Key Management Service
Best Practice

Tencent Cloud

Copyright Notice

Trademark Notice

Tencent Cloud

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

Best Practice
Symmetric encryption and decryption
Encrypting/Decrypting Sensitive Data
Overview

Last updated：2023-08-24 17:07:52

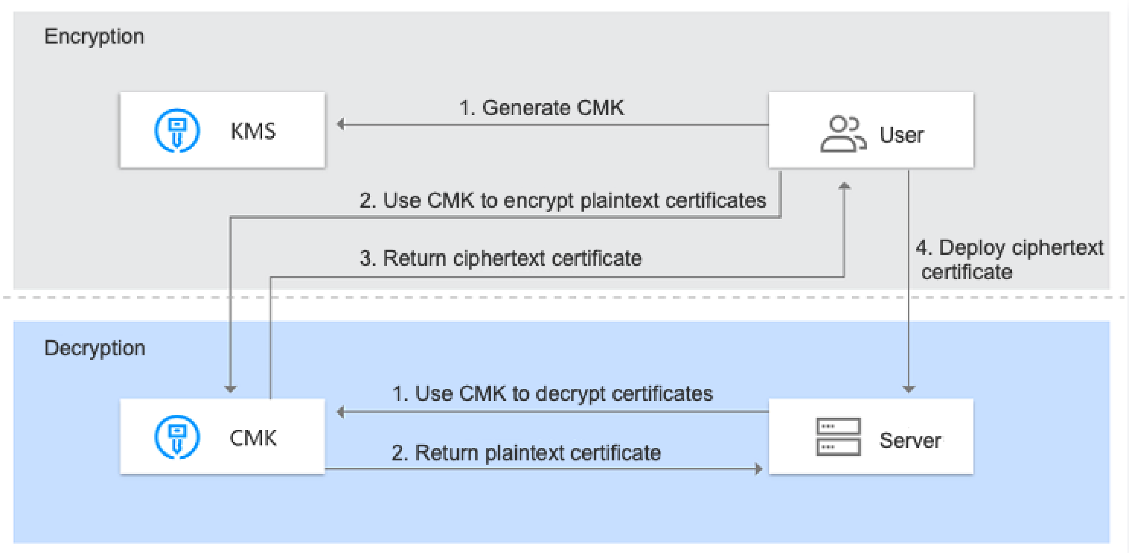
Sensitive information encryption is a core capability of Key Management System (KMS), suitable for protecting small-sized sensitive data (less than 4 KB), such as keys, certificates, configuration files, etc. Encrypt sensitive data using Customer Master Keys (CMK) instead of storing plaintext directly. When in use, decrypt the ciphertext to memory, ensuring that plaintext is not written to disk. The entire interaction and transmission process uses HTTPS requests to guarantee the security of sensitive data.
If you require high-performance encryption and decryption services for massive data using KMS, please refer to the Envelope Encryption scenario.

Example of Sensitive Information

-	Key/Certificate	Backend Configuration File
Use	Encrypted Business Data, Communication Channel, Digital Signature	Storing system architecture and other business information, such as database IP and password.
Risk of data loss	Confidential information theft, encrypted channel eavesdropping, signature forgery	Business data is breached and used as a springboard to attack other systems.

Schematic Diagram

In this scenario, sensitive data is protected through encryption and decryption using the Customer Master Key (CMK), which is safeguarded by a third-party certified Hardware Security Module (HSM). The CMK performs encryption and decryption within the HSM, ensuring that the plaintext CMK remains inaccessible to any unauthorized personnel, including Tencent Cloud staff.


Encryption
KMS
1. Generate CMK
User
2. Use CMK to encrypt plaintext certificates
3. Return ciphertext certificate
4. Deploy ciphertext certificate
Decryption
CMK
1. Use CMK to decrypt certificates
2. Return plaintext certificate
Server

Features

- Permission Control: Integrated with Tencent Cloud Access Management (CAM), account access to CMK is controlled through identity and policy management.
- Built-in Auditing: Integrated with Tencent CloudAudit, it can record all API requests, including key management operations and key usage. This ensures traceability and auditability of data operations.
- Centralized Key Management: Achieve centralized management of keys for various applications through Tencent Cloud KMS service.
- Security Compliance: The underlying Key Management System uses Hardware Security Modules (HSM) certified by the National Cryptography Administration or FIPS-140-2 to protect the security of keys, ensuring their confidentiality, integrity, and availability.
- Sensitive data encryption: Supports encryption and decryption of sensitive data (less than 4 KB), such as keys, certificates, configuration files, etc.

Supports and Limits

- Pay attention to the secure storage of SecretId and SecretKey:
  - Tencent Cloud API authentication primarily relies on SecretID and SecretKey, which are the user's unique authentication credentials. Business systems require these credentials to call Tencent Cloud APIs.
- Pay attention to the access control of SecretID and SecretKey:
  - It is recommended to use sub-accounts and manage risks by granting interface permissions based on business requirements.
- Caution regarding plaintext data storage:

- Data has already been encrypted through sensitive data encryption. To ensure data security, please make sure that the original plaintext data is deleted.

Operation Guide

Last updated：2023-08-24 11:44:44

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

Preliminary Preparations

- Dependent environment of the sample code: Python 2.7.
- KMS service activation: Activate the KMS product from the Tencent Cloud Console.
- Activate TencentCloud API key service: Obtain the SecretID, SecretKey, and endpoint. The general format of the endpoint is `*.tencentcloudapi.com`, such as the KMS endpoint `kms.tencentcloudapi.com`. For more information, please refer to the documentation of each 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
```

Flowchart

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

1. Create a customer master key (CMK) through the Console or the CreateKey API, which means creating a CMK.
2. Encrypt sensitive user data by calling the KMS encryption API (Encrypt) and obtain the ciphertext.
3. Store the encrypted data according to your business requirements.
4. When reading data, decrypt it into plaintext by calling the KMS decryption API (Decrypt) through the API.

Instructions

Step 1: Create a Customer Master Key (CMK)

For information on how to create a customer master key, please refer to the Create Key documentation.

Step 2: Encrypt sensitive information

Prerequisite: Ensure that the customer master key created in Step 1 is in the enabled state.

Using the Console

The online tool is suitable for single or non-batch encryption and decryption operations, such as the initial generation of encrypted keys. Developers do not need to develop additional tools for non-batch encryption and decryption operations, allowing them to focus on implementing core business capabilities. For more information, please refer to the Encryption and Decryption documentation.

Python SDK Method

Use the Encrypt function to encrypt user data, with a maximum data size of 4KB for any data. This can be used to encrypt database passwords, RSA keys, or other small sensitive information. This example uses the Tencent Cloud Python SDK, but you can also use other supported programming languages.
The KeyId and Plaintext are required parameters for this API operation. For more information on other parameters, please refer to the Encrypt API documentation.

Encryption Python SDK Sample:

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

Python Sample Code:

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

from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloud
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) # Invoke the encryption interface
        return rsp
    except TencentCloudSDKException as err:
        print(err)
        return None

if __name__ == '__main__':
    # User-defined parameters
    secretId = os.getenv('SECRET_ID')        # read from environment variable or use white
    secretKey = os.getenv('SECRET_KEY')   # read from environment variable or use white
    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 encrypted data according to the application scenarios of your business.

Step 4: Decrypt Sensitive Data

Using the Console

For more details, please refer to the Encryption and Decryption documentation.

Python SDK Method

The Decrypt API is used to decrypt data.
The CiphertextBlob is a mandatory parameter for this API operation. For more details, please refer to the Decrypt interface documentation for other parameter descriptions.

Python Sample Code

```
# -- coding: utf-8 --
import base64
import os
from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloud
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) # Invoke the decryption interface
        return rsp
    except TencentCloudSDKException as err:
        print(err)
        return None

if __name__ == '__main__':
    # User-defined parameters
    secretId = os.getenv('SECRET_ID')       # read from environment variable or use white
    secretKey = os.getenv('SECRET_KEY')   # read from environment variable or use white
    region = "ap-guangzhou"
    keyId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    ciphertextBlob = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

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

Envelope Encryption/Decryption Overview

Last updated：2023-08-24 17:08:22

Envelope Encryption is a high-performance encryption and decryption solution for handling massive data. For larger files or performance-sensitive data encryption, use the GenerateDataKey API to generate a Data Encryption Key (DEK). By only transmitting the DEK to the KMS server (encrypted and decrypted via CMK), all business data is processed using efficient local symmetric encryption, minimizing the impact on the user experience.

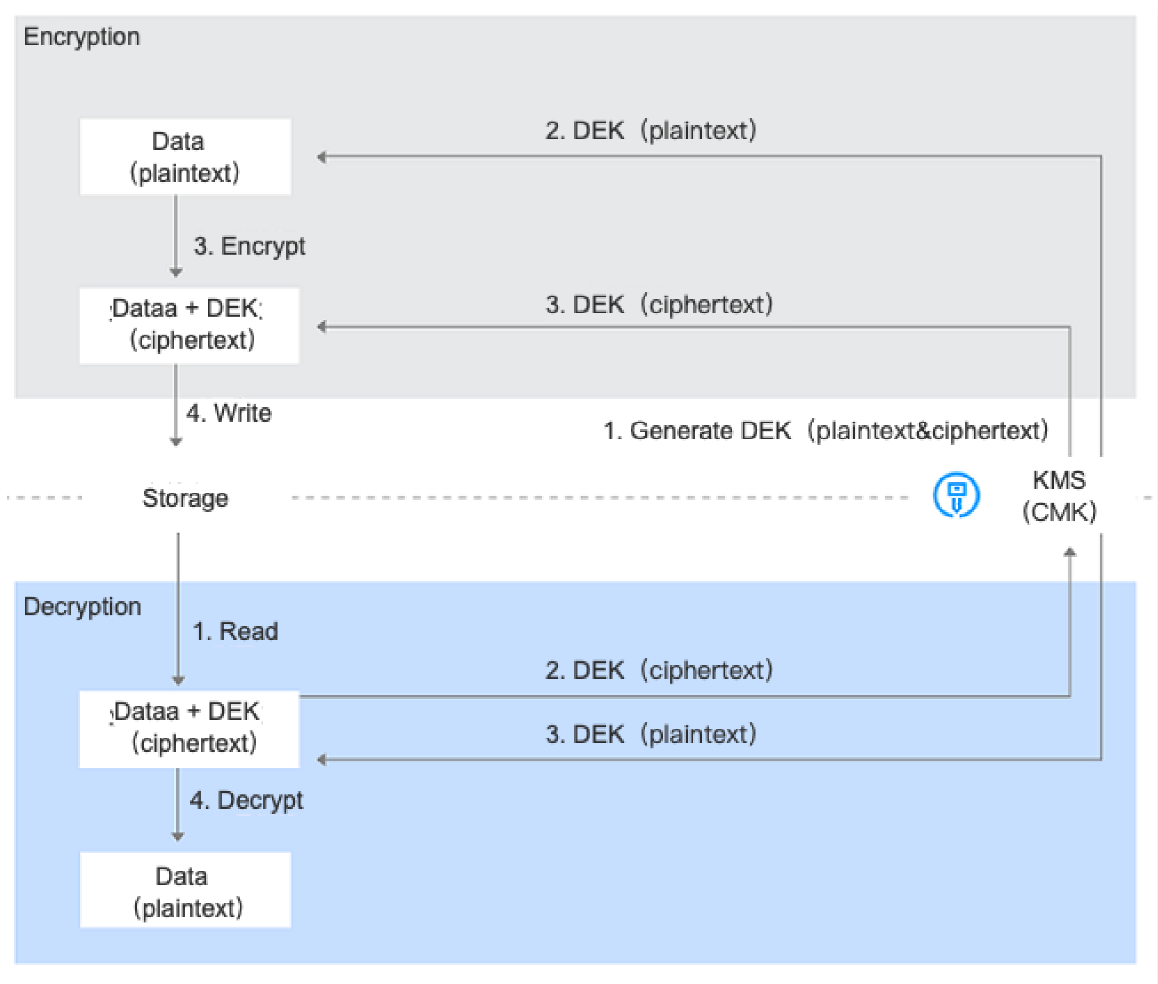
In practical business scenarios with high data encryption performance requirements and large volumes of encrypted data, generating DEKs for local data encryption and decryption ensures the necessary encryption performance while also relying on KMS to guarantee the randomness and security of the data keys.

KMS Encryption Scheme Comparison

Comparison Item	Encrypting Sensitive Information	Envelope Encryption
Associated Keys	CMK	CMK、DEK
Performance	Symmetric Encryption, Remote Invocation	Minimal Remote Symmetric Encryption, Massive Local Symmetric Encryption
Primary Scenarios	Keys, certificates, and small data, suitable for scenarios with lower invocation frequency.	Massive large-scale data, suitable for scenarios with high performance requirements.

Schematic Diagram

In this scenario, the CMK generated by KMS serves as a crucial resource for generating and obtaining the plaintext and ciphertext of the DEK. Based on the actual business scenario, users first encrypt local data in memory using the DEK plaintext, then store the DEK ciphertext and encrypted data on disk. Subsequently, in the business decryption scenario, KMS is used to decrypt the DEK ciphertext, and finally, the decrypted DEK plaintext is used to decrypt the data in memory.



Features

- Efficient: All business data is processed using efficient local symmetric encryption, minimizing the impact on the user experience. As for the creation and encryption/decryption overhead of DEKs, except in extreme cases where a "one-time pad" scheme is required, most scenarios allow for the reuse of a DEK's plaintext and ciphertext over a period of time. Therefore, in most cases, this overhead is minimal.
- Secure and user-friendly: The security of Envelope Encryption is ensured by the protection provided by KMS keys. Business data is safeguarded by DEK, while Tencent Cloud KMS protects the DEK and offers enhanced availability. Your primary key is mainly used to generate DEKs, and only entities with key access permissions can perform operations.

Supports and Limits

- Pay attention to the secure storage of SecretId and SecretKey:
  - Tencent Cloud API authentication primarily relies on SecretID and SecretKey, which

are the user's unique authentication credentials. Business systems require these credentials to call Tencent Cloud APIs.

- Pay attention to the access control of SecretID and SecretKey:
    - It is recommended to use sub-accounts and manage risks by granting interface permissions based on business requirements.
- Caution should be exercised when handling plaintext keys in business systems:
    - In Envelope Encryption scenarios, symmetric encryption is used. Therefore, the plaintext key must not be stored on disk and should be used within the memory of the business process.
- DEK processing by the backend system:
    - In envelope encryption scenarios, symmetric encryption is used. Depending on business requirements, the same data key can be reused, or different data keys can be used for different users and times to avoid DEK duplication.

Operation Guide

Last updated：2023-08-24 11:45:06

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

Preliminary Preparations

- Dependent environment of the sample code: Python 2.7.
- KMS service activation: Activate the KMS product from the Tencent Cloud Console.
- Cloud API Key Service Activation: Obtain the SecretID, SecretKey, and the endpoint. The endpoint for KMS is `kms.tencentcloudapi.com`. For more details, refer to the individual product descriptions.
- 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
```

Flowchart

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

1. Create a CMK.
2. For envelope encryption, the business application calls the KMS GenerateDataKey API to generate a data key. The system encrypts the data using the plaintext key and stores both the ciphertext key and the ciphertext on disk.
3. Data decryption: The system reads the ciphertext key and ciphertext, decrypts the ciphertext key through the KMS decryption interface, returns the plaintext key, and finally decrypts the ciphertext data using the plaintext key.

Practical Steps

Step 1: Create a Customer Master Key (CMK)

For a guide on creating a customer master key (CMK), please refer to the Create Key quick start documentation.

Step 2: Data Envelope Encryption

Based on business requirements, when a new DEK is needed (e.g., for encrypting new user data or when the DEK has been reused for a certain period and a new DEK is required), a new data key can be created through the KMS interface. After generating the data key, use the plaintext key to encrypt in memory, and finally store the ciphertext and ciphertext key on disk.

Generating a DEK and encrypting your data

Obtain the Data Encryption Key (DEK) using GenerateDataKey. DEK is a secondary key generated based on the CMK and can be used for local data encryption and decryption. KMS does not store or manage DEKs, so the caller is responsible for storage.
The examples below are implemented in the Tencent Cloud SDK for Python, which can also be implemented in other supported programming languages.
The KeyId is a mandatory parameter for this API operation. You can refer to the GenerateDataKey interface documentation for explanations of other parameters.

Python SDK Sample

```
# -- coding: utf-8 --
import base64
import os
from Crypto.Cipher import AES
from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloud
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
        Invoke the Generate Data Key interface
        generatedatakeyResp = client.GenerateDataKey(req)
        # The plaintext key is used in memory, while the ciphertext key is used for persiste
        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 = os.getenv('SECRET_ID')        # read from environment variable or use white
    secretKey = os.getenv('SECRET_KEY')  # read from environment variable or use white
    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 Retrieval

First, read the encrypted key stored on disk and decrypt it by calling the decryption interface. Finally, decrypt the data using the decrypted plaintext key.

Decryption (KMS Python SDK)

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 is a mandatory parameter for this API operation. You can refer to the Decrypt interface documentation for explanations of other parameters.

Python SDK Sample

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
import os
from Crypto.Cipher import AES
from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloud
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) # Invoke the decryption interface 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 = os.getenv('SECRET_ID')        # read from environment variable or use white
    secretKey = os.getenv('SECRET_KEY')   # read from environment variable or use white
    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：2023-08-24 17:08:57

Asymmetric encryption and decryption require two keys: a **public key** and a **private key.** These keys form a complementary pair in cryptography and possess bidirectional properties, meaning that either key can be used for encryption, but only the other key can be used for decryption. The public key can be shared with anyone, even untrusted parties, while the private key must be securely guarded by the owner.

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 length of 2048 bits, with KeyUsage set to ASYMMETRIC_DECRYPT_RSA_2048.

SM2

SM2 is a Chinese national standard public key algorithm, used to replace the RSA algorithm in China's commercial cryptographic system. For applications with corresponding Chinese cryptographic requirements, consider using this key type with KeyUsage set to 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

In practical applications, asymmetric encryption and decryption encompass two typical scenarios: **encrypted communication** and **digital signatures**.

Encrypted Communication

Encrypted communication is a typical application of asymmetric encryption algorithms. The process of encrypted communication is similar to symmetric encryption, with the distinction that a public key is used for encryption and a 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 technology is another typical application of asymmetric encryption algorithms. Digital signatures consist of two processes: signing and verification. The private key is used for signing, while the public key is used for verification, which is the opposite of the encryption communication process.
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.

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 digital signature solutions. For more information, please see Asymmetric Signature Verification.

> ⚠ **Note**
> Due to the unique usage scenarios of public and private keys, KMS does not support automatic rotation for asymmetric CMKs. If you need to update your keys periodically or irregularly, you can create new asymmetric keys manually.

Asymmetric Data Encryption and Decryption

Last updated：2023-08-24 11:45:31

Flowchart

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 in the Key Management System (KMS). For more information, please see Create a Master Key.
2. Obtain the public key from the Key Management System (KMS). For more details, please refer to Get the Public Key of an Asymmetric Key.
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. After receiving the ciphertext, the information recipient uses KMS's decryption feature to decrypt it. For API details, please refer to Asymmetric Key Sm2 Decryption and Asymmetric Key RSA Decryption. For 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.

Instructions

RSA sample

1. Create an Asymmetric Encryption Key

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

Response:

```
{
   "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 Public Key

Request:

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

Response:

```
{
  "Response": {
    "RequestId": "408fa858-cd6d-4011-b8a0-653805**",
    "KeyId": "22d79428-61d9-11ea-a3c8-525400**",
    "PublicKey": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgV
    "PublicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCA
  }
}
```

3. Encrypt using the public key

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

Store it in the file:

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 D
```

3.4 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** --Algor
```

   Response:

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

Note:
The process for using SM2 asymmetric key encryption and decryption is similar. For details on the private key decryption API, please refer to Using SM2 to Decrypt Data with Asymmetric Keys.

Asymmetric Signature Verification Overview

Last updated：2023-08-24 11:45:41

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

1. Create an asymmetric key pair in the Key Management System (KMS). For more information, please see Create a Master Key.
2. The information sender uses the created user private key to generate a signature for the data to be transmitted. For more information, please see Signing.
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:

   4.1 Call the KMS signature verification API for validation. For more information, please see Verify Signature.

   4.2 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 signature verification algorithms.

SM2 Signature Verification

Last updated：2023-08-24 11:45:49

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

Instructions

Step 1: Creating an asymmetric signature key

> **Note**
> When creating a customer master key (CMK) in Key Management Service (KMS) by calling the Create CMK API, you must provide the correct key usage parameter, KeyUsage=ASYMMETRIC_SIGN_VERIFY_SM2, to enable the signature functionality.

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":  "MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLIge0vtct949CwtadHOD
    "PublicKeyPem": "-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0D
  }
}
```

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

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLIge0vtct949CwtadHODzisgXJa
hujq+PvM***bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

> **Note**
> Additionally, you can log in to the KMS Console, click **User Key** > **Key ID/Key Name** to access the key information page, and directly download the asymmetric public key.

Step 3: Creating the plaintext information file

Create a testing plaintext file:

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

> **Note**
> When the generated file content contains invisible characters (such as newline characters), you need to perform a truncate operation on the file (**truncate -s -1 test_verify.txt**) to ensure signature accuracy.

Step 4: Calculating the message abstract

- If the length of the message to be signed does not exceed 4,096 bytes, you can skip this step and proceed directly 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 12345678123
```

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 encod
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400** --Algorit

// Generate the signature for the Base64-encoded original message
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400* --Algorit
```

Returned result:

```
{
    "Response": {
      "Signature": "U7Tn0SRReGCk4yuuVWaeZ4**",
      "RequestId": "408fa858-cd6d-4011-b8a0-653805**"
    }
}
```

Save the signature content (Signature) into a file named 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** --Signa
// Verify the message abstract (verify the signature for the message abstract using the
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400** --Signa
```

> **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
```

Returned result:

```
Signature Verification Successful
```

RSA Signature Verification

Last updated：2023-08-24 11:45:59

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

Instructions

Step 1: Creating an asymmetric signature key

> **Note**
> When calling the Create Primary Key interface in KMS, you must pass the correct key usage parameter, KeyUsage=ASYMMETRIC_SIGN_VERIFY_RSA_2048, in order to use the signature feature.

- **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":  "MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLIge0vtct949CwtadHODzis
"PublicKeyPem": "-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQ
}
}
```

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

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLIge0vtct949CwtadHODzisgXJa
hujq+PvM***bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

Note
You can log in to the KMS Console, click **User Key** > **Key ID/Key Name** to access the key information page, and directly download the asymmetric public key.

Step 3: Creating the plaintext information 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 signed is no longer than 4,096 bytes, you can skip this step and proceed directly 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_verity.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 f
       tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400*
       // Generate the signature for the Base64-encoded original message
       tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400*
       ```

     - **RSA_PKCS1_SHA_256**

       ```
       // Generate the signature for the message abstract using the content of the f
       tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400*
       // Generate the signature for the Base64-encoded original message
       tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400*
       ```

   - **Returned result:**

     ```
     {
     "Response": {
       "Signature": "U7Tn0SRReGCk4yuuVWaeZ4**",
       "RequestId": "408fa858-cd6d-4011-b8a0-653805**"
     }
     }
     ```

- **Save the signature content Signature into 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 method).
   - **Request:**
     - **RSA_PSS_SHA_256**

```
// Verify the message abstract (verify the signature for the message abstract
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400
// Verify the Base64-encoded original message.
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400
```

     - **RSA_PKCS1_SHA_256**

```
// Verify the message abstract (verify the signature for the message abstract
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400
// Verify the Base64-encoded original message.
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400
```

   - **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
// Use the RSA_PKCS1_SHA_256 algorithm to verify the signature.
openssl dgst -verify public_key.pem -sha256 -signature ./signContent.bin ./test_ver
```

- **Returned result:**

```
Verified OK
```

ECC Signature Verification

Last updated：2023-08-24 17:16:16

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

Instructions

Step 1: Creating an asymmetric signature key

> **Note**
> When creating a customer master key (CMK) in Key Management System (KMS) using the CreateKey API, you must provide the correct key usage ASYMMETRIC_SIGN_VERIFY_ECC to enable the signature functionality.

- **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":  "MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLIge0vtct949CwtadHODzis
"PublicKeyPem": "-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQ
}
}
```

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

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLIge0vtct949CwtadHODzisgXJa
hujq+PvM***bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

> **Note**
> You can also log in to the KMS Console, click **User Key** > **Key ID/Key Name** to access the key information page, and directly download the asymmetric public key.

Step 3: Creating the plaintext information 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 signed is no longer than 4,096 bytes, you can skip this step and proceed directly 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_verity.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 er
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400** --Al
// Generate the signature for the Base64-encoded original message.
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400* --Alg
```

   - **Returned result:**

```
{
"Response": {
  "Signature": "U7Tn0SRReGCk4yuuVWaeZ4**",
  "RequestId": "408fa858-cd6d-4011-b8a0-653805**"
}
}
```

   - **Save the signature content Signature into a file named 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 method).

- **Request:**

```
// Verify the message abstract (verify the signature for the message abstract using
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400** --S
// Verify the Base64-encoded original message.
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400** --S
```

- **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_ver
```

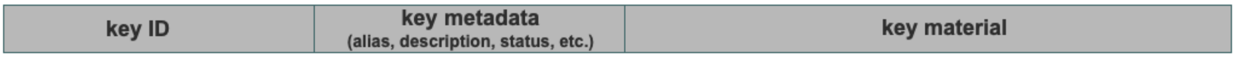
- **Returned result:**

```
Verified OK
```

Importing External Keys
Overview

Last updated：2023-08-24 17:09:24

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, when creating a CMK through the KMS service, secure key material is generated by the underlying encryption engine. If you prefer to use your own key material, implementing a Bring Your Own Key (BYOK) solution, you can generate a CMK with empty key material using the KMS service. Then, import your key material into the customer master key, forming an external CMK (EXTERNAL CMK). The KMS service will then handle the distribution and management of this external key.



key ID	key metadata (alias, description, status, etc.)	key material

CMK Structure 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.

Supports and Limits

- It is essential to ensure the security of the imported key material:
  - When using the key import feature, ensure the security and reliability of the random source used to generate your key material. Currently, KMS only allows importing 128-bit symmetric keys for the Chinese cryptographic version and 256-bit symmetric keys for the FIPS version.
- Ensure the availability of the imported key material:
  - While KMS provides high availability and backup recovery capabilities for its own

service, the availability of imported key material must be managed by the user. It is strongly recommended that you securely store the original backup of your key material to ensure timely re-importation into KMS in case of accidental deletion or expiration of the key material.

- It is essential to ensure the correctness of key importing operations:
  - When you import key material into a CMK, it becomes permanently associated with that CMK, meaning no other key material can be imported into the external CMK. When encrypting data with the external CMK, the encrypted data must be decrypted using the same CMK (i.e., the CMK's metadata and key material must match the imported key) or decryption will fail. Please handle key material and CMK deletion operations with caution.
- 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：2023-08-24 11:46:33

Flowchart

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

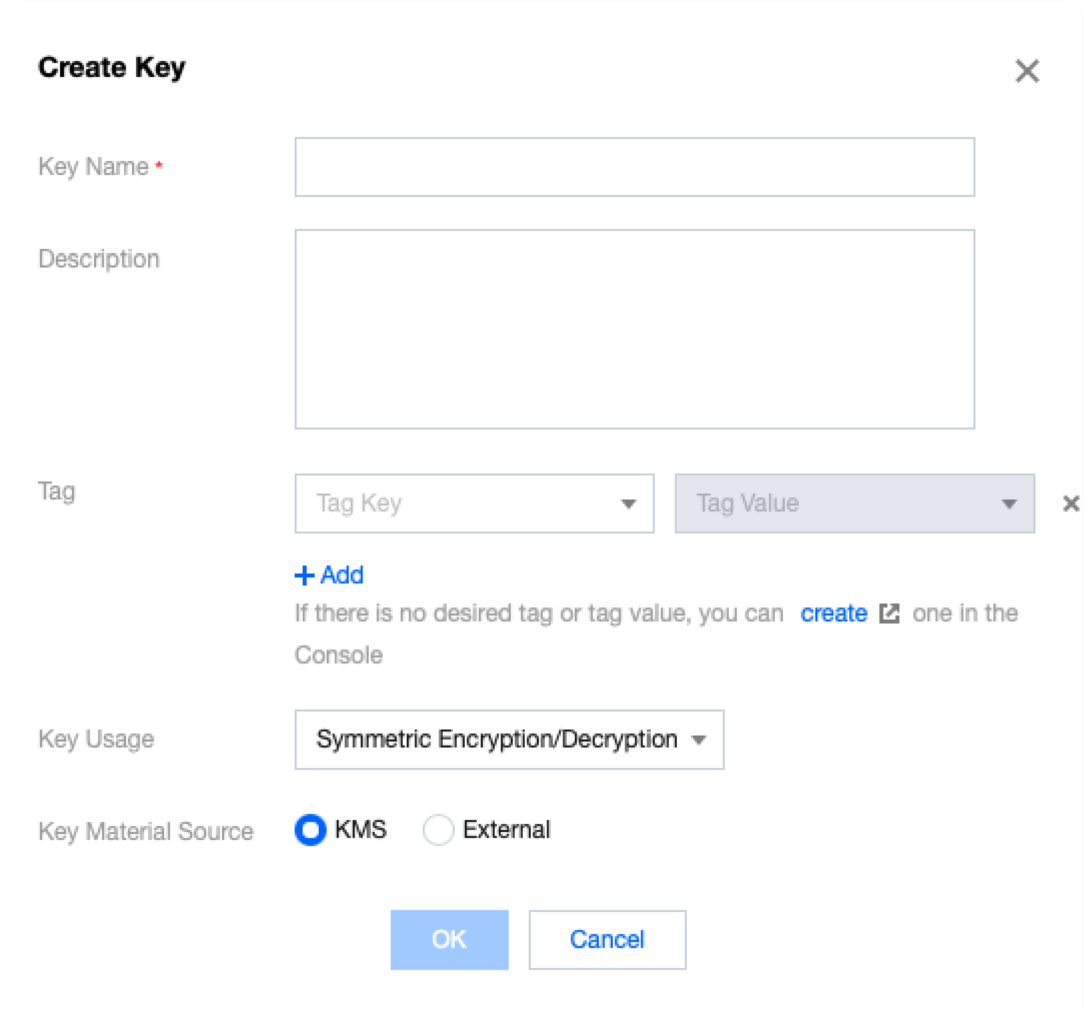
1. Create a customer master key (CMK) with an "external" key source, either through the console or API, to create an external CMK.
2. Obtain the parameters for importing key material through API operations, including a public key for encrypting the key material and an import token.
3. Using the encryption public key obtained in step 2, encrypt your key material locally with a hardware security module (HSM) or other secure encryption measures.
4. Through API operations, import the encrypted key material and the import token obtained in step 2 into the created external CMK, completing the external key import process.

Instructions

Step 1: Create an external CMK

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

- **Console Method**
  (1) Log in to the Key Management System (Compliance) console.
  (2) Select the region where you want to create the key and click **Create** to start the process.
  (3) In the Create Key window, enter the key name, select the key material source as external, read the methods and precautions for importing external key material, and check the confirmation box.



Create Key

Key Name *

Description

Tag

If there is no desired tag or tag value, you can create one in the Console

Key Usage: Symmetric Encryption/Decryption

Key Material Source: ● KMS ○ External

(4) Click **Confirm** to create the external CMK. You can view the created external CMK in the console, with the "Key Source" displayed as "External".

- **API Invocation Method**
  This example uses Tencent Cloud Command Line Tool (TCCLI), but you can use any supported programming language for subsequent calls.
  Specify the Type parameter as 2 when requesting the CreateKey API, and execute the command as follows.

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

CreateKey function source code example:

```
def create_external_key(client, alias):
    """
    Generate a Bring Your Own Key (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: Obtain the parameters for importing key material

To ensure the security of the key material, you need to encrypt it before importing. You can obtain the parameters for importing key material through the API, which includes a public key for encrypting the key material and an import token.
Run the following command on TCCLI:

```
tccli kms GetParametersForImport --KeyId <keyid> --WrappingAlgorithm RSAES_PKCS1_V
```

Sample source code of the `GetParametersForImport` function:

```
def get_parameters_for_import(client, keyid):
    """
    Obtain the parameters for importing customer master key (CMK) material.
    The returned token serves as one of the parameters for executing the ImportKeyMater
    The returned PublicKey is used to encrypt the key material for self-import.
    The returned Token and PublicKey will expire after 24 hours. If you need to re-import a
    WrappingAlgorithm specifies the algorithm for encrypting key material, currently supp
    WrappingKeySpec specifies the type of encryption for the key material, currently only
    """
    try:
        req = models.GetParametersForImportRequest()
        req.KeyId = keyid
        req.WrappingAlgorithm = 'RSAES_PKCS1_V1_5' # RSAES_PKCS1_V1_5 | RSAES_OAE
        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

Encrypt your key material locally using the encryption public key obtained in step 2. The encryption public key is a 2048-bit RSA public key, and the encryption algorithm used must be consistent with the one specified when obtaining the import key material parameters. Since the API returns a base64-encoded encryption public key, you need to perform base64 decoding before using it. Currently, KMS supports encryption algorithms RSAES_OAEP_SHA_1, RSAES_OAEP_SHA_256, and RSAES_PKCS1_V1_5.

The following is a test example of encrypting key material using OpenSSL. In actual use, it is recommended to encrypt the key material with a hardware security module (HSM) or other more secure encryption measures.

(1) Call the GetParametersForImport interface to obtain the Token and PublicKey. Write the PublicKey to the file public_key.base64.

(2) Generate random numbers 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**
> For the SM2 version, the key material length must be 128, while for the FIPS version, it must be 256.

(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

# 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

# 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
```

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

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

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

Step 4: Import Key Material

Finally, import the encrypted key material and the import token obtained in step 2 into the external CMK created in step 1, using API operations.

- The import token is bound to the public key of the encrypted key material, and a token can only import key material for the master key specified during its generation. The validity period of the import token is 24 hours, and it can be reused within this period. After expiration, a new import token and encryption public key must be obtained.
- If you call GetParametersForImport multiple times to obtain import materials, only the token and publicKey from the last call will be valid, and the previously returned values will automatically expire.
- You can import key material for external keys that have never had key material imported before, as well as re-import expired and deleted key material, or reset the expiration time of the 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> -
```

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 import of the external key is complete. You can use the external CMK just like any other regular 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.

Schedule CMK Deletion

Delete the external CMK using the scheduled deletion feature. The scheduled deletion operation requires a mandatory waiting period of 7 to 30 days. Once the expiration time is reached, the external CMK will be permanently deleted. Deleted CMKs cannot be recovered, and data encrypted with them will be irretrievable. Please proceed with caution.

Delete Key Material

There are two ways to delete key material. Once the key material has expired or been deleted, the external CMK will no longer be usable, and ciphertext encrypted by that CMK cannot be decrypted unless you re-import the same key material.

- Complete the process by using the API operation DeleteImportedKeyMaterial. After the key material is deleted, the key status will change to "Pending Import."
- In the Import Key Material API operation, set the ValidTo input parameter to specify an expiration time. The KMS service will automatically delete the key material when it reaches the expiration time.

> **Note**
> The effect of waiting for the key material to expire and become invalid is the same as manually deleting the key material.

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**

- When you import key material into a CMK, it becomes permanently associated with that material, meaning you cannot import other key materials into the external CMK. If you need to re-import key material after deleting it, the imported material must be identical to the deleted material for the import to be successful.
- When encrypting data with an external CMK, the encrypted data must be decrypted using the same CMK used during encryption (i.e., the CMK's metadata and key material must match the imported key). Otherwise, decryption will fail. Please handle key material and CMK deletion operations with caution.

White-box Key Management Overview

Last updated：2023-08-24 11:46:43

Tencent Cloud Key Management System (KMS) offers a white-box key management solution. White-box cryptography is a technique that provides resistance against white-box attacks, ensuring the security of encrypted data and keys even when an attacker has complete control over the encryption device terminal, can observe and modify internal data during program execution, and can perform reverse analysis of cryptographic operations. Compared to traditional encryption techniques, white-box cryptography makes it difficult for keys to be extracted in a white-box environment. Additionally, white-box key management supports device binding authentication capabilities, ensuring the security protection of sensitive key information and encrypted data.

White-box keys are used to protect sensitive root key information on endpoints, such as API SecretKeys, authentication keys or tokens used by internal systems, and other local sensitive root key information, providing an end-to-end, comprehensive data security solution. The white-box key management solution integrates keys and algorithms, and by introducing randomization factors, effectively conceals the keys, significantly increasing the difficulty of key sniffing and cracking, thereby protecting this highly sensitive information.

Features

High security

Based on high-strength obfuscation and reinforcement algorithms and multi-layer security protection technologies, KMS white-box encryption guarantees the security of keys used for cryptographic operations on untrusted devices.

Dynamic White Box

An original key can be converted to a white-box key through the same white-box encryption technology, which enables dynamic key rotation without changing the white-box library required.

Multiple Algorithms

KMS white-box encryption supports the globally popular algorithm AES and mainstream Chinese algorithm SM4 to meet the encryption compliance needs in different scenarios.

Multi-Platform support

It is suitable for various platforms such as Windows and Linux.

Support for device fingerprint binding

Device fingerprint information can be bound to achieve enhanced protection of decryption keys. Keys can take effect only on the specified devices, and decryption operations cannot be performed on other devices.

System Requirements

Currently, the white-box key management service supports the following operating systems. It cannot run properly on other operating systems.

- Supported Linux versions: Centos6, Centos7, and Ubuntu. Note: The glibc version must be 2.14 or higher.
- Supported Windows versions: Windows 7, Windows 10, Windows Server 2012, Windows Server 2016.

How to Use

Log in to the Key Management System (Compliant) Console white-box key management page, click **Activate Now** to enter the white-box key purchase page, and start using it after payment. For detailed operation guidance, please refer to the white-box key User Guide.

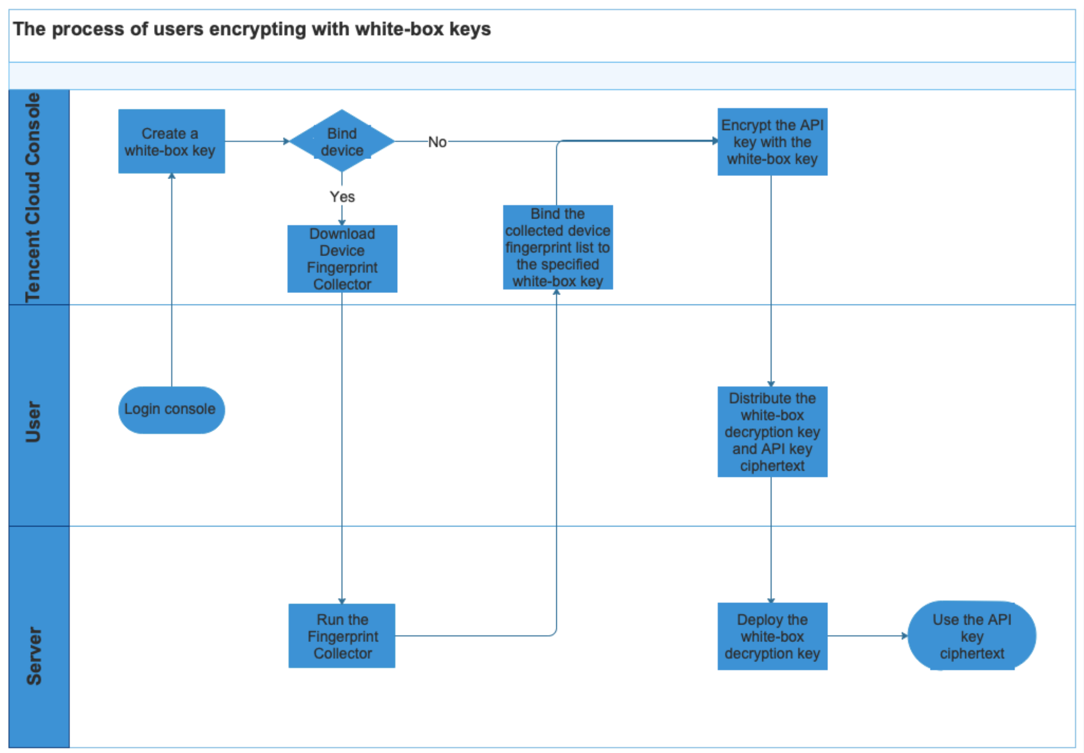
Operation Guide

Last updated：2023-08-24 17:09:38

Flowchart

API keys are a common type of key used to establish a trusted connection and data communication between client applications and backend servers. The following example demonstrates the white-box key protection solution provided by Tencent Cloud using API keys.

The process of users encrypting with white-box keys is shown in the following diagram:


The process of users encrypting with white-box keys
Tencent Cloud Console
User
Server
Create a white-box key
Bind device
No
Yes
Download Device Fingerprint Collector
Bind the collected device fingerprint list to the specified white-box key
Encrypt the API key with the white-box key
Login console
Distribute the white-box decryption key and API key ciphertext
Run the Fingerprint Collector
Deploy the white-box decryption key
Use the API key ciphertext

As can be seen from the above process, the API key is not exposed in an open environment. In the actual user's business server, only the ciphertext of the API key and the required white-box decryption key are saved. The white-box decryption algorithm and key obfuscation do not require additional plaintext keys for decryption, ensuring the security of the entire link. In the case of API key leakage events on GitHub, adopting a white-box key management solution can effectively prevent the plaintext API key from being exposed in the source code, thereby preventing malicious exploitation of the API key.

Instructions

Step 1. Create a white-box key

> **Note**

White-box keys are chargeable. For more information, see Billing Overview and Purchase Methods.

The creation of a white-box key pair is achieved by calling the white-box service through the console or API.
In this example, we use Tencent Cloud's Command Line Tool (TCCLI). You can use any supported programming language to make API calls in the future. Request command:

```
tccli kms CreateWhiteBoxKey --region ap-guangzhou --Alias test-gz01 --Description 'this is
```

Sample returned result:

```
{
   "Response": {
      "RequestId": "55f7ca05-17af-4bfb-87a5-80a80a4b0761",
      "EncryptKey": "BA4AADZhzTBr7vmCDyHwQRCKTSCXm8pnGs38mJfA1Pw+VaP8/MW
      "DecryptKey": "CA4AAEFLh2SUfMm1UIuxgTiD7g/rBKIxShZ/5jxRByvOqgtNyx42vDGA
      "KeyId": "1c820b96-73bd-11ea-a490-5254006d0810"
   }
}
```

⚠ **Note**

- Currently, supported encryption algorithms include SM4 and AES_256.
- The returned `EncryptKey` and `DecryptKey` are the encryption key and decryption key respectively, and they are both Base64-encoded.

Step 2: Bind device fingerprint (optional)

1. Log in to the machine that requires device fingerprint collection, run the fingerprint collection tool (getDeviceFingerprint), and obtain a 70-character string, which is the device fingerprint.
2. Upload and bind the collected device fingerprint list to the specified white-box key through the Key Management System Console or API. For detailed instructions and operation steps, please refer to the Device Binding Guide.

   ⓘ **Note**

   Please download the fingerprint collection tool (getDeviceFingerprint) directly from the Key Management System Console.

Request command:

```
tccli kms OverwriteWhiteboxDeviceFingerprints --region ap-guangzhou --KeyId 1c820b
```

Sample returned result:

```
{
   "Response": {
      "RequestId": "55f7ca05-17af-4bfb-87a5-80a80a4b0761",
    }
}
```

> **Note**
> DeviceFingerprints will completely replace the fingerprints of the specified key. If DeviceFingerprints is empty, it means deleting all device fingerprints associated with that key.

Step 3. Base64-encode the plaintext

Prepare the plaintext that needs to be managed with white-box key management and perform base64 encoding on it. For example, suppose the plaintext to be encrypted is: 1234567890. Using the openssl command, the result after generating the base64 encoding is: `MTIzNDU2Nzg5MAo=` .

```
echo 1234567890 | openssl base64
```

Step 4. Encrypt the API key with the white-box key

Request command:

```
tccli kms EncryptByWhiteBox --region ap-guangzhou --KeyId a1a9376a-7261-11ea-a490-
```

Response:

```
{
   "Response": {
      "RequestId": "1bf315d1-3b20-4089-b458-51c367967b4b",
      "InitializationVector": "EUi3Vv7DiCf73D6XbVzMYg==",
      "CipherText": "HKyXV1Xoodi1P/sdf/cYLw=="
   }
}
```

Note

There are two main fields returned:

- InitializationVector: randomly generated initialization vector (Iv), which is used to increase the difficulty of cracking the ciphertext.
- CipherText: Base64-encoded ciphertext after encryption.

Step 5. Distribute the white-box decryption key and API key ciphertext

The administrator distributes the DecryptKey, InitializationVector, and CipherText generated in the above steps to the developers or operations personnel of each business system. The DecryptKey will be deployed to the corresponding business system's files, while the InitializationVector and CipherText will be used as parameters for the SDK.
In the description:

1. The decryption key (DecryptKey) needs to be saved in the form of a binary file for the system to read during startup. The command is as follows:

   1.1 Save the `DecryptKey` to a file.

   ```
   echo "CA4AAEFLh2SUfMm1UIuxgTiD7g/rBKlxShZ/5jxRByvOqgtNyx42vDGACEtPYZ
   ```

   1.2 Base64-decode the public key to get its content:

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

   1.3 Place the generated binary file decrypt_key.bin in the specified directory `decrypt_key_bin_dir` on the same server as the business system (which has already integrated the decryption SDK).

2. Use `InitializationVector`, `CipherText`, `decrypt_key_bin_dir`, and `decrypt_key.bin` in the form of strings as the input parameters of the `whitebox_decrypt` method in the SDK.

Step 6. Deploy the white-box decryption key

Each business system chooses to download the language-specific decryption SDK in the corresponding programming language and integrates the SDK into the business system.

Step 7. Use the API key ciphertext

In the business logic, call the decryption function of the SDK (whitebox_decrypt) and pass in the parameters: decrypt_key_bin_dir, decrypt_key.bin, InitializationVector, CipherText, and algorithmType to obtain the decrypted plaintext. The algorithmType is the algorithm type used when generating the key, with values of 0 or 1. 0 represents AES_256, and 1 represents SM4. For information on how to decrypt with white-box keys, please refer to White-box Key Decryption Code Example. Detailed code examples are available for each language SDK.

Device Binding Guide

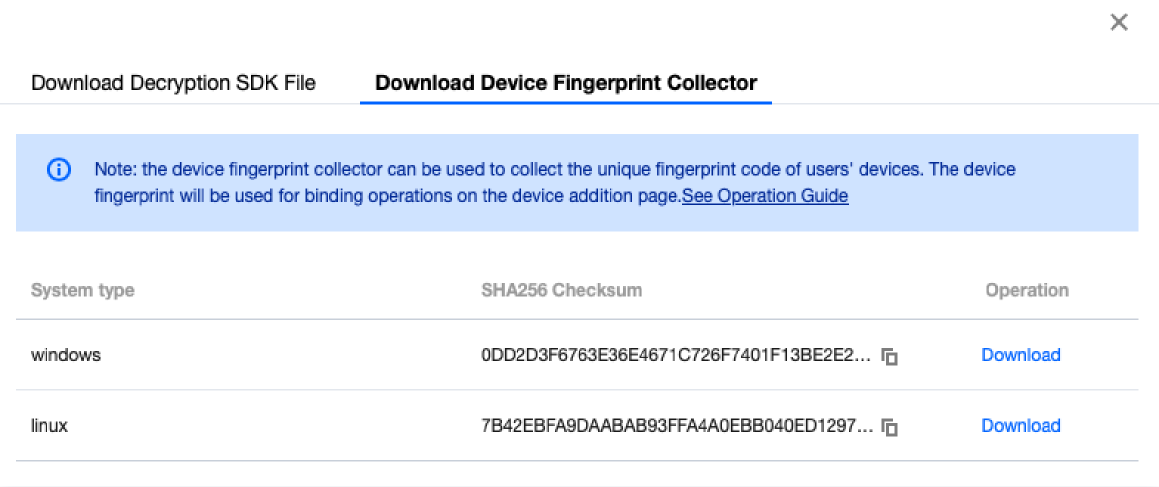
Last updated：2023-08-24 17:10:01

Overview

Device fingerprints are unique identifiers extracted from physical hosts about hardware devices, similar to human fingerprints, with different devices possessing distinct fingerprints. White-box keys support integration with device fingerprints, enabling binding between the white-box key and a specific device or group of devices. This ensures that decryption operations can only be executed on devices with established binding relationships, further strengthening the security of sensitive information.

Instructions

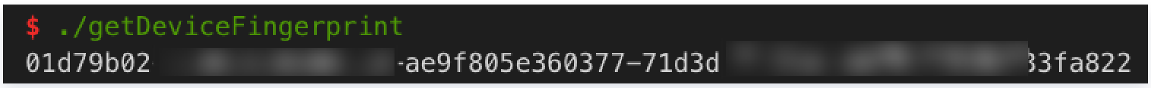
1. The administrator navigates to the Key Management System user key management page.
2. Click **Download Device Fingerprint Collector** to open the download dialog box. Based on the type of operating system for the device to be bound, click **Download** to obtain the corresponding device fingerprint collection tool.


×
Download Decryption SDK File
Download Device Fingerprint Collector

Note: the device fingerprint collector can be used to collect the unique fingerprint code of users' devices. The device fingerprint will be used for binding operations on the device addition page.See Operation Guide

System type	SHA256 Checksum	Operation
windows	0DD2D3F6763E36E4671C726F7401F13BE2E2...	Download
linux	7B42EBFA9DAABAB93FFA4A0EBB040ED1297...	Download

3. The administrator distributes the device fingerprint collection tool to system maintenance personnel and informs them of the devices requiring fingerprint collection.
4. System operators log in to the operating system of the corresponding device, run the device fingerprint collection tool, and collect device fingerprints as shown in the following figure:



```
$ ./getDeviceFingerprint
01d79b02 [redacted] -ae9f805e360377-71d3d [redacted] 3fa822
```

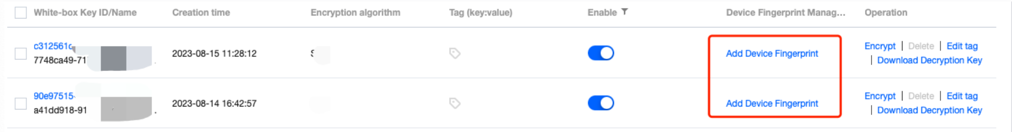
Note

> The Device Fingerprint Collector tool supports Windows and Linux operating systems and must be run on the host operating system. It currently supports physical machines and Tencent Cloud CVMs but does not support Docker environments.

5. System maintenance personnel provide the collected device fingerprint list to the administrator.

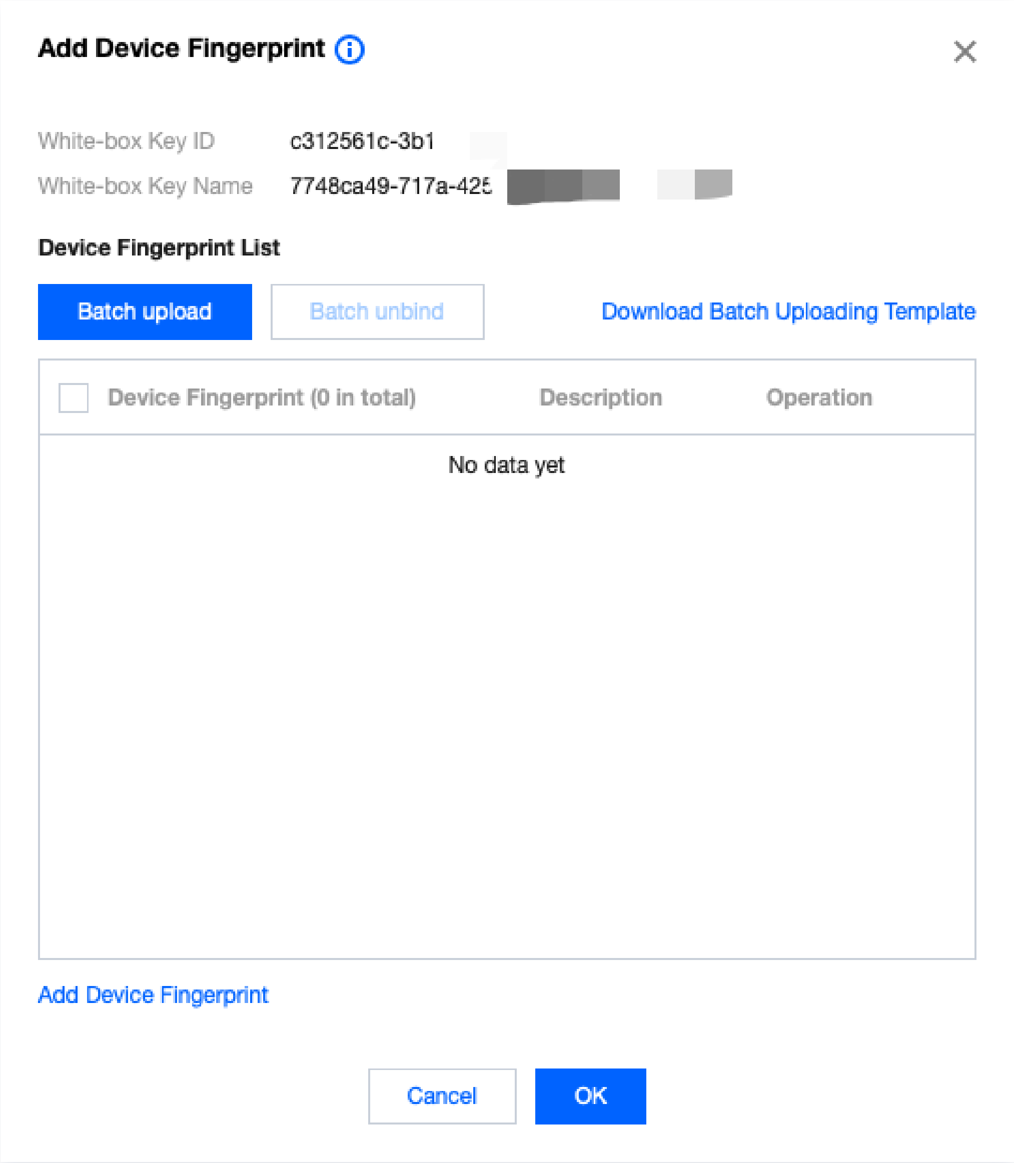
> **Note**
> Depending on the actual application scenario and access control strategy, the system administrator and console administrator can be the same person.

6. On the Key Management System user key management page, the administrator selects the white-box key to be bound with a fingerprint.
   - If it is a newly created white-box key or has not been bound to a device fingerprint before, click **Add Device Fingerprint**.
   - If a device fingerprint has already been bound, click **Manage Device Fingerprints.**



□ White-box Key ID/Name	Creation time	Encryption algorithm	Tag (key:value)	Enable	Device Fingerprint Manag...	Operation
□ c312561c [redacted] 7748ca49-71 [redacted]	2023-08-15 11:28:12	[redacted]		●	Add Device Fingerprint	Encrypt | Delete | Edit tag | Download Decryption Key
□ 90e97515 [redacted] a41dd918-91 [redacted]	2023-08-14 16:42:57	[redacted]		●	Add Device Fingerprint	Encrypt | Delete | Edit tag | Download Decryption Key

7. In the pop-up dialog box, enter the collected device fingerprints and provide a description for each fingerprint. There are two methods for inputting this information:
   - Click **Add Device Fingerprint** and enter the information directly on the page:



Add Device Fingerprint ⓘ ✕

White-box Key ID c312561c-3b1 [redacted]

White-box Key Name 7748ca49-717a-425 [redacted]

Device Fingerprint List

Batch upload | Batch unbind | Download Batch Uploading Template

□ Device Fingerprint (0 in total)	Description	Operation
No data yet		

Add Device Fingerprint

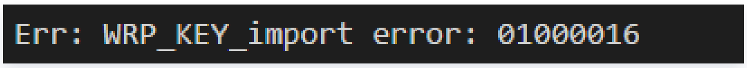
Cancel | OK

- ○ Click **Download Batch Uploading Template**, enter the information in the downloaded CSV file, and save the file. Click **Batch Upload**, select the saved CSV file, and upload the fingerprint information in bulk.
  CSV text content example:

```
Device Fingerprint, Description
123456,test description
```

8. After completing the input, click **OK** to bind the white-box key with the specified device fingerprint list.
9. After completing the device fingerprint binding, running decryption operations on non-bound devices will result in an error with the error code 01000016, as shown in the

following image:



```
Err: WRP_KEY_import error: 01000016
```

Supports and Limits

1. Device binding is an optional enhancement to the white-box key functionality.
    - If no device binding has been established, white-box decryption operations can be executed on any device.
    - If devices are bound, white-box decryption operations can only be executed correctly on the bound devices.
2. **Device fingerprint binding must be performed before downloading the decryption key,** otherwise the binding will not take effect.
3. The Device Fingerprint Collector tool supports Windows and Linux operating systems and must be run on the host operating system. It currently supports physical machines and Tencent Cloud CVMs but does not support Docker environments.

Best practices for protecting SecretKey with KMS white-box key

Last updated: 2023-08-24 17:10:26

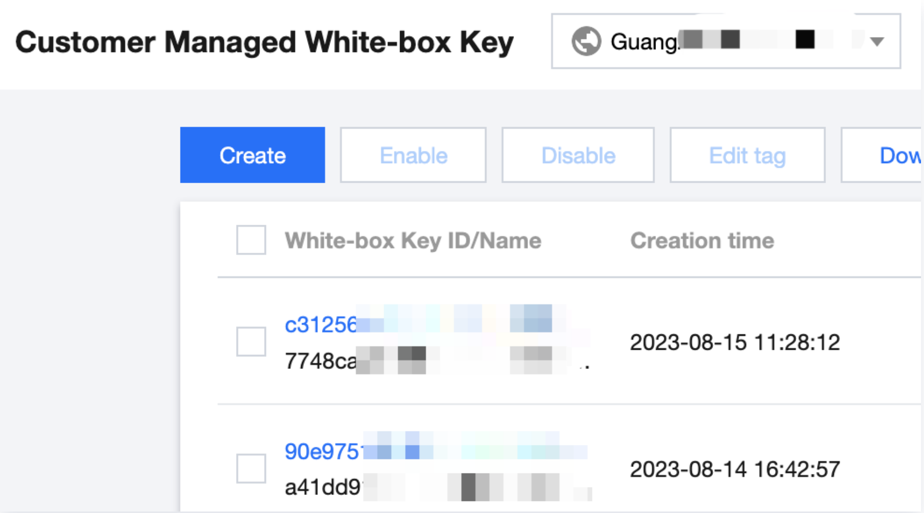
This document presents an example of using white-box key encryption and decryption for API SecretKey, with detailed steps as follows:

Key management and distribution

Step 1. Create a white-box key

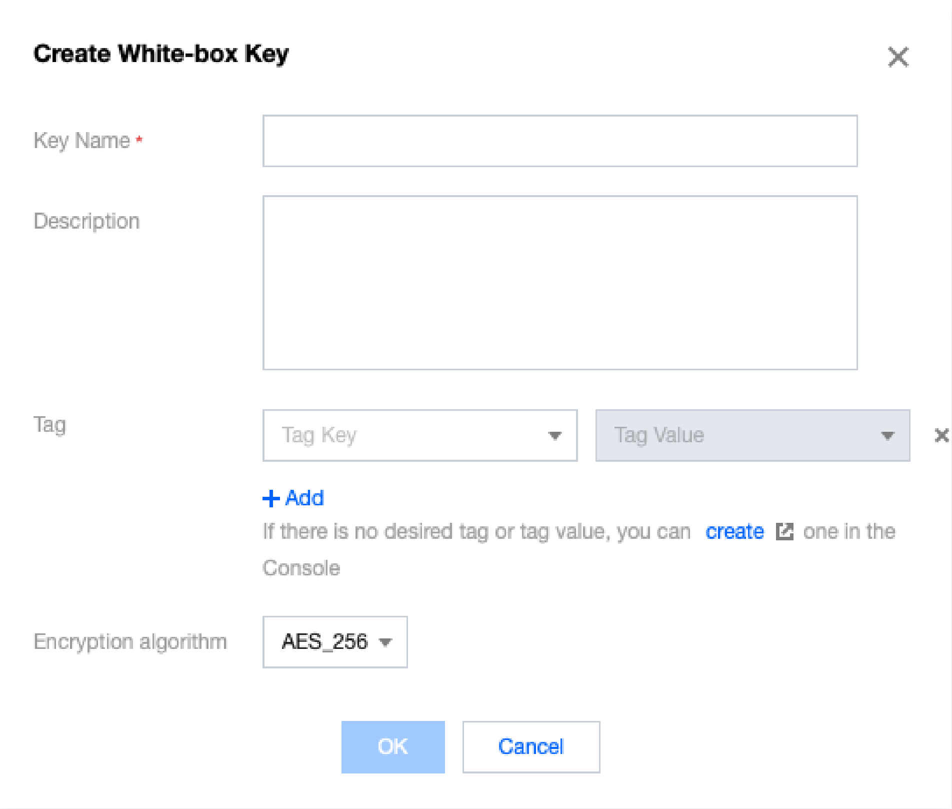
> ⚠ **Note**
>
> - White-box keys are chargeable. For more information, see Billing Overview and Purchase Methods.
> - Creating a white-box key pair is achieved by invoking the white-box service. Both console and API methods are supported. This example uses the console method.

1. Log in to the Key Management System (Compliant) Console, select **White-box Key Management** > **User Key Management** from the left menu, switch the "Region" according to your business needs, and click **Create.**



□	White-box Key ID/Name	Creation time
□	c31256[redacted] 7748ca[redacted]	2023-08-15 11:28:12
□	90e975[redacted] a41dd9[redacted]	2023-08-14 16:42:57

2. In the pop-up dialog box, enter the white-box key name, select the encryption algorithm, and provide the description and tags (both optional). Click **Confirm** to complete the creation of the white-box key.



Create White-box Key

Key Name *

Description

Tag: Tag Key, Tag Value

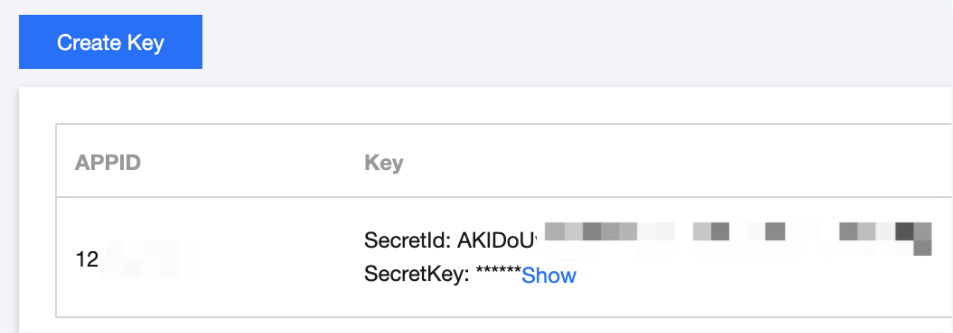
+ Add

If there is no desired tag or tag value, you can create one in the Console

Encryption algorithm: AES_256

Step 2. Obtain the API SecretKey from the console

1. Log in to the API Key Management Console with your root account to view your API keys.
2. In the key operation column, click **Show**, complete the identity verification, and obtain and copy the SecretKey.



APPID	Key
12[redacted]	SecretId: AKIDoU[redacted] SecretKey: ******Show

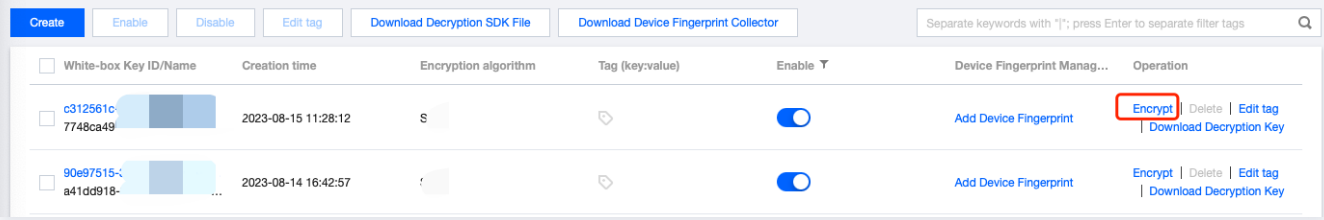
Step 3. Perform base64 encoding on the plaintext SecretKey

Encode the SecretKey content obtained in Step 2 using base64. For example, if the plaintext SecretKey to be encrypted is: lY9Ynrabcdj05YH1234LE370HOM, use the openssl command to generate the base64-encoded result: `bFk5WW5yYWJjZGowNVlIMTIzNExFMzcwSE9NCg==` .

```
echo lY9Ynrabcdj05YH1234LE370HOM | openssl base64
```

Step 4. Encrypt the API SecretKey using the white-box key

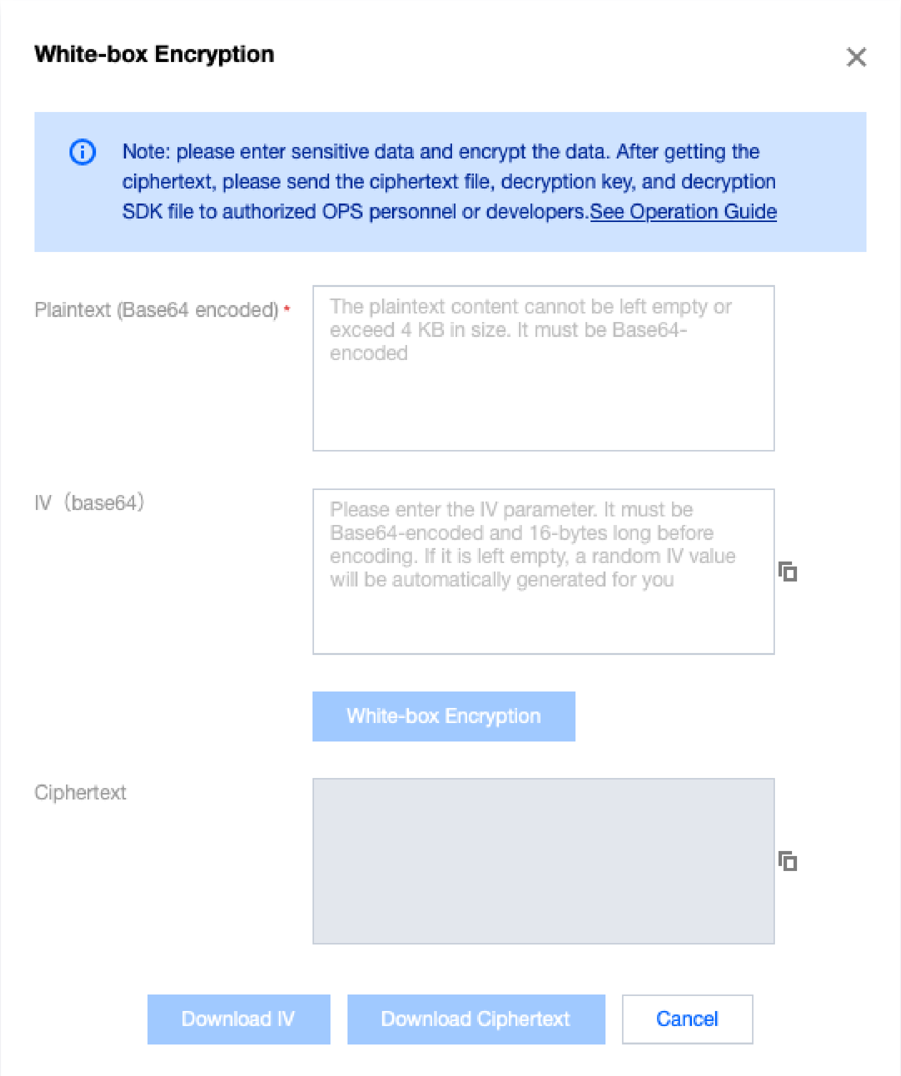
1. Log in to the Key Management System (Compliant) Console. In the white-box key list, click on the "White-box Key ID/Name" or **Encrypt** in the "Operation" column.



Create
Enable
Disable
Edit tag
Download Decryption SDK File
Download Device Fingerprint Collector
Separate keywords with "|"; press Enter to separate filter tags

□ White-box Key ID/Name	Creation time	Encryption algorithm	Tag (key:value)	Enable	Device Fingerprint Manag...	Operation
□ c312561c 7748ca49	2023-08-15 11:28:12	S		●	Add Device Fingerprint	Encrypt | Delete | Edit tag | Download Decryption Key
□ 90e97515- a41dd918- ...	2023-08-14 16:42:57			●	Add Device Fingerprint	Encrypt | Delete | Edit tag | Download Decryption Key

2. In the pop-up dialog, fill in the encoded content obtained in Step 3 into the plaintext (base64) text box, and click **White-box Encryption**.



White-box Encryption

Note: please enter sensitive data and encrypt the data. After getting the ciphertext, please send the ciphertext file, decryption key, and decryption SDK file to authorized OPS personnel or developers.See Operation Guide

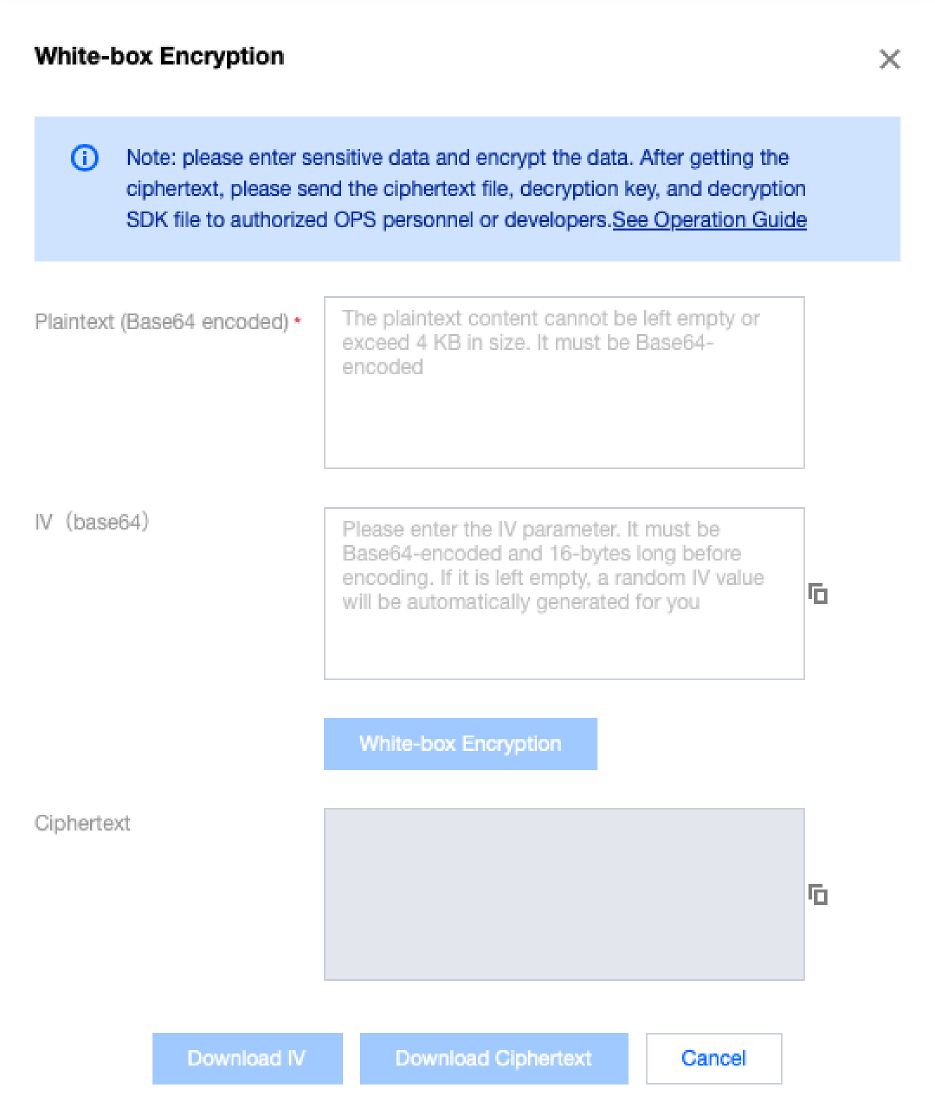
Plaintext (Base64 encoded) *: The plaintext content cannot be left empty or exceed 4 KB in size. It must be Base64-encoded

IV（base64）: Please enter the IV parameter. It must be Base64-encoded and 16-bytes long before encoding. If it is left empty, a random IV value will be automatically generated for you

Ciphertext:

3. After successful encryption, the randomly generated initialization vector (IV) and the encrypted ciphertext will be returned. Click **Download IV** and **Download Ciphertext** to complete the content download.

Note
Both the initialization vector (abbreviated as IV) and the encrypted ciphertext have been base64 encoded.



White-box Encryption

Note: please enter sensitive data and encrypt the data. After getting the ciphertext, please send the ciphertext file, decryption key, and decryption SDK file to authorized OPS personnel or developers.See Operation Guide

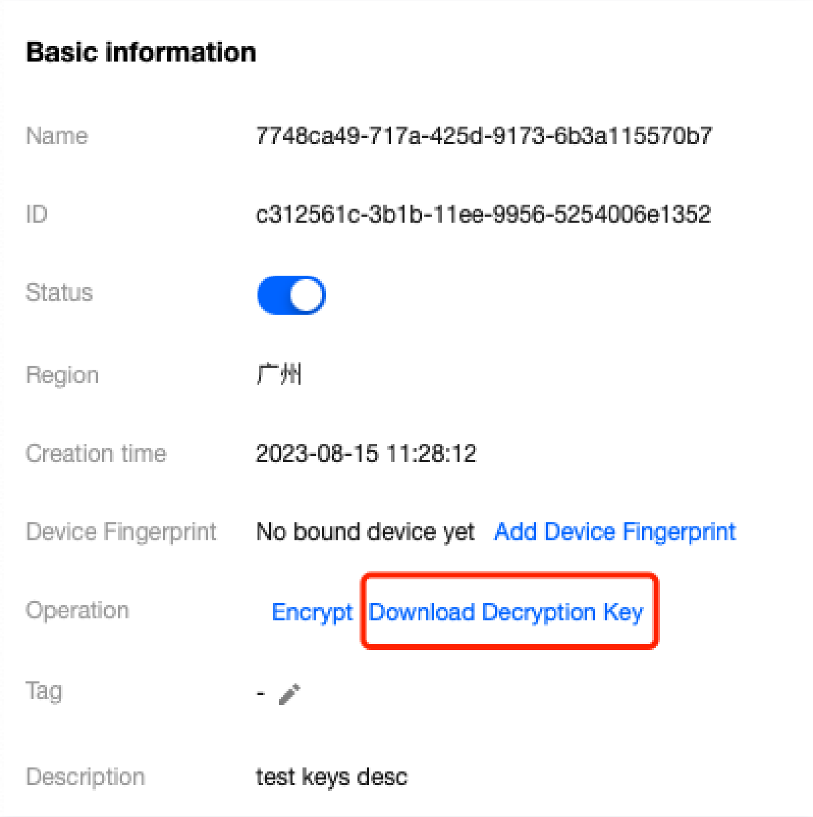
Plaintext (Base64 encoded) *: The plaintext content cannot be left empty or exceed 4 KB in size. It must be Base64-encoded

IV（base64）: Please enter the IV parameter. It must be Base64-encoded and 16-bytes long before encoding. If it is left empty, a random IV value will be automatically generated for you

Ciphertext

Step 5. Download the decryption key

1. Log in to the Key Management System (Compliance) Console. In the white-box key list, click "White-box Key ID/Name" to access the key's basic information page.
2. On the Key Basic Information page, click **Download Decryption Key** and name it decrypt_key_sm4.bin.

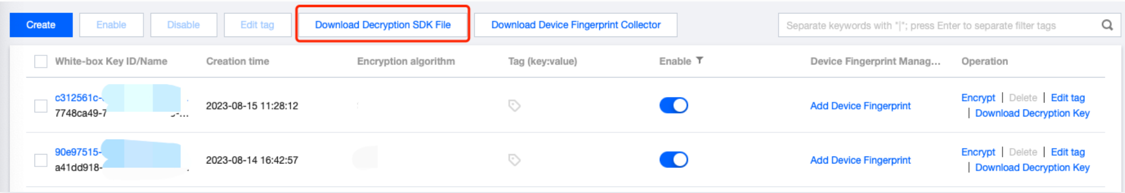


Basic information

Name	7748ca49-717a-425d-9173-6b3a115570b7
ID	c312561c-3b1b-11ee-9956-5254006e1352
Status	●
Region	广州
Creation time	2023-08-15 11:28:12
Device Fingerprint	No bound device yet Add Device Fingerprint
Operation	Encrypt Download Decryption Key
Tag	-
Description	test keys desc

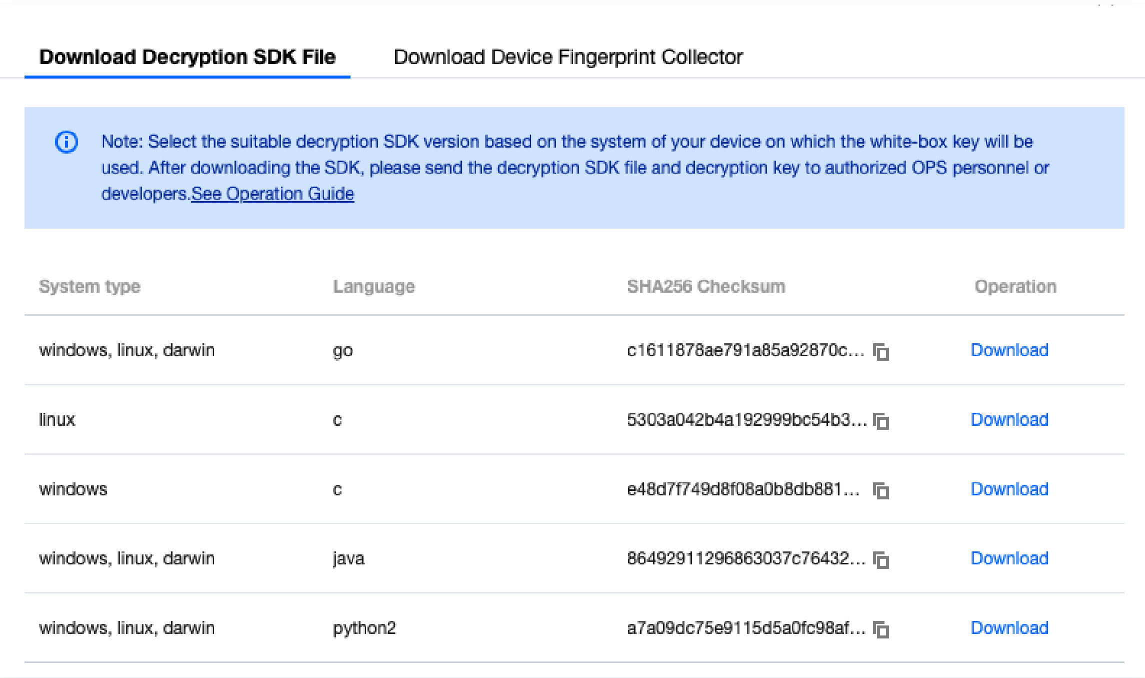
Step 6: Download the decryption SDK file

1. Log in to the Key Management System (Compliance) Console , in the white-box key list, click **Download Decryption SDK File** on the right.


Create
Enable
Disable
Edit tag
Download Decryption SDK File
Download Device Fingerprint Collector
Separate keywords with "|"; press Enter to separate filter tags

□ White-box Key ID/Name	Creation time	Encryption algorithm	Tag (key:value)	Enable	Device Fingerprint Manag...	Operation
□ c312561c-[redacted] 7748ca49-7[redacted]...	2023-08-15 11:28:12			●	Add Device Fingerprint	Encrypt | Delete | Edit tag | Download Decryption Key
□ 90e97515-[redacted] a41dd918-[redacted]	2023-08-14 16:42:57			●	Add Device Fingerprint	Encrypt | Delete | Edit tag | Download Decryption Key

2. In the pop-up window, select and download the decryption SDK for the programming language used by your business system, and integrate the SDK into your system.



Download Decryption SDK File
Download Device Fingerprint Collector

Note: Select the suitable decryption SDK version based on the system of your device on which the white-box key will be used. After downloading the SDK, please send the decryption SDK file and decryption key to authorized OPS personnel or developers.See Operation Guide

System type	Language	SHA256 Checksum	Operation
windows, linux, darwin	go	c1611878ae791a85a92870c...	Download
linux	c	5303a042b4a192999bc54b3...	Download
windows	c	e48d7f749d8f08a0b8db881...	Download
windows, linux, darwin	java	86492911296863037c76432...	Download
windows, linux, darwin	python2	a7a09dc75e9115d5a0fc98af...	Download

Step 7: Distribute white-box decryption key and API SecretKey ciphertext

The administrator distributes the decryption key, IV, and ciphertext files downloaded in the above steps to the developers or operators of each business system. The decryption key is deployed to the corresponding business system files, while the initialization vector IV and ciphertext are used as parameters for the SDK.

> **Note**
>
> The downloaded decryption key is a binary bin file, which needs to be placed on the same server as the executable file (already integrated with the decryption SDK). The file path will be used as the decryption parameter for the SDK.
> For example, in the Code Sample, the specified directory is ./data, which means the key should be placed in the data subdirectory under the same parent directory as the executable file.

Service Integration

Utilizing the encrypted API SecretKey

- In the business logic, call the SDK decryption function (whitebox_decrypt) with the following parameters: decrypt_key_bin_dir (the directory where the decryption key is stored in Step 7), decrypt_key_sm4.bin (the decryption key downloaded in Step 5, with its corresponding filename), InitializationVector (the IV downloaded in Step 4), CipherText (the ciphertext of the SecretKey encrypted with white-box in Step 4), and algorithmType to obtain the decrypted plaintext.

- The `algorithmType` is the algorithm type used when generating the key, with values of 0 or 1. A value of 0 represents AES_256, while 1 represents SM4.
- For information on how to decrypt using white-box keys, please refer to White-box Key Decryption Code Examples. Detailed code examples are available for each language SDK.

White-box key decryption code example

Last updated：2023-08-24 11:47:21

This document provides example code for four programming languages: Golang, Python, C, and Java.

Sample decryption code for Go

```
package main

import (
  "encoding/base64"
  "fmt"
  "unsafe"
)

func main() {
  fmt.Println("--------- test case for AES_256 ---------")
  // Directory where the decryption key file is located
  decryptKeyFileDirectoryName := "./data"
  // Decryption key file name
  decryptKeyFileName := "decrypt_key_aes256.bin"
  // Initialization vector, Base64 encoded
  iv := "EUi3Vv7DiCf73D6XbVzMYg=="
  // Encrypted ciphertext using white-box key, Base64 encoded
  cipherText := "HKyXV1Xoodi1P/sdf/cYLw=="
  // Encryption algorithm used when creating the white-box key, 0: AES_256, 1: SM4
  algoType := 0
  fmt.Println(fmt.Sprintf("demo start for decryptKeyFileName=%s, iv=%s, cipherText=%s

  whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv, cipherText, al

  fmt.Println("--------- test case for SM4 ---------")
  // Directory where the decryption key file is located
  decryptKeyFileDirectoryName = "./data"
  // Decryption key file name
  decryptKeyFileName = "decrypt_key_sm4.bin"
  // Initialization vector, Base64 encoded
  iv = "9+COkyNOrT8mvWN6CgTjKw=="
  // Encrypted ciphertext using white-box key, Base64 encoded
  cipherText = "83ji4vKFwtVSAN1LSh1aOQ=="
```

```
    // Encryption algorithm used when creating the white-box key, 0: AES_256, 1: SM4
    algoType = 1
    fmt.Println(fmt.Sprintf("demo start for decryptKeyFileName=%s, iv=%s, cipherText=%s

    whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv, cipherText, al

    fmt.Println("--------- test cases finished ---------")
}
```

Sample decryption code for Python

```
#!/usr/bin/python

import os
from ctypes import *
import base64

if __name__ == "__main__":
    print("--------- test case for AES_256 ---------")
    # Directory where the decryption key file is located
    decryptKeyFileDirectoryName = "../data";
    # Decryption key file name
    decryptKeyFileName = "decrypt_key_aes256.bin"
    # Initialization vector, Base64 encoded
    iv = "EUi3Vv7DiCf73D6XbVzMYg=="
    # Encrypted ciphertext of the white-box key, Base64 encoded
    cipherText = "HKyXV1Xoodi1P/sdf/cYLw=="
    # Encryption algorithm used when creating the white-box key, 0: AES_256, 1: SM4
    algoType = 0
    try:
        print("decryptKeyFileName=%s, iv=%s, cipherText=%s, algoType=%d" % (decryptK
        plain = demo_clt_cbc_dec(decryptKeyFileDirectoryName, decryptKeyFileName, base
        print ("decrypt success")
        print ("plain: %s" % plain)
    except YdwbCryptoException as e:
        print (e.msg)

    print("--------- test case for SM4 ---------")
    # Directory where the decryption key file is located
    decryptKeyFileDirectoryName = "../data";
    # Decryption key file name
    decryptKeyFileName = "decrypt_key_sm4.bin"
    # Initialization vector, Base64 encoded
    iv = "9+COkyNOrT8mvWN6CgTjKw=="
    # Encrypted ciphertext of the white-box key, Base64 encoded
    cipherText = "83ji4vKFwtVSAN1LSh1aOQ=="
    # Encryption algorithm used when creating the white-box key, 0: AES_256, 1: SM4
    algoType = 1
    try:

        print("decryptKeyFileName=%s, iv=%s, cipherText=%s, algoType=%d" % (decryptK
        plain = demo_clt_cbc_dec(decryptKeyFileDirectoryName, decryptKeyFileName, base
        print ("decrypt success")
        print ("plain: %s" % plain)
    except YdwbCryptoException as e:
        print (e.msg)

    print("--------- test case finished  ---------")

    pass
```

Sample decryption code for C

The sample code divides into Windows code and Linux code:

- Windows platform: Open the demo.sln file in the vs folder. The demo uses Visual Studio 2017 and includes both static and dynamic linking demos. Simply compile and run. Due to path issues, if you use the exe in the command line, you need to copy the /data directory to the parent directory.
- Linux: you need to add `lib` to environment variables:

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:../lib
  ```

Below is the sample code:

```
#include "../include/wrp.h"
#include "base64.h"
#include <stdint.h>
#include <stdio.h>
#include <string.h>

int demo_aes()
{
        unsigned char cipher_base64[TEST_BUFF_SIZE] = "snPqPZaFN9CQc5WH/Tx5jA==";
        unsigned char cipher[TEST_BUFF_SIZE] = {0};
        int cipher_len = Base64decode(cipher, cipher_base64);
        if (cipher_len > TEST_BUFF_SIZE)
        {
```

```
                printf("base64 decode cipher text failed, memory is not enough.\n");
                return (-1);
        }

        unsigned char iv_base64[TEST_BUFF_SIZE] = "WBbaiNLcEYSbjKxoJt66UQ==";  // Ent
        unsigned char iv_bin[TEST_BUFF_SIZE] = {0};
        int iv_len = Base64decode(iv_bin, iv_base64);

        if (iv_len != 16)
        {
                printf("iv is not invalidate.\n");
                return (-1);
        }


    char * whitebox_decrypt_key = "decrypt_key_aes256.bin";   //Enter the decryption key
    whitebox_decrypt(ALG_AES, whitebox_decrypt_key, cipher, cipher_len, iv_bin);

    return 0;
}

int demo_sm4()
{
        unsigned char cipher_base64[TEST_BUFF_SIZE] = "IwNgzruYfHQ6oQz2PLdyRQ==";
        unsigned char cipher[TEST_BUFF_SIZE] = {0};
    int cipher_len = Base64decode(cipher, cipher_base64);
        if (cipher_len > TEST_BUFF_SIZE)
        {
                printf("base64 decode cipher text failed, memory is not enough.\n");
                return (-1);
        }

        unsigned char iv_base64[TEST_BUFF_SIZE] = "4qaj6cVd8msMVBqNTRG4Pg==";  // I
        unsigned char iv_bin[TEST_BUFF_SIZE] = {0};
        int iv_len = Base64decode(iv_bin, iv_base64);

        if (iv_len != 16)
        {
                printf("iv is not invalidate.\n");
                return (-1);
        }

    char * whitebox_decrypt_key = "decrypt_key_sm4.bin";   // Enter the decryption key fil
    whitebox_decrypt(ALG_SM4, whitebox_decrypt_key, cipher, cipher_len, iv_bin);
    return 0;
}

int main(int argc, const char *argv[])
{
    demo_aes();
    demo_sm4();
    return 0;
}
```

Sample decryption code for Java

```
import com.tencent.yunding.lightjce.CipherWhiteBox;
import com.tencent.yunding.lightjce.params.*;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Base64;

public class Main {

    public static void main(String[] args) throws Exception {
        System.out.println("--------- test case for AES_256 ---------");
        // Directory where the decryption key file is located
        String decryptKeyFileDirectoryName = "../data";
        // Decryption key file name
        String decryptKeyFileName = "decrypt_key_aes256.bin";
        // Initialization vector, Base64 encoded
        String iv = "EUi3Vv7DiCf73D6XbVzMYg==";
        // Encrypted ciphertext using white-box key, Base64 encoded
        String cipherText = "HKyXV1Xoodi1P/sdf/cYLw==";
        // Encryption algorithm used when creating the white-box key, 0: AES_256, 1: SM4
        int algoType = 0;
        System.out.printf("demo start for decryptKeyFileName=%s, iv=%s, cipherText=%s,
        whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv, cipherText

        System.out.println("--------- test case for SM4 ---------");
        // Directory where the decryption key file is located
        decryptKeyFileDirectoryName = "../data";
        // Decryption key file name
        decryptKeyFileName = "decrypt_key_sm4.bin";
        // Initialization vector, Base64 encoded
        iv = "9+COkyNOrT8mvWN6CgTjKw==";
        // Encrypted ciphertext using white-box key, Base64 encoded
        cipherText = "83ji4vKFwtVSAN1LSh1aOQ==";
        // Encryption algorithm used when creating the white-box key, 0: AES_256, 1: SM4
        algoType = 1;
        System.out.printf("demo start for decryptKeyFileName=%s, iv=%s, cipherText=%s,

        whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv, cipherText

        System.out.println("--------- test case finished  ---------");

    }

    public static void whitebox_decrypt(String decrypt_key_bin_dir, String fileName, String
        byte[] cipher = Base64.getDecoder().decode(CipherText);
        byte[] iv = Base64.getDecoder().decode(Iv);
        String decryptKeyFilePath = decrypt_key_bin_dir + "/" + fileName;

        if (algorithmType == 0) {
            byte[] result1 = decAESData(decryptKeyFilePath, cipher, iv);
            System.out.println("AES decrypted text length: " + result1.length);
            System.out.println("AES decrypted text       : " + new String(result1));
        } else if (algorithmType == 1) {
            byte[] result2 = decSM4Data(decryptKeyFilePath, cipher, iv);
            System.out.println("SM4 decrypted text length: " + result2.length);
            System.out.println("SM4 decrypted text       : " + new String(result2));
        }
    }

    public static byte[] decAESData(String keyFilePath, byte[] data, byte[] iv) throws Excep
        CipherWhiteBox instance = CipherWhiteBox.getInstance(SymAlgType.AES, BlockMo
        File file = new File(keyFilePath);
        instance.init(file, iv, CryptMode.decrypt_mode);
        instance.update(data);
        return instance.doFinal();
    }

    public static byte[] decSM4Data(String keyFilePath, byte[] data, byte[] iv) throws Excep
        CipherWhiteBox instance = CipherWhiteBox.getInstance(SymAlgType.SM4, BlockMo
        File file = new File(keyFilePath);
        instance.init(file, iv, CryptMode.decrypt_mode);
        instance.update(data);
        return instance.doFinal();
    }

}
```

Cloud Products Integrated with KMS Transparent Encryption

Last updated：2023-08-24 17:10:55

Overview

Tencent Cloud Key Management Service (KMS) is a secure, reliable, and user-friendly key management service that helps you effortlessly create and manage keys while ensuring their security. Tencent Cloud KMS can seamlessly integrate with most Tencent Cloud products. In products that have integrated KMS, simply select a key managed by KMS to easily encrypt and decrypt data within the cloud product.

Integrating KMS encryption with cloud products offers you the following benefits:

- Cloud products integrate KMS to encrypt user data storage, with encryption keys controlled by the user. KMS uses hardware security modules (HSMs) certified by the National Cryptography Administration or FIPS-140-2 to generate and protect keys, meeting both domestic and international compliance review standards.
- KMS provides users with a transparent encryption solution. Users only need to enable the encryption service for cloud products that have integrated KMS, without worrying about the encryption details, to achieve seamless data encryption and decryption in the cloud.
- There is no need for users to build and maintain their own key management infrastructure, reducing development costs and providing a secure and convenient user experience.

> **Note**
> Since other cloud products do not manage the keys, before using the encryption data in cloud products integrated with KMS, you need to complete the role authorization operation for KMS on the cloud product through Tencent Cloud Access Management (CAM).

Supported Key Types

Customer Master Keys (CMKs) are keys created by users or cloud products through the Key Management System, primarily used for encrypting and protecting Data Encryption Keys (DEKs). A single CMK can encrypt multiple DEKs. KMS offers two types of CMKs:

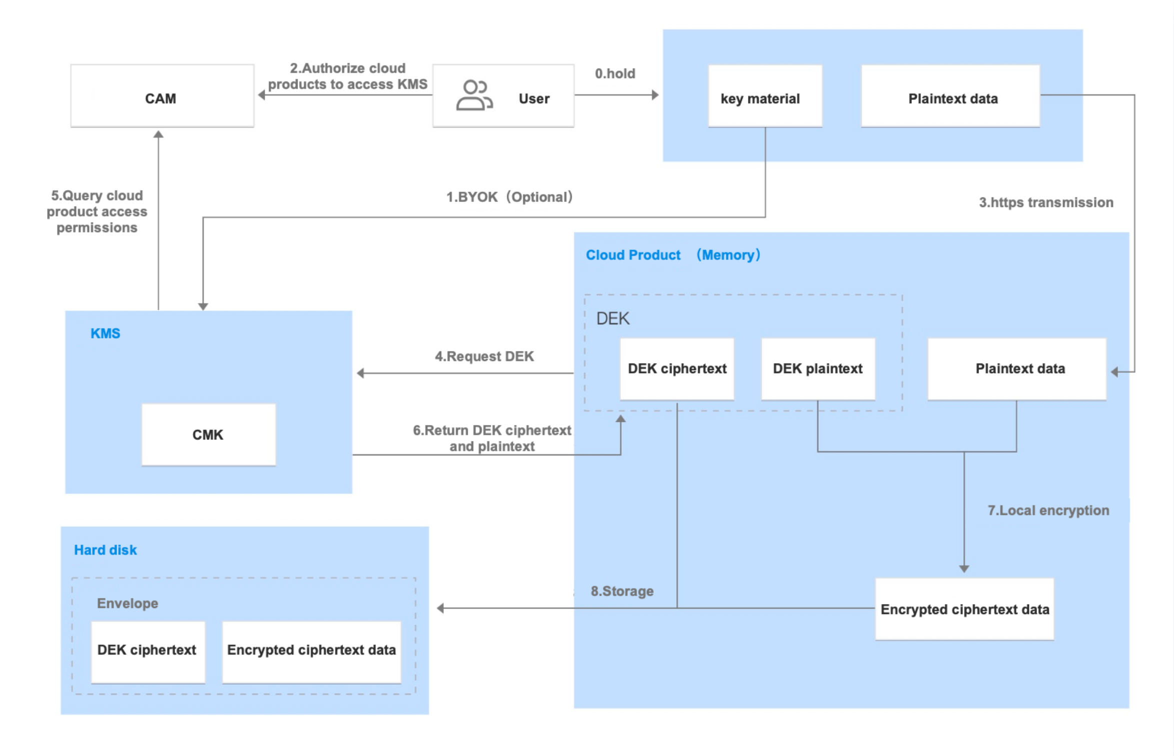
- Custom Keys
  Users can create keys independently through the Key Management System. There are two ways to obtain keys: **Created by KMS** or **Imported by the user** (BYOK). For more information, please refer to the Key Management System's Create Key and External Key Import sections.

- Default Cloud Product Keys
  When a user uses KMS encryption for the first time with a corresponding cloud product, the product automatically creates a key for the user through the Key Management System. Cloud product keys can be queried through the KMS console, but disabling and scheduled deletion operations are not supported.

Encryption Principle

The specific encryption design varies slightly for different cloud products, business forms, and customer requirements. Typically, cloud products use envelope encryption by calling KMS APIs to encrypt and decrypt data.

The principle of using KMS encryption in cloud products is illustrated in the following diagram:



The encryption process is described as follows:

1. Activate the KMS service and complete the role authorization of KMS for cloud products.
2. Create a customer master key (CMK) through KMS, allowing users to choose between the default cloud product key or a custom key.
3. By invoking the GenerateDataKey interface with a CMK, you can generate both the ciphertext and plaintext of the Data Encryption Key (DEK), with the DEK being encrypted and protected by the CMK.
4. The plaintext DEK is cached in the memory of the cloud product's backend, where it encrypts the user's data locally, resulting in ciphertext data.
5. Cloud products store the encrypted Data Encryption Key (DEK) and the encrypted

ciphertext data on disk.

Beta Version KMS Migration Guide

Last updated: 2023-08-24 11:47:47

Overview

Tencent Cloud's beta version of the Key Management System is scheduled for end-of-life (EOL) due to architectural improvements. Users are advised to migrate to the new KMS (product name: Key Management System (Compliance)) as soon as possible.
The official website has officially launched the all-new Key Management System (Compliance) as a service replacement. The new KMS fully meets compliance standards, offers a richer set of key management features, and significantly improves reliability in its design.

> ⚠ **Note**
> The beta version of the Key Management System uses the API/SDK 2017 interface to provide services. If you need to confirm whether you are using the beta version of the Key Management System, please submit a ticket to contact us.

Pricing

The Key Management System (KMS) consists of two parts: CMK storage fees and API call fees. For more details, please refer to the Billing Overview.

Step-by-step Instructions

Step 1: Users of the beta version of KMS can first reactivate the new Key Management System (Compliance) on the official website.
Step 2: Create a Customer Master Key (CMK) using the new KMS.
Step 3: Decrypt the data encrypted by the beta version of KMS: Follow the API/SDK 2017 interface specifications, use the old SDK, and call the Decrypt interface to obtain the plaintext data.
Step 4: Re-encrypt the data using the SDK of the new Key Management System (Compliance).

Sensitive Data Encryption Migration Steps

Sensitive information encryption is a core capability of the Key Management System (KMS), primarily used in practice to protect the security of sensitive data on server hard drives (less than 4KB), such as keys, certificates, configuration files, etc. For more information, please refer to Sensitive Information Encryption.

1. Activate the Key Management System (Compliance) service.
2. According to business requirements, create the corresponding customer master key (CMK) in the Key Management System (Compliance).
3. To decrypt data encrypted by the beta version of the Key Management System: Follow the API/SDK 2017 interface specifications, use the older SDK, call the Decrypt interface to obtain plaintext data, and refer to the Decryption API documentation.
4. Sensitive data encryption in the new Key Management System (Compliance): Following Tencent Cloud API 3.0 standards, use the new SDK to call the Encrypt interface for encryption. For more information, please refer to the Encryption API documentation.
5. Sensitive data decryption in the new Key Management System (Compliance): Following Tencent Cloud API 3.0 standards, use the new SDK to call the Decrypt interface for decryption. For more information, please refer to the Decryption API documentation.

Envelope Encryption Migration Steps

Envelope Encryption is a high-performance encryption and decryption solution for handling massive amounts of data. For more information, please see Envelope Encryption.

1. Activate the Key Management System (Compliance) service.
2. According to business requirements, create the corresponding customer master key (CMK) in the Key Management System (Compliance).
3. To decrypt encrypted data using the beta version of the Key Management System: You only need to handle the migration of the DataKey. According to the API/SDK 2017 interface specifications, use the old version of the SDK and call the Decrypt API to obtain the plaintext DataKey. For more details, please refer to the Decrypt API documentation.
4. The new Key Management System (Compliance) supports envelope encryption: following Tencent Cloud API 3.0 standards, it uses the new SDK to call the Encrypt API for encryption. For more information, please refer to the Encryption API documentation.
5. In the new Key Management System (Compliance), envelope decryption is performed by following Tencent Cloud API 3.0 standards, using the new SDK to call the Decrypt API to decrypt the DataKey and obtain the plaintext. The plaintext DataKey is then used to decrypt the data. For more information, please refer to the Decrypt API documentation.

Implementing Exponential Backoff to Deal with Service Frequency

Last updated：2023-08-24 11:47:56

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:** If the returned error indicates a non-temporary failure or retrying does not lead to success, you should terminate/cancel the program call and report the exception.
- **Retry:** If the returned error is uncommon or relatively rare, such as network packets being damaged during transmission but still sent, you can immediately attempt a retry in this case.
- **Delayed Retry:** If the returned error is due to common connection or busy-related issues, the service may need a short recovery time to clear accumulated workloads. In such cases, wait for an appropriate amount of time before retrying.

This article elaborates on the delayed retry strategy. The mentioned waiting time (i.e., delay time) can be implemented by gradually increasing the delay or using a timed strategy (such as exponential backoff). Since the call frequency to KMS API services is limited, you can use the delayed retry method to avoid issues caused by exceeding the rate limit when your call concurrency is too high.

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
```

Policy Implementation

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

```python
# -- coding: utf-8 --
import base64
import math
import time
import os
from tencentcloud.common import credential
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloud
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's Custom Parameters
    secretId = os.getenv('SECRET_ID')       # read from environment variable or use
    secretKey = os.getenv('SECRET_KEY')   # read from environment variable or use
    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)  # Invoke the encryption interface
        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`.
- Based on your code logic, business strategy, and other factors, plan and establish a timed strategy to set the optimal **Initial Delay Value** (InitDelayValue) and **Retry Count** (Retries). This helps avoid setting the threshold too low or too high, which may impact the overall operation of your business.