-encoded original message
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-
525400***** --Algorithm SM2DSA --Message "dG***Ao=" --MessageType RAW
```

Returned result:

```
{
  "Response": {
    "Signature": "U7Tn0SRReGck4yuuVWaeZ4*****",
    "RequestId": "408fa858-cd6d-4011-b8a0-653805*****"
```

```
}  
}
```

Save the signature content `Signature` in the file `signContent.sign` :

```
echo "U7Tn0SRReGck4yuuVWaeZ4*****" | base64 -d > signContent.bin
```

## Step 6: Verifying the signature

- Call the KMS signature verification API to verify the signature (**recommended**).

Request:

```
// Verify the Base64-encoded original message  
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-  
525400***** --SignatureValue "U7Tn0SRReGck4yuuVWaeZ4*****" --Message  
"dG***Ao=" --Algorithm SM2DSA --MessageType RAW  
// Verify the message abstract (verify the signature for the message  
abstract using the content of the file `encoded.base64` mentioned in  
step 4 as the `Message` parameter of `VerifyByAsymmetricKey`).  
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-  
525400***** --SignatureValue "U7Tn0SRReGck4yuuVWaeZ4*****" --Message  
"QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ*****=" --Algorithm SM2DSA --  
MessageType DIGEST
```

### Note:

The value of the parameter `Message` and `MessageType` used in the signature API call should be the same as those of the signature verification API call.

Returned result:

```
{  
  "Response": {  
    "SignatureValid": true,  
    "RequestId": "6758cbf5-5e21-4c37-a2cf-8d47f5*****"  
  }  
}
```

- Verify the signature locally using the KMS public key and signature content.

Request:

```
gmssl sm2utl -verify -in ./test_verify.txt -sigfile ./signContent.bin  
-pubin -inkey ./public_key.pem -id 1234567812345678
```

Returned result:

```
Signature Verification Successful
```

# RSA Signature Verification

Last updated: 2024-01-11 16:31:21

This document describes how to use the RSA signature verification algorithm.

## Operation Directions

### Step 1: Creating an asymmetric signature key

#### Note:

To use the signature feature, the correct key purpose `KeyUsage= ASYMMETRIC_SIGN_VERIFY_RSA_2048` is required when calling the [CreateKey](#) API to create a CMK.

#### Request:

```
tccli kms CreateKey --Alias test_rsa --KeyUsage
ASYMMETRIC_SIGN_VERIFY_RSA_2048
```

#### Returned result:

```
{
  "Response": {
    "KeyId": "22d79428-61d9-11ea-a3c8-525400*****",
    "Alias": "test_rsa",
    "CreateTime": 1583739580,
    "Description": "",
    "KeyState": "Enabled",
    "KeyUsage": "ASYMMETRIC_SIGN_VERIFY_RSA_2048",
    "TagCode": 0,
    "TagMsg": "",
    "RequestId": "0e3c62db-a408-406a-af27-dd5ced*****"
  }
}
```

### Step 2: Downloading 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":
    "MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzsigXJahujq+
    PvM*****bBs/f3axWbvgvHx8Jmqw==",
    "PublicKeyPem": "-----BEGIN PUBLIC KEY-----
    \nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzsigXJa\nhu
    jq+PvM*****bBs/f3axWbvgvHx8Jmqw==\n-----END PUBLIC KEY-----
    \n"
  }
}
```

- Convert the public key `PublicKeyPem` into the PEM format and save it in the file `public_key.pem`.

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzsigXJa
hujq+PvM*****bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

**Note:**

You can also log in to the [KMS console](#), click **Customer Managed CMK** on the left sidebar, click a key ID/name in the key list to view the key information, and download the public asymmetric key.

### Step 3: Creating the plaintext file

Create the testing plaintext file.

```
echo "test" > test_verify.txt
```

**Note:**

If there are any invisible characters such as line breaks in the generated content, you need to truncate the file (e.g., `truncate -s -1 test_verify.txt`) to provide a correct signature.

## Step 4: Calculating the message abstract

### Note:

- If the message to be generated a signature for is not longer than 4,096 bytes, you can skip this step to [Step 5](#).
- If the message to be generated a signature for is longer than 4,096 bytes, you need to calculate a message abstract locally first.

Use OpenSSL to calculate the message abstract for `test_verify.txt`.

```
openssl dgst -sha256 -binary -out digest.bin test_verify.txt
```

## Step 5: Calling the KMS signature API to generate a signature

Call the KMS [SignByAsymmetricKey](#) API to calculate the signature.

1. Base64-encode the original message or message abstract before signature calculation.

```
// Base64-encode the message abstract
openssl enc -e -base64 -A -in digest.bin -out encoded.base64
// Base64-encode the original message
openssl enc -e -base64 -A -in test_verify.txt -out encoded.base64
```

2. Calculate the signature.

### Request:

- RSA\_PSS\_SHA\_256

```
// Generate the signature for the message abstract using the
content of the file `encoded.base64` as the `Message`
parameter of `SignByAsymmetricKey`.
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-
525400***** --Algorithm RSA_PSS_SHA_256 --Message
"qJQj83hSyOuU7Tn0SRReGck4yuuVWaeZ44BP*****==" --MessageType
DIGEST
// Generate the signature for the Base64-encoded original
message
```

```
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400***** --Algorithm RSA_PSS_SHA_256 --Message "dG***Ao=" --MessageType RAW
```

- RSA\_PKCS1\_SHA\_256

```
// Generate the signature for the message abstract using the content of the file `encoded.base64` as the `Message` parameter of `SignByAsymmetricKey`.
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400***** --Algorithm RSA_PKCS1_SHA_256 --Message "qJQj83hSyOuU7Tn0SRReGck4yuuVWaeZ44BP*****==" --MessageType DIGEST
// Generate the signature for the Base64-encoded original message
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400***** --Algorithm RSA_PKCS1_SHA_256 --Message "dG***Ao=" --MessageType RAW
```

- Returned result:

```
{
  "Response": {
    "Signature": "U7Tn0SRReGck4yuuVWaeZ4*****",
    "RequestId": "408fa858-cd6d-4011-b8a0-653805*****"
  }
}
```

- Save the signature content `Signature` in the file `signContent.sign` :

```
echo "U7Tn0SRReGck4yuuVWaeZ4*****" | base64 -d > signContent.bin
```

## Step 6: Verifying the signature

1. Call the KMS signature verification API to verify the signature (**recommended**).

- Request:

- RSA\_PSS\_SHA\_256

```
// Verify the message abstract (verify the signature for the
message abstract using the content of the file
`encoded.base64` mentioned in step 4 as the `Message`
parameter of `VerifyByAsymmetricKey`).
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-
a3c8-525400***** --SignatureValue
"U7Tn0SRReGck4yuuVWaeZ4*****" --Message
"QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ*****=" --Algorithm
RSA_PSS_SHA_256 --MessageType DIGEST
// Verify the Base64-encoded original message.
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-
a3c8-525400***** --SignatureValue
"U7Tn0SRReGck4yuuVWaeZ4*****" --Message "dG***Ao=" --
Algorithm RSA_PSS_SHA_256 --MessageType RAW
```

#### ○ RSA\_PKCS1\_SHA\_256

```
// Verify the message abstract (verify the signature for the
message abstract using the content of the file
`encoded.base64` mentioned in step 4 as the `Message`
parameter of `VerifyByAsymmetricKey`).
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-
a3c8-525400***** --SignatureValue
"U7Tn0SRReGck4yuuVWaeZ4*****" --Message
"QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ*****=" --Algorithm
RSA_PKCS1_SHA_256 --MessageType DIGEST
// Verify the Base64-encoded original message.
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-
a3c8-525400***** --SignatureValue
"U7Tn0SRReGck4yuuVWaeZ4*****" --Message "dG***Ao=" --
Algorithm RSA_PKCS1_SHA_256 --MessageType RAW
```

#### ○ Returned result:

```
{
  "Response": {
    "SignatureValid": true,
    "RequestId": "6758cbf5-5e21-4c37-a2cf-8d47f5*****"
```

```
}  
}
```

**Note:**

The value of the parameter `Message` and `MessageType` used in the signature API call should be the same as those of the signature verification API call.

2. Verify the signature locally using the KMS public key and signature content.

**Request:**

```
// Use the `RSA_PSS_SHA_256` algorithm to verify the signature.  
openssl dgst -verify public_key.pem -sha256 -sigopt  
rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -signature  
./signContent.bin ./test_verify.txt  
// Use the `RSA_PKCS1_SHA_256` algorithm to verify the signature.  
openssl dgst -verify public_key.pem -sha256 -signature  
./signContent.bin ./test_verify.txt
```

**Returned result:**

```
Verified OK
```

# ECC Signature Verification

Last updated: 2024-01-11 16:31:21

This document describes how to use the ECC signature verification algorithm.

## Operation Directions

### Step 1: Creating an asymmetric signature key

#### Note:

To use the signature feature, the correct key purpose `ASYMMETRIC_SIGN_VERIFY_ECC` is required when calling the KMS [CreateKey](#) API to create a CMK.

#### ● Request:

```
tccli kms CreateKey --Alias test_ecc --KeyUsage
ASYMMETRIC_SIGN_VERIFY_ECC
```

#### ● Returned result:

```
{
  "Response": {
    "KeyId": "22d79428-61d9-11ea-a3c8-525400*****",
    "Alias": "test_ecc",
    "CreateTime": 1583739580,
    "Description": "",
    "KeyState": "Enabled",
    "KeyUsage": "ASYMMETRIC_SIGN_VERIFY_ECC",
    "TagCode": 0,
    "TagMsg": "",
    "RequestId": "0e3c62db-a408-406a-af27-dd5ced*****"
  }
}
```

### Step 2: Downloading 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":
    "MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzISgXJahujq+
    PvM*****bBs/f3axWbvgvHx8Jmqw==",
    "PublicKeyPem": "-----BEGIN PUBLIC KEY-----
    \nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzISgXJa\nhu
    jq+PvM*****bBs/f3axWbvgvHx8Jmqw==\n-----END PUBLIC KEY-----
    \n"
  }
}
```

- Convert the public key `PublicKeyPem` into the PEM format and save it in the file `public_key.pem`:

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzISgXJa
hujq+PvM*****bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

**Note:**

You can also log in to the [KMS console](#), click **Customer Managed CMK** on the left sidebar, click a key ID/name in the key list to view the key information, and download the public asymmetric key.

### Step 3: Creating the plaintext file

Create the testing plaintext file.

```
echo "test" > test_verify.txt
```

**Note:**

If there are any invisible characters such as line breaks in the generated content, you need to truncate the file (e.g., `truncate -s -1 test_verify.txt`) to provide a correct signature.

## Step 4: Calculating the message abstract

### Note:

- If the message to be generated a signature for is not longer than 4,096 bytes, you can skip this step to [Step 5](#).
- If the message to be generated a signature for is longer than 4,096 bytes, you need to calculate a message abstract locally first.

Use OpenSSL to calculate the message abstract for `test_verify.txt`.

```
openssl dgst -sha256 -binary -out digest.bin test_verify.txt
```

## Step 5: Calling the KMS signature API to generate a signature

Call the KMS [SignByAsymmetricKey](#) API to calculate the signature.

1. Base64-encode the original message or message abstract before signature calculation.

```
// Base64-encode the message abstract.
openssl enc -e -base64 -A -in digest.bin -out encoded.base64
// Base64-encode the original message.
openssl enc -e -base64 -A -in test_verify.txt -out encoded.base64
```

2. Calculate the signature.

### Request:

```
// Generate the signature for the message abstract using the
content of the file `encoded.base64` as the `Message` parameter of
`SignByAsymmetricKey`.
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-
525400***** --Algorithm ECC_P256_R1 --Message
"qJQj83hSyOuU7Tn0SRReGck4yuuVWaeZ44BP*****==" --MessageType
DIGEST
// Generate the signature for the Base64-encoded original message.
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-
525400***** --Algorithm ECC_P256_R1 --Message "dG***Ao=" --
```

```
MessageType RAW
```

○ Returned result:

```
{
  "Response": {
    "Signature": "U7Tn0SRReGck4yuuVWaeZ4*****",
    "RequestId": "408fa858-cd6d-4011-b8a0-653805*****"
  }
}
```

○ Save the signature content `Signature` in the file `signContent.sign` :

```
echo "U7Tn0SRReGck4yuuVWaeZ4*****" | base64 -d > signContent.bin
```

## Step 6: Verifying the signature

1. Call the KMS signature verification API to verify the signature (**recommended**).

○ Request:

```
// Verify the message abstract (verify the signature for the
message abstract using the content of the file `encoded.base64`
mentioned in step 4 as the `Message` parameter of
`VerifyByAsymmetricKey`).
tcli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-
525400***** --SignatureValue "U7Tn0SRReGck4yuuVWaeZ4*****" --
Message "QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ*****=" --Algorithm
ECC_P256_R1 --MessageType DIGEST
// Verify the Base64-encoded original message.
tcli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-
525400***** --SignatureValue "U7Tn0SRReGck4yuuVWaeZ4*****" --
Message "dG***Ao=" --Algorithm ECC_P256_R1 --MessageType RAW
```

○ Returned result:

```
{
  "Response": {
    "SignatureValid": true,
  }
}
```

```
"RequestId": "6758cbf5-5e21-4c37-a2cf-8d47f5*****"  
}  
}
```

**Note:**

The value of the parameter `Message` and `MessageType` used in the signature API call should be the same as those of the signature verification API call.

2. Verify the signature locally using the KMS public key and signature content.

**Request:**

```
openssl dgst -verify public_key.pem -sha256 -signature  
./signContent.bin ./test_verify.txt
```

**Returned result:**

```
Verified OK
```

# Post-Quantum Cryptography Practice In KMS

Last updated: 2025-03-03 15:41:23

## Overview

With the rapid development of quantum computers, traditional cryptography faces severe challenges: public key cryptosystems based on prime factorization (RSA), discrete logarithm (DH), and elliptic curve cryptography (ECC) can all be cracked by quantum computers using Shor's algorithm. In the face of quantum threats, post-quantum cryptography (PQC) has been designed to resist the cracking by quantum computers, and Key Management Service (KMS) supports the following two PQC cryptographic algorithms:

- Kyber-based PQC encryption and decryption algorithm to protect data confidentiality.
- Dilithium-based PQC signature verification algorithm to ensure data integrity.

## Data Encryption Algorithm

The Kyber algorithm is based on the Module Learning-With-Error (MLWE) challenge and provides a basic IND-CPA secure public key encryption scheme (PKE). An IND-CCA2 secure key encapsulation mechanism (KEM) can be obtained through the Fujisaki-Okamoto (FO) transform. KMS integrates Kyber-KEM with AES-256 to implement a data encapsulation scheme (KEM-DEM), providing users with an IND-CCA2 secure and efficient encryption solution.

## Operation Steps

1. Log in to the [KMS \(Compliance Edition\)](#) console.
2. Refer to the document [Creating a Key](#), select asymmetric encryption/decryption for the key purpose, and choose Kyber\_AES for the encryption algorithm.
3. Refer to the document [Post-Quantum Cryptography Encryption](#) and [Post-Quantum Cryptography Decryption](#), use Tencent Cloud SDK to call relevant APIs to perform encryption and decryption operations.

## Data Signature Algorithm

The security of the Dilithium algorithm is based on the NP problem of finding the shortest vector in a lattice. The algorithm design takes into account the size of the public key and signature. NIST Level 3 can ensure high security strength. Dilithium supports DET and Random signatures, and its usage scenarios are flexible. It can be called through the SDK of KMS to use related signature verification algorithms.

## Operation Steps

1. Log in to the [KMS \(Compliance Edition\)](#) console.

2. Refer to the document [Creating a Key](#), select asymmetric signature verification for the key purpose, and choose Dilithium for the encryption algorithm.
3. Refer to the document [Post-Quantum Cryptography Signature](#) and [Post-Quantum Cryptography Signature Verification](#), use Tencent Cloud SDK to call relevant APIs to perform signature verification operations.

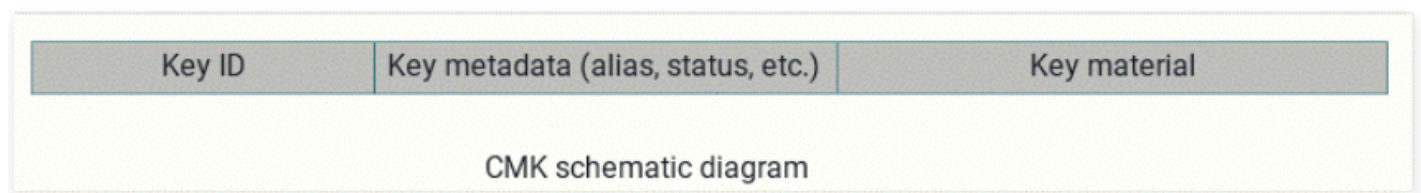
# Importing External Key

## Overview

Last updated: 2024-01-11 16:31:22

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.



## 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: 2026-03-25 16:19:59

## 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 Base64-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.

#### Note:

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 -keyform 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 -keyform 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 -keyform 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 <keyid>
```

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.

#### Note:

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
```

 **Note:**

- 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.

# Implementing Exponential Backoff to Deal with Service Frequency

Last updated: 2024-01-11 16:31:22

## Suggestions for Dealing with Exceptions

If exceptional errors occur when you call KMS APIs to send requests from your application to the remote KMS server, you can deal with the errors as suggested below:

- **Cancel the call:** if an error shows that the failure is not temporary and persists after several re-executions, you need to terminate or cancel the application call and report the error.
- **Try again immediately:** if an uncommon error is returned, for example, network packets are damaged during transfer but still sent, in this case, you can try again immediately.
- **Increase delays between re-executions:** if an error is generally caused by connections, it indicates that the server is busy and needs to clear the loads first. You can try again in a while in such cases.

The following paragraphs introduce how to increase delays between re-executions. The delay can be gradually increased or scheduled (by implementing exponential backoff). As the frequency of KMS API calls is limited, you can increase delays between re-executions to avoid errors caused by high frequency.

## Exponential Backoff

### Pseudocode

```
// Gradually increase re-execution delays
InitDelayValue = 100
For (Retries = 0; Retries < MAX_RETRIES; Retries = Retries+1)
    wait for (2^Retries * InitDelayValue) milliseconds
    Status = KmsApiRequest()
    IF Status == SUCCESS
        BREAK // Succeeded, stop calling the API again.
    ELSE IF Status = THROTTLED || Status == SERVER_NOT_READY
        CONTINUE // Failed due to throttling or server busy, try
again.
    ELSE
        BREAK // another error occurs, stop calling the API again.
END IF
```

### Method

Python: implement exponential backoff for frequency errors in KMS API calls to `Encrypt`

```
# -*- coding: utf-8 -*-
import base64
import math
import time
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 BackoffFunction(RetryCount):
    InitDelayValue = 100
    DelayTime = math.pow(2, RetryCount) * InitDelayValue
    return DelayTime

if __name__ == '__main__':
    # User-defined parameters
    secretId = "replace-with-real-secretId"
    secretKey = "replace-with-real-secretKey"
    region = "ap-guangzhou"
    keyId = "replace-with-realkeyid"
    plaintext = "abcdefg123456789abcdefg123456789abcdefg"
    Retries = 0
    MaxRetries = 10
    client = KmsInit(region, secretId, secretKey)
    req = models.EncryptRequest()
```

```
req.KeyId = keyId
req.Plaintext = base64.b64encode(plaintext)
while Retries < MaxRetries:
    try:
        Retries += 1
        rsp = client.Encrypt(req) # Call the API `Encrypt`
        print 'plaintext: ',plaintext,'CiphertextBlob: ',rsp.CiphertextBlob
        break
    except TencentCloudSDKException as err:
        if err.code == 'InternalError' or err.code == 'RequestLimitExceeded':
            if Retries == MaxRetries:
                break
            time.sleep(BackoffFunction(Retries + 1))
            continue
        else:
            print(err)
            break
    except Exception as err:
        print(err)
        break
```

**Note:**

- To deal with other specific errors, you can directly modify the content of the statement `except`.
- You can customize the schedule policy based on your code logic and business policy to set the optimal initial delay value (InitDelayValue) and the number of retries (Retries), preventing your business from being affected by a too-low or too-high threshold.

# Cloud Product Integration with KMS for Transparent Encryption

Last updated: 2025-08-27 15:35:40

## Overview

Tencent Cloud Key Management Service (KMS) is a secure, reliable, and simple and easy-to-use managed service that helps you easily create and manage keys to protect the security of the key. Tencent Cloud KMS seamlessly integrates with most Cloud services on Tencent Cloud. For Cloud products integrated with KMS, you only need to select a key managed by KMS to encrypt/decrypt data within the Cloud Product.

Cloud Product Integration with KMS encryption brings the following benefits:

- Cloud services encrypt user data through integration with KMS, with encryption keys controlled by users. KMS uses FIPS-140-2 certified Hardware Security Modules (HSM) to generate and protect keys.
- Provide users with a transparent encryption solution. Users only need to enable the encryption service for Cloud products integrated with KMS, with no need to worry about encryption details, to achieve transparent data encryption/decryption in the cloud.
- No need to manually build and maintain a Key Management Infrastructure, reducing development costs while ensuring secure and convenient user use.

### Note:

Since other cloud products are not key managers, before using Cloud products integrated with KMS to encrypt data, you need to pass through Tencent Cloud CAM to complete the role-based authorization operation of KMS for Cloud services.

## Supported Key Types

Root Key (Customer Master Key, CMK) is a key created by users or Cloud services through the Key Management System, primarily used to encrypt and protect data keys. A root key can encrypt multiple data keys (DEKs).

KMS provides the following two root keys:

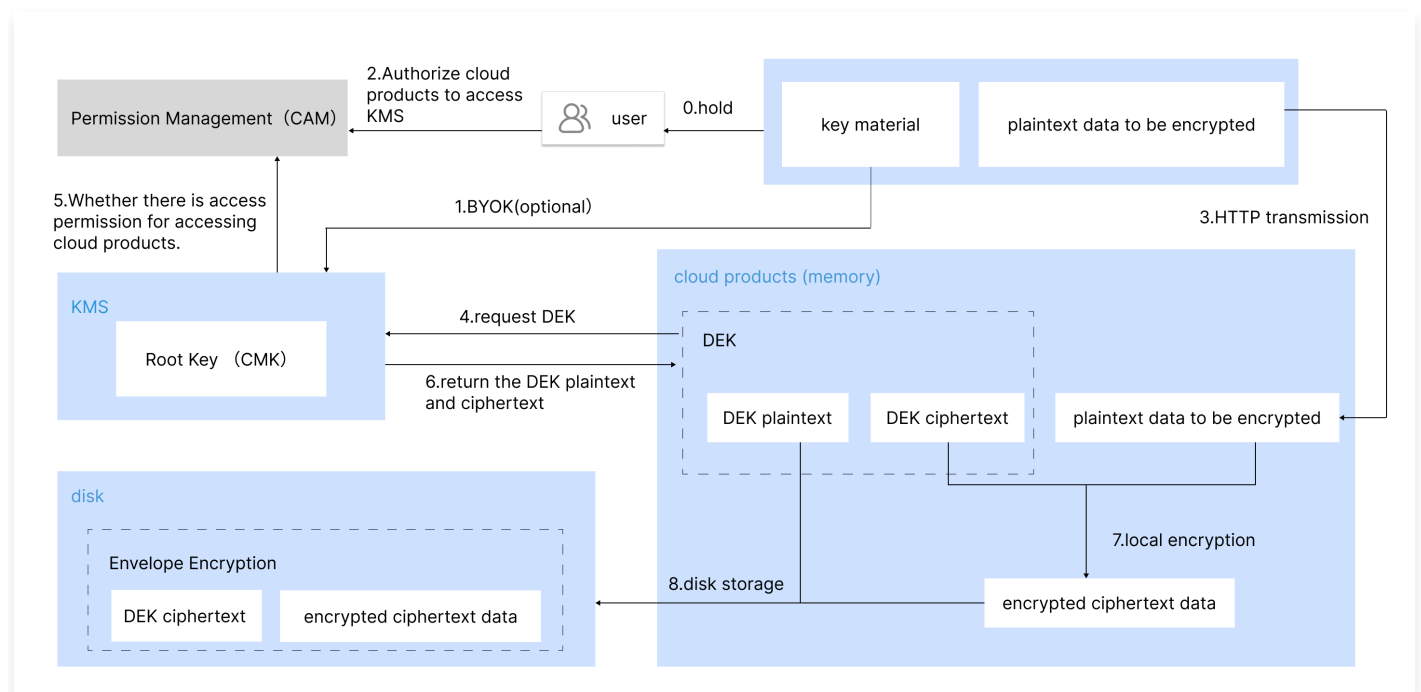
- Custom key
- Users can autonomously create keys through the Key Management System. There are two key source methods: **KMS creation** or **manual import** (BYOK). For more information, see [key creation](#) and [external key import](#) in the key management system.
- Cloud service key by default
- When a user first uses KMS encryption through corresponding cloud services, the cloud platform automatically creates a key for the user via the Key Management System. Cloud service keys can be

queried in the KMS console but do not support disable or scheduled deletion operations.

## Encryption Principles

The business forms and customer requirements of different cloud products vary slightly in their specific encryption designs. Typically, cloud products use the [Envelope encryption](#) way to achieve data encryption and decryption by calling the KMS API.

The encryption principle for cloud products using KMS is illustrated in the figure below:



The encryption process is as follows:

1. Enable KMS service and complete role authorization for cloud products.
2. Create a root CMK in KMS. Users can use the default cloud service key or customize one.
3. Generate a data key DEK ciphertext and DEK plaintext by calling the GenerateDataKey API with CMK. The DEK is encrypted for protection by CMK.
4. The DEK plaintext is cached in the product backend memory. Encrypt user data locally to get ciphertext data.
5. Cloud services store the DEK ciphertext and encrypted ciphertext data on disk.

## Currently Supported Cloud Products

- [TDSQL-C for MySQL](#)
- [TencentDB for MySQL](#)
- [TencentDB for MariaDB](#)
- [TencentDB for MongoDB](#)
- [TencentDB for PostgreSQL](#)

- [Cloud Object Storage \(COS\)](#)
- [Cloud File Storage \(CFS\)](#) (allowlist for usage)
- [Cloud Block Storage \(CBS\)](#)
- [Secrets Manager \(SSM\)](#)