密钥管理系统

实践教程

腾讯云

文档目录

实践教程
对称加解密
敏感信息加解密
概述

最近更新时间：2024-09-10 17:42:31

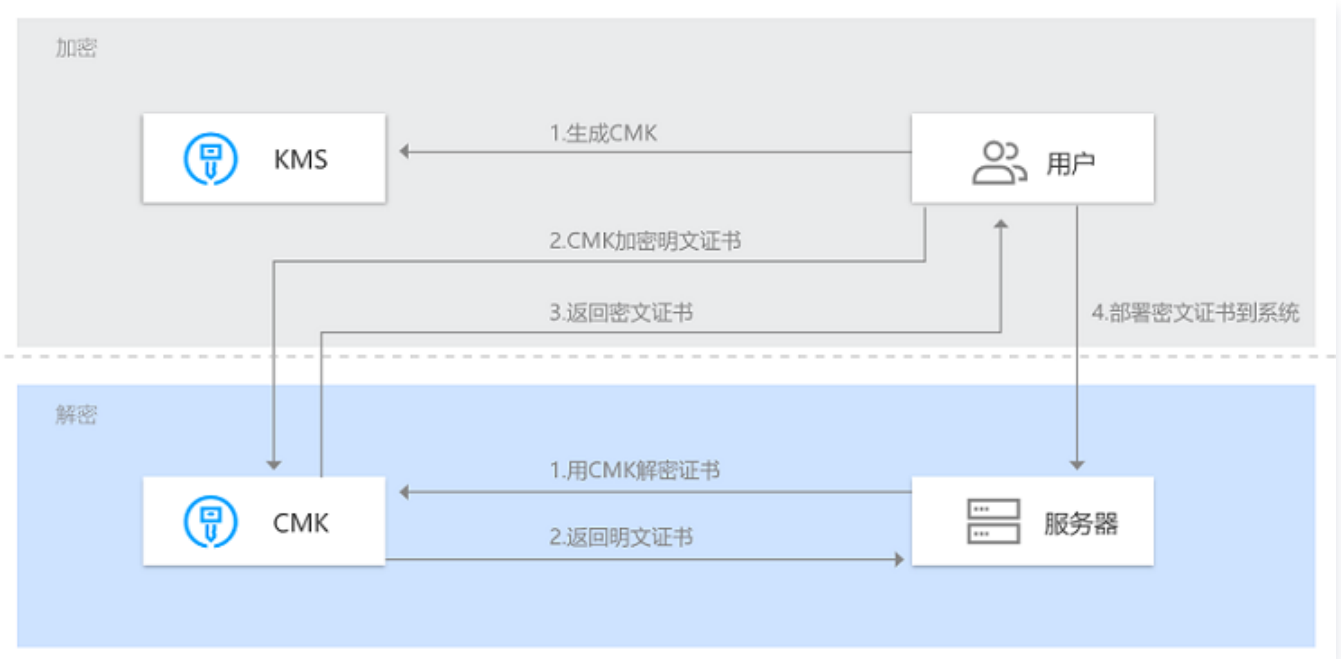
敏感信息加密是密钥管理系统 KMS 核心的能力，适用于保护小型敏感数据（小于4KB），如密钥、证书、配置文件等。使用 CMK 加密敏感数据信息，而非直接明文存储。使用时，再将密文解密到内存，保证明文不落盘。整个交互传输过程都使用 HTTPS 请求，确保敏感数据安全。
若需要通过 KMS 对海量数据进行高性能的加解密服务，请参见 信封加密 场景。

敏感信息举例

–	密钥，证书	后台配置文件
用途	加密业务数据，通信通道，数字签名	保存系统架构和其他业务信息，例如数据库 IP、密码
丢失风险	保密信息被盗、加密通道遭监听、签名被伪造	业务数据被拖库、成为攻击其他系统的跳板

示意图

本场景中，敏感数据通过主密钥 CMK 进行加解密保护，主密钥 CMK 受到经过第三方认证硬件安全模块（HSM）的保护。主密钥 CMK 在 HSM 内部实现加解密，包括腾讯云在内的任何无权限人员都无法看到 CMK 明文。



功能特点

- 权限控制：与腾讯云访问管理（CAM）集成，通过身份管理和策略管理控制账号对CMK的访问权限。

- 内置审计：与腾讯操作审计集成，可记录所有 API 请求，包括密钥管理操作和密钥使用情况。保证数据操作可溯源可审计。
- 集中化密钥管理：通过腾讯云 KMS 服务实现对各类应用程序的密钥的集中管理。
- 安全合规：密钥管理系统底层使用国家密码局或 FIPS-140-2 认证的硬件安全模块（HSM）来保护密钥的安全，确保密钥的保密性、完整性和可用性。
- 敏感数据加密：支持敏感数据（小于4KB）的加密和解密操作。如密钥、证书、配置文件等。

注意事项

- 需注意 SecretId 和 SecretKey 的保密存储：
  - 腾讯云接口认证主要依靠 SecretID 和 SecretKey，SecretID 和 SecretKey 是用户的唯一认证凭证。业务系统需要该凭证调用腾讯云接口。
- 需注意 SecretID 和 SecretKey 的权限控制：
  - 建议使用子账号，根据业务需要进行接口授权的方式管控风险。
- 需注意明文数据的存储：
  - 敏感数据加密已将数据进行加密处理，为保障数据的安全性，需确保原始明文数据的删除。

操作指南

最近更新时间：2024-09-10 17:42:31

该操作指南以 Python 为例，其它编程语言类似。

前期准备

- 示例代码依赖环境：Python 2.7。
- KMS 产品服务开通：从 腾讯云控制台 开通 KMS 产品。
- 云 API 密钥服务开通：获取 SecretID、SecretKey 以及调用地址（endpoint），endpoint 一般形式为 `*.tencentcloudapi.com`，例如 KMS 的调用地址为 `kms.tencentcloudapi.com`，具体参考各产品说明。
- SDK 安装，执行以下命令，详细可参见 tencentcloud-sdk-python github 项目。

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

操作流程

您可以通过以下4个步骤完成敏感数据加密的操作。

1. 通过控制台 Console 或 API（CreateKey）创建一个用户主密钥 CMK，即创建用户主密钥 CMK。
2. 通过 API 调用 KMS 加密接口（Encrypt）将用户敏感数据进行加密，获取密文。
3. 根据业务需求将密文数据存储。
4. 读取数据时，通过 API 调用 KMS 解密接口（Decrypt）解密成明文。

操作步骤

步骤1：创建用户主密钥 CMK

用户主密钥的创建方式请参见 创建密钥 文档。

步骤2：敏感信息加密

前提条件

确保步骤1创建的用户主密钥为启用状态。

控制台方式

在线工具适合处理单次或者非批量的加解密操作，例如首次生成密钥密文，开发者无需为非批量的加解密操作而去开发额外的工具，将精力集中在实现核心业务能力上，详情请参见 加密解密 文档。

Python SDK 方式

通过 Encrypt 来针对用户的数据进行加密，用于加密的数据大小最多为4KB任意数据，可用于加密数据库密码，RSA Key，或其它较小的敏感信息。本文示例使用腾讯云 Python SDK 实现，您也可以使用其它支持的编程语言。
该 API 操作的 KeyId 和 Plaintext 为必选参数，详情请参见 Encrypt 接口文档来查看其它参数说明。

加密 Python SDK 示例

以下示例代码展示了如何使用指定 CMK 对数据进行加密操作。

Python 代码示例

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

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

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

def Encrypt(client, keyId="", plaintext=""):
    try:
        req = models.EncryptRequest()
        req.KeyId = keyId
        req.Plaintext = base64.b64encode(plaintext)
        rsp = client.Encrypt(req) # 调用加密接口
        return rsp
    except TencentCloudSDKException as err:
        print(err)
        return None

if __name__ == '__main__':
    # 用户自定义参数
    secretId = os.getenv('SECRET_ID')          # read from environment variable or
use whitebox encryption to protect secret ID
    secretKey = os.getenv('SECRET_KEY')    # read from environment variable or use
whitebox encryption to protect secret key    region = "ap-guangzhou"
    region = "ap-guangzhou"
    keyId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    plaintext = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

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

步骤3：将加密后的数据存储

根据业务的应用场景，将密文进行存储。

步骤4：敏感数据解密

控制台方式

详情请参见 加密解密 文档。

Python SDK 方式

通过 Decrypt 来针对用户的数据进行解密。
该 API 操作的 CiphertextBlob 为必选参数，详情请参见 Decrypt 接口文档查看其它参数说明。

Python 代码示例

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

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

def Decrypt(client, keyId="", ciphertextBlob=""):
    try:
        req = models.DecryptRequest()
        req.CiphertextBlob = ciphertextBlob
        rsp = client.Decrypt(req) # 调用解密接口
        return rsp
    except TencentCloudSDKException as err:
        print(err)
        return None

if __name__ == '__main__':
    # 用户自定义参数
    secretId = os.getenv('SECRET_ID')          # read from environment variable or
use whitebox encryption to protect secret ID
```

```
    secretKey = os.getenv('SECRET_KEY')    # read from environment variable or use
whitebox encryption to protect secret key    region = "ap-guangzhou"
    region = "ap-guangzhou"
    keyId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    ciphertextBlob = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

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

信封加解密

概述

最近更新时间：2024-09-10 17:42:31

信封加密（Envelope Encryption）是一种应对海量数据的高性能加解密方案。信封加密主要使用 DEK 在客户本地进行数据的加解密，DEK 被 CMK 所保护，由 GenerateDataKey 接口生成，通过 GenerateDataKey 生成 DEK 时需要指定对应 CMK 的 KeyId。
通过 GenerateDataKey 生成 DEK 后，通常建议将 DEK 的密文进行本地落盘存储（存储于 DB 、配置文件等），在使用 DEK 进行数据解密前，通常只需要调用 KMS 的 Decrypt 接口对 DEK 密文进行解密并将得到的 DEK 明文缓存在内存中，后续针对本地海量的数据加解密，只需要使用内存中缓存的 DEK 明文作为密钥。

信封加密优点

- **减少 KMS 对云 API 的依赖**：有效降低云 API 接口调用次数，减少云 API 调用带来的网络开销，降低网络链路抖动带来的影响。
- **提升本地数据加解密性能**：因为 DEK 缓存在本地，在进行本地海量数据的加解密运算时可以有效提升本地数据加解密的性能。

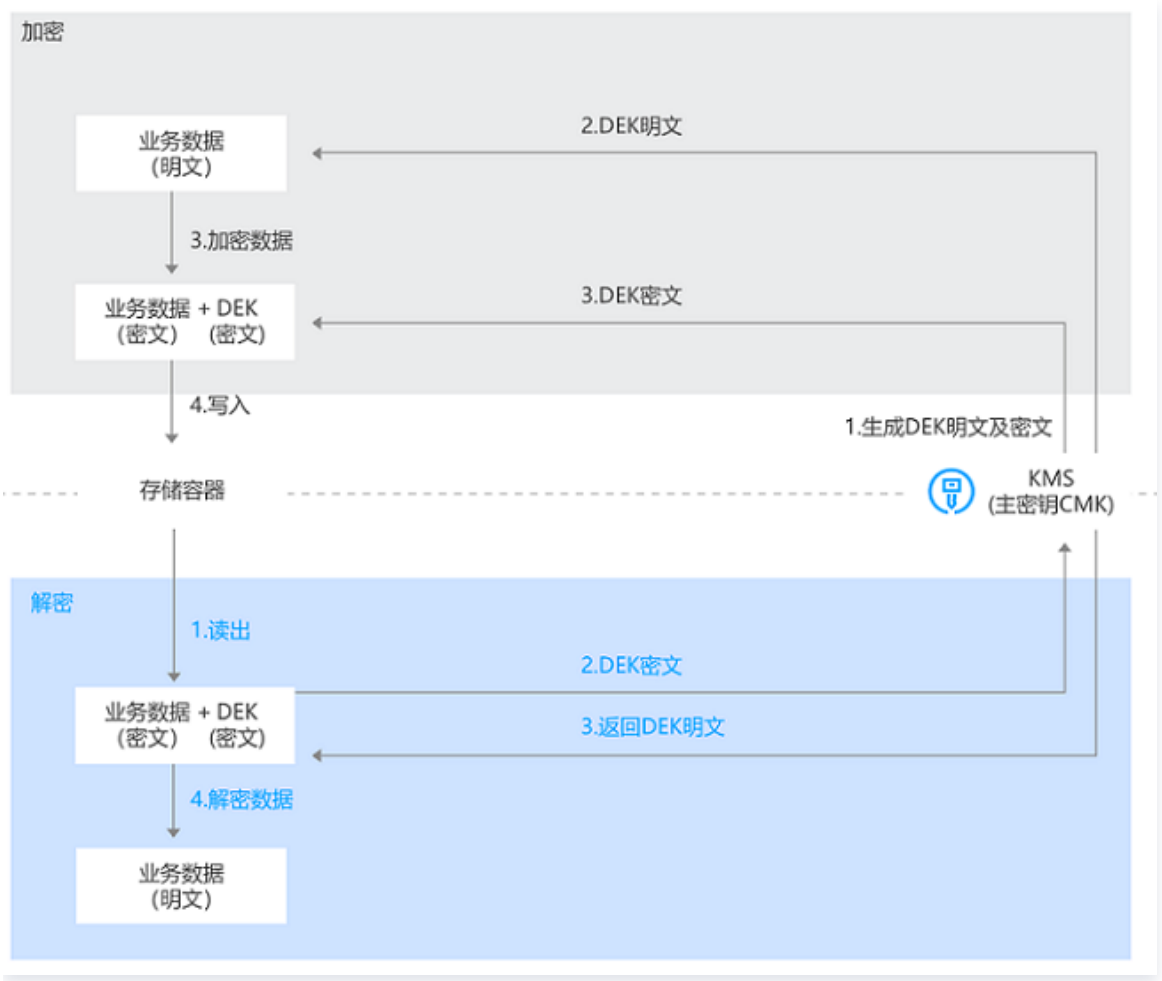
KMS 加密方案对比

对比项	敏感信息加密	信封加密
相关密钥	CMK	CMK、DEK
性能	对称加密，远程调用	少量远程对称加密，海量本地对称加密
主要场景	密钥、证书、小型数据，适用于调用频率较低的场景	海量大型数据，适用于对性能要求较高的场景
云 API 调用	每次加解密都需要调用云 API	进程启动后调用1次云 API 对 DEK 密文进行解密

示意图

本场景中，KMS 生成的 CMK 作为重要资源，通过 CMK 生成和获取 DEK 的明文和密文。用户根据实际业务场景，首先在内存中通过 DEK 明文来对本地数据进行加密，然后将 DEK 密文和密文数据落盘，其次在业务解密场景中需通过 KMS 来解密 DEK 密文，最后通过解密出来的 DEK 明文在内存中解密。

> **注意：**
> 注意在实际的使用中，请勿将 DEK 的明文信息进行日志打印。



功能特点

- 高效：所有的业务数据都是采用高效的本地对称加密处理，对业务的访问体验影响很小。而对于 DEK 的创建和加解密开销，除了非常极端的情况下，您需要采用"一次一钥"的方案，大部分场景下可以在一段时间内复用一个 DEK 的明文和密文，所以大多数情况下这部分开销非常小。
- 安全易用：信封加密的安全性由 KMS 密钥安全提供保障。由DEK 保护业务数据，而腾讯云 KMS 则保护 DEK 并提供更好的可用性，您的主密钥主要用来生成 DEK，并且只有具备密钥访问权限的对象才能操作。

注意事项

- 需注意 SecretId 和 SecretKey 的保密存储：
  - 腾讯云接口认证主要依靠 SecretID 和 SecretKey，SecretID 和 SecretKey 是用户的唯一认证凭证。业务系统需要该凭证调用腾讯云接口。
- 需注意 SecretID 和 SecretKey 的权限控制：
  - 建议使用子账号，根据业务需要进行接口授权的方式管控风险。
- 需注意业务系统对明文密钥的处理：
  - 信封加密场景中采用的是对称加密，故明文密钥不可落盘，需在业务流程的内存中使用。
- 需注意后台系统对数据密钥的处理：
  - 信封加密场景中采用的是对称加密，可根据业务需求复用同一个数据密钥或针对不同用户、不同时间使用不同的数据密钥进行加密，避免 DEK 重复。

操作指南

最近更新时间：2024-09-10 17:42:31

本实践指南以 Python 为例，其它语言类似。

前期准备

- 示例代码依赖环境：Python 2.7。
- KMS 产品服务开通：从 腾讯云控制台 开通 KMS 产品。
- 云 API 密钥服务开通：获取 SecretID、SecretKey 以及调用地址（endpoint），KMS 的调用地址为 `kms.tencentcloudapi.com`，具体参考各产品说明。
- SDK 安装：执行以下命令，详细可参见 tencentcloud-sdk-python github 开源项目。

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

操作流程

您可以通过以下3个步骤完成信封加密场景的操作。

1. 创建主密钥 CMK。
2. 数据信封加密，业务程序通过 API 调用 KMS GenerateDataKey 接口生成数据密钥，系统通过明文密钥将数据加密，并将密文密钥和密文落盘。
3. 数据读取解密，系统读取密文密钥和密文，通过 KMS 解密接口解密密文密钥，返回明文密钥，最后通过明文密钥解密密文数据。

实践步骤

步骤1：创建主密钥 CMK

主密钥 CMK 的创建指南，请参见 创建密钥 快速入门文档。

步骤2：数据信封加密

根据业务需求，在需要新的 DEK 时（例如针对新的用户需要进行加密，或者 DEK 复用超过一定时间，使用新的 DEK 等），可通过 KMS 接口创建新的数据密钥。**生成数据密钥后在内存中使用明文密钥加密，最后将密文和密文密钥落盘。**

生成数据密钥并对用户数据进行加密

通过 GenerateDataKey 来获取数据密钥 DEK，数据加密密钥是基于 CMK 生成的二级密钥，可用于用户本地数据加密解密。KMS 对 DEK 不做保存管理，需要调用方进行存储。
本文示例使用腾讯云 Python SDK 实现，您也可以使用其它支持的编程语言调用。
该 API 操作的 KeyId 为必选参数，您可以查看 GenerateDataKey 接口文档来查看其它参数说明。

Python SDK 示例

```
# -*- 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
TencentCloudSDKException
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.kms.v20190118 import kms_client, models

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

def GenerateDatakey(client, keyId, keyspec='AES_128'):
    try:
        req = models.GenerateDataKeyRequest()
        req.KeyId = keyId
        req.KeySpec = keyspec
        # 调用生成数据密钥接口
        generatedatakeyResp = client.GenerateDataKey(req)
        # 明文密钥需要在内存中使用，密文密钥用于持久化存储
        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)

# 用户自定义逻辑，此处仅作为参考
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__':
    # 用户自定义参数
    secretId = os.getenv('SECRET_ID')          # read from environment variable or
use whitebox encryption to protect secret ID
    secretKey = os.getenv('SECRET_KEY')    # read from environment variable or use
whitebox encryption to protect secret key    region = "ap-guangzhou"
    region = "ap-guangzhou"
```

```
    keyId = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    keySpec = "AES_256"
    plaintext = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

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

    LocalEncrypt(rsp.Plaintext, plaintext)
```

步骤3：数据读取解密

先读取落盘的密文密钥，并通过调用解密接口，解密密文密钥。最后根据解密出的明文密钥解密数据。

解密（KMS Python SDK）

通过 Decrypt 来针对用户的数据进行解密。
本文示例使用腾讯云 Python SDK实现，后续您可以使用任何支持的编程语言调用。
该 API 操作的 CiphertextBlob 为必选参数，您可以查看 Decrypt 接口文档来查看其它参数说明。

Python SDK 示例

将 DEK 密文密钥通过调用 KMS 解密接口进行解密，然后用获取的 DEK 明文解密用户数据密文。

```
# -*- 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
TencentCloudSDKException
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.kms.v20190118 import kms_client, models

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

def DecryptDataKey(client, ciphertextBlob):
    try:
        req = models.DecryptRequest()
        req.CiphertextBlob = ciphertextBlob
        rsp = client.Decrypt(req) #调用解密接口对 DEK 解密
        return rsp
    except TencentCloudSDKException as err:
```

```
        print(err)

# 用户自定义逻辑，此处仅作为参考
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__':
    # 用户自定义参数
    secretId = os.getenv('SECRET_ID')          # read from environment variable or
use whitebox encryption to protect secret ID
    secretKey = os.getenv('SECRET_KEY')    # read from environment variable or use
whitebox encryption to protect secret key    region = "ap-guangzhou"
    region = "ap-guangzhou"
    dekCipherBlob="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    ciphertext="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

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

    LocalDecrypt(rsp.Plaintext, ciphertext)
```

非对称加解密
概述

最近更新时间：2024-09-10 17:42:31

非对称加解密需要两个密钥：**公开密钥**和**私有密钥**，这两个密钥在密码学中是一对且具有双向性，即公钥和私钥中的任一个均可用作加密，但只能由另一个进行解密 。公开密钥可以交给任何人，即使对方是不可信任的，而私有密钥必须自行秘密保管。
相比对称加密，非对称加密无需考虑采用可靠的通道进行密钥分发，通常应用在信任等级不对等的系统之间 ，实现敏感数据加密传递或数字签名验签。

非对称密钥的类型

当前腾讯云 KMS 支持如下三种非对称密钥算法类型：

RSA

目前 KMS 支持模长为2048比特的 RSA 密钥，KeyUsage = ASYMMETRIC_DECRYPT_RSA_2048。

SM2

SM2 是国密标准的公钥密钥算法，在中国的商用密码体系中被用来替换 RSA 算法。对于有相应的国密要求的应用，可以考虑使用此类型的密钥，KeyUsage = ASYMMETRIC_DECRYPT_SM2。

ECC

ECC 是基于椭圆曲线的加密算法，KeyUsage = ASYMMETRIC_SIGN_VERIFY_ECC。

非对称加密的典型场景

非对称加解密在实际应用中包含**加密通信**和**数字签名**两种典型场景：

加密通信

加密通信是非对称加密算法的一种典型应用。加密通信的过程类似于对称加密，区别在于需要使用公钥进行加密，而且需要使用私钥进行解密。
加密通信场景的原理说明如下：
1. 信息接收者创建公钥私钥对，并将公钥给到一个或多个信息发送者。
2. 信息发送者使用公钥对敏感信息进行加密，并将加密后的密文通过传输介质发送给信息接收者。
3. 信息接收者从传输介质上获取到数据后，用自己持有的私钥对信息进行解密，还原出原文信息。

由于密文只有通过私钥才可以解密，而私钥是不公开的，所以即使由于传输介质的安全性比较低而导致信息泄露，拿到密文的人也无法将其破译，从而保证了敏感信息的安全。
腾讯云 KMS 提供加密通信的解决方案，具体操作详情请参见 非对称数据加解密 。

数字签名

数字签名技术是非对称加密算法的另一种典型应用。数字签名分为签名和验证两个过程，在签名时使用私钥，验证时则使用公钥，这一实现过程正好与加密通信相反。
数字签名场景的原理说明如下：
1. 信息发送者创建公钥私钥对，并将公钥给到一个或多个信息接收者。

2. 信息发送者使用哈希函数从消息原文中生成消息摘要，然后使用私有密钥对这个摘要进行加密，即得到消息原文对应的数字签名。
3. 信息发送者将消息原文和数字签名一并传送给信息接收者。
4. 信息接收者得到消息原文和数字签名后，用同一个哈希函数从消息原文中生成摘要 A，另外，用发送者提供的公钥对数字签名进行解密，得到摘要 B，对比 A 和 B 是否相同，验证原文是否被篡改。

由于签名是使用私钥加密产生，而私钥不公开，这使得签名具有唯一的特征。所以数字签名既可以保证数据在传输过程中不会被篡改，又可以保证信息发送者的身份正确性，防止交易中的抵赖发生。
腾讯云 KMS 提供数字签名解决方案，具体操作详情请参见 非对称签名验签 。

> **注意：**
> 由于公私钥使用场景的特殊性，KMS 不支持对非对称的 CMK 进行自动轮转。如果您需要定期或不定期更新所使用的密钥，可以自行创建新的非对称密钥。

RSA 非对称数据加解密

最近更新时间：2024-09-10 17:42:31

操作流程

当您需要进行传递敏感信息时（例如密钥的交换），需要对敏感数据进行加密，使用非对称密钥加解密的方案从信息接收者的角度来说，您需要进行以下操作：

1. 在密钥管理系统 KMS 中创建非对称加密密钥，详情请参见 创建主密钥 。
2. 在密钥管理系统 KMS 中获取公钥，详情请参见 获取非对称密钥的公钥 。
3. 信息接收者将公钥分发给信息发送者。
4. 信息发送者用得到的公钥对敏感数据进行本地加密后，将密文发送给信息接收者。
5. 信息接收者拿到密文后，调用 KMS 的解密功能对密文解密。API 详情请参见 非对称密钥 Sm2 解密 和 非对称密钥 RSA 解密 ，TCCLI 方式请参见 非对称密钥解密 。

在整个敏感数据的传输的过程中，使用密文进行传输，而唯一能解密该密文的密钥托管在密钥管理系统 KMS 中保护，包括腾讯云在内的任何人都无法获取到您的密钥，极大程度上提高了敏感数据加密传输安全性。

操作步骤

RSA 示例

1. 创建非对称加密密钥

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

返回结果：

```
{
    "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. 下载公钥

请求：

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

返回结果：

```
{
    "Response": {
        "RequestId": "408fa858-cd6d-4011-b8a0-653805******",
        "KeyId": "22d79428-61d9-11ea-a3c8-525400******",
        "PublicKey":
"MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeUc5aO9TfiDplIO4Wo
vBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFiTx3O87wdKWcF2vHL9Ja+95VuCmKYeK1uh
Pyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCxDhwj08b24T+/FYB9a4icuqQypCvjY1X9j8ivAsPEdHZoc9Di7J
XBTZdVeZC1igCVgl6mwzdHTJCRydE2976zyjC7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6RbxebLqtmN
ypNERIR7jTCt9L+fgYOX7anmuF5v7z0GfFsen9Tqb1LsZuQR0vgqCauOj************",
        "PublicKeyPem": "-----BEGIN PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzQk7x7ladgVFEEGYDbeU\nc5aO9TfiDplIO
4WovBOVpIFoDS31n46YiCGiqj67qmYslZ2KMGCd3Nt+a+jdzwFi\nTx3O87wdKWcF2vHL9Ja+95VuCmKY
eK1uhPyqqj4t9Ch/cyvxb0xaLBzztTQ9dXCx\nDhwj08b24T+/FYB9a4icuqQypCvjY1X9j8ivAsPEdHZ
oc9Di7JXBTZdVeZC1igCV\ngl6mwzdHTJCRydE2976zyjC7l6QsRT6pRsMF3696N07WnaKgGv3K/Zr/6R
bxebLq\ntmNypNERIR7jTCt9L+fgYOX7anmuF5v7z0GfFsen9Tqb1LsZuQR0************\n1QIDAQA
B\n-----END PUBLIC KEY-----\n"
    }
}
```


3. 使用公钥加密

3.1 将公钥 PublicKey 存入文件 public_key.base64，并进行 base64解码。

存入文件：

base64 解码获取公钥实际内容：

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

3.2 创建测试明文文件：

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

3.3 使用 OPENSSL 进行公钥加密 test_rsa.txt 文件内容。

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

3.4 将公钥加密后的数据进行 base64 编码，方便传输。

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

4. 通过 KMS 使用私钥解密

将上述 encrypted.base64 base64 编码之后的密文作为 AsymmetricRsaDecrypt 的 Ciphertext 参数，进行私钥的解密。

请求：

```
tccli kms AsymmetricRsaDecrypt --KeyId 22d79428-61d9-11ea-a3c8-525400****** --
Algorithm RSAES_OAEP_SHA_256 --Ciphertext
"DEb/JBmuhVkYS34r0pR7Gv1WTc4khkxqf7S1WIr7/GXsAs/tfP/v/2+1SwsIG7BqW7kUZqr38/FGkaIE
qYeewot37t3+Jx0t5w7/yXkUnyUfyfPpXlHXf94g3wFOjijEWWsjWWzaXTkTr8uWOfRBenq+bcaY783FI
y03XjJW/Y0wKWjD3tULvKndCJO/3bkb65kn1Fbsfm20xrUUwqV/p2DVLXBdG1ymr0DjsbG7R0tb3ytc2L
mH33YPAQE32eP27ciKzSml+w2tdUM3dw3nEZcTGMs1wFDGk0O1WB052jZ7TitUD9zCftFv2dKlZD3LRx1
+vHqpNVgPhLmL******=="
```

返回结果：

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

说明：

使用 SM2 非对称密钥加解密流程类似，私钥解密接口详情请参见 非对称密钥Sm2解密 。

非对称签名验签

概述

最近更新时间：2024-05-08 18:36:31

在传递敏感信息的场景中，信息发送者可以通过非对称签名验签的方式提供身份证明，具体操作流程如下：

1. 在密钥管理系统 KMS 中创建非对称密钥对，详情请参见 创建主密钥 。
2. 信息发送者使用创建的用户私钥对需要传输的数据生成签名，详情请参见 签名 。
3. 信息发送者将签名和数据传递给信息接收者。
4. 信息接收者拿到签名和数据之后，进行签名验证，有以下两种方式进行校验。
   4.1 调用 KMS 的验证签名功能接口进行校验，详情请参见 验证签名 。
   4.2 下载 KMS 非对称密钥公钥，在本地通过 gmssl、openssl、密码库等验签方法进行验证。

> **说明：**
> 非对称签名验签目前支持 SM2/RSA/ECC 签名验签算法。

SM2 签名验签

最近更新时间：2024-09-10 17:42:31

本文将为您介绍如何使用 SM2 签名验签算法。

操作步骤

步骤1：创建非对称签名密钥

> **注意**
> 在密钥管理系统（KMS）中调用 创建主密钥 接口创建用户主密钥时，在 KMS 中创建密钥的时候，必须传入正确的密钥用途 KeyUsage= ASYMMETRIC_SIGN_VERIFY_SM2，才可以使用签名的功能。

请求：

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

返回结果：

```
{
    "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******"
    }
}
```

步骤2：下载公钥

请求：

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

返回结果：

```
{
    "Response": {
        "RequestId": "408fa858-cd6d-4011-b8a0-653805******",
        "KeyId": "22d79428-61d9-11ea-a3c8-525400******",
```

```
        "PublicKey":
"MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJahujq+PvM**********
*****bBs/f3axWbvgvHx8Jmqw==",
        "PublicKeyPem": "-----BEGIN PUBLIC KEY-----
\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJa\nhujq+PvM*******
********bBs/f3axWbvgvHx8Jmqw==\n-----END PUBLIC KEY-----\n"
    }
}
```

将公钥 PublicKeyPem 转成 pem 格式，并存入 public_key.pem 文件：

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJa
hujq+PvM**************bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

> **说明**
> 另外，您可以登录 KMS 控制台，单击**用户密钥 > 密钥ID/密钥名称**进入密钥信息页面，直接下载非对称密钥公钥。

步骤3：创建信息的明文文件

创建测试明文文件：

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

> **注意**
> 当生成的文件内容中，存在不可见的字符情况下（如换行符等），需对文件进行truncate操作（**truncate -s -1 test_verify.txt**），从而保证签名准确。

步骤4：计算消息摘要

- 如果待签名的消息的长度不超过4096字节，可以跳过本步骤，直接进入 步骤 5 。
- 如果待签名的消息的长度超过4096字节，则需先在用户端本地计算消息摘要。
  使用 gmssl 对 test_verity.txt 文件内容进行摘要计算：

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

步骤5：通过 KMS 签名接口生成签名

调用 KMS 的 签名 API 接口计算信息的签名。

1. 消息原文或消息摘要进行计算签名之前，需先进行 base64 编码。

```
//消息摘要进行base64编码
gmssl enc -e -base64 -A -in digest.bin -out encoded.base64
```

```
//消息原文进行base64编码
gmssl enc -e -base64 -A -in test_verify.txt -out encoded.base64
```

2. 进行签名的计算。

请求：

```
// 将上述 encoded.base64 的文件内容作为 SignByAsymmetricKey 的 Message 参数，以消息摘要的形式进行签名
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm SM2DSA --Message "qJQj83hSyOuU7Tn0SRReGCk4yuuVWaeZ44BP******==" --MessageType DIGEST

// 以消息原文的形式进行签名（原文要进行Base64编码）
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm SM2DSA --Message "dG***Ao=" --MessageType RAW
```

返回结果：

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

将签名内容 Signature 存入 signContent.sign 文件：

```
echo "U7Tn0SRReGCk4yuuVWaeZ4******" | base64 -d > signContent.bin
```

步骤6：验证签名

- 通过 KMS 验证签名接口校验(**推荐使用该方法进行验签**)

请求：

```
// 对消息原文进行验证（原文要进行Base64编码）
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "dG***Ao=" --Algorithm SM2DSA --MessageType RAW
// 对消息摘要进行验证（将步骤4 encoded.base64 文件内容作为 VerifyByAsymmetricKey 的 Message 参数，以消息摘要的形式进行验签）
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ**********=" --Algorithm SM2DSA --MessageType DIGEST
```

> **说明**
> 签名接口和验签接口中使用的参数 Message 和 MessageType 的取值要保持一致。

返回结果：

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

- **通过 KMS 公钥和签名内容在本地进行验证**

  请求：

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

  返回结果：

```
Signature Verification Successful
```

RSA 签名验签

最近更新时间：2024-09-10 17:42:31

本文将为您介绍如何使用 RSA 签名验签算法。

操作步骤

步骤1：创建非对称签名密钥

> **注意：**
> 在 KMS 中调用 创建主密钥 接口创建用户主密钥时，必须传入正确的密钥用途 KeyUsage= ASYMMETRIC_SIGN_VERIFY_RSA_2048，这样才可以使用签名功能。

- **请求：**

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

- **返回结果：**

```
{
"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******"
}
}
```

步骤2：下载公钥

- **请求：**

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

- **返回结果：**

```
{
"Response": {
"RequestId": "408fa858-cd6d-4011-b8a0-653805******",
"KeyId": "22d79428-61d9-11ea-a3c8-525400******",
```

```
"PublicKey":
"MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJahujq+PvM*******
*******bBs/f3axWbvgvHx8Jmqw==",
"PublicKeyPem": "-----BEGIN PUBLIC KEY-----
\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJa\nhujq+PvM*****
**********bBs/f3axWbvgvHx8Jmqw==\n-----END PUBLIC KEY-----\n"
}
}
```

- **将公钥 PublicKeyPem 转成 pem 格式，并存入文件 public_key.pem。**

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJa
hujq+PvM**************bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

> **注意**
> 您可以登录 KMS 控制台，单击**用户密钥 > 密钥 ID/密钥名称**进入密钥信息页面，直接下载非对称密钥公钥。

步骤3：创建信息的明文文件

创建测试明文文件。

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

> **注意**
> 当生成的文件内容中，存在不可见的字符情况下（如换行符等），需对文件进行 truncate 操作（如 truncate -s -1 test_verify.txt），从而保证签名准确。

步骤4：计算消息摘要

> **注意**
> - 如果待签名的消息长度不超过4096字节，可以跳过本步骤，直接进入 步骤5 。
> - 如果待签名的消息的长度超过4096字节，则需先在用户端本地计算消息摘要。

使用 openssl 对 test_verity.txt 文件内容进行摘要计算。

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

步骤5：通过 KMS 签名接口生成签名

调用 KMS 的 签名 接口计算信息的签名。

1. 消息原文或消息摘要进行计算签名之前，需先进行 base64 编码。

```
//消息摘要进行base64编码
openssl enc -e -base64 -A -in digest.bin -out encoded.base64
//消息原文进行base64编码
openssl enc -e -base64 -A -in test_verify.txt -out encoded.base64
```

2. 进行签名的计算。
   - **请求：**
     - **RSA_PSS_SHA_256**

```
// 将上述 encoded.base64 的文件内容作为 SignByAsymmetricKey 的 Message 参数，以消息摘要的形式进行签名。
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm RSA_PSS_SHA_256 --Message "qJQj83hSyOuU7Tn0SRReGCk4yuuVWaeZ44BP******==" --MessageType DIGEST
// 以消息原文的形式进行签名（原文要进行Base64编码）
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm RSA_PSS_SHA_256 --Message "dG***Ao=" --MessageType RAW
```

     - **RSA_PKCS1_SHA_256**

```
// 将上述 encoded.base64 的文件内容作为 SignByAsymmetricKey 的 Message 参数，以消息摘要的形式进行签名。
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm RSA_PKCS1_SHA_256 --Message "qJQj83hSyOuU7Tn0SRReGCk4yuuVWaeZ44BP******==" --MessageType DIGEST
// 以消息原文的形式进行签名（原文要进行Base64编码）
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm RSA_PKCS1_SHA_256 --Message "dG***Ao=" --MessageType RAW
```

   - **返回结果：**

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

   - **将签名内容 Signature 存入文件 signContent.sign：**

```
echo "U7Tn0SRReGCk4yuuVWaeZ4******" | base64 -d > signContent.bin
```

步骤6：验证签名

1. 通过 KMS 验证签名接口校验。（建议使用该方法进行验证签名）
   - **请求：**
     - **RSA_PSS_SHA_256**

```
// 对消息摘要进行验证（将步骤4 encoded.base64 文件内容作为 VerifyByAsymmetricKey 的 Message 参数，以消息摘要的形式进行验签）。
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ**********=" --Algorithm RSA_PSS_SHA_256 --MessageType DIGEST
// 对消息原文进行验证（原文要进行Base64编码）。
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "dG***Ao=" --Algorithm RSA_PSS_SHA_256 --MessageType RAW
```

     - **RSA_PKCS1_SHA_256**

```
// 对消息摘要进行验证（将步骤4 encoded.base64 文件内容作为 VerifyByAsymmetricKey 的 Message 参数，以消息摘要的形式进行验签）。
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ**********=" --Algorithm RSA_PKCS1_SHA_256 --MessageType DIGEST
// 对消息原文进行验证（原文要进行 Base64 编码）。
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "dG***Ao=" --Algorithm RSA_PKCS1_SHA_256 --MessageType RAW
```

   - **返回结果：**

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

> **说明**
> 签名接口和验签接口中使用的参数 Message 和 MessageType 的取值要保持一致。

2. 通过 KMS 公钥和签名内容在本地进行验证。
   - **请求：**

```
//采用 RSA_PSS_SHA_256 算法进行签名的验签。
```

```
openssl dgst -verify public_key.pem -sha256 -sigopt rsa_padding_mode:pss -
sigopt rsa_pss_saltlen:-1 -signature ./signContent.bin ./test_verify.txt
//采用 RSA_PKCS1_SHA_256 算法进行签名的验签。
openssl dgst -verify public_key.pem -sha256 -signature ./signContent.bin
./test_verify.txt
```

- **返回结果：**

```
Verified OK
```

ECC 签名验签

最近更新时间：2024-09-10 17:42:32

本文将为您介绍如何使用 ECC 签名验签算法。

操作步骤

步骤1：创建非对称签名密钥

> **注意**
> 在密钥管理系统（KMS）中调用 创建主密钥 接口创建用户主密钥时，必须传入正确的密钥用途 ASYMMETRIC_SIGN_VERIFY_ECC，这样才可以使用签名功能。

- **请求：**

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

- **返回结果：**

```
{
"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******"
}
}
```

步骤2：下载公钥

- **请求：**

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

- **返回结果：**

```
{
"Response": {
"RequestId": "408fa858-cd6d-4011-b8a0-653805******",
"KeyId": "22d79428-61d9-11ea-a3c8-525400******",
```

```
"PublicKey":
"MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJahujq+PvM*******
*******bBs/f3axWbvgvHx8Jmqw==",
"PublicKeyPem": "-----BEGIN PUBLIC KEY-----
\nMFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJa\nhujq+PvM*****
**********bBs/f3axWbvgvHx8Jmqw==\n-----END PUBLIC KEY-----\n"
}
}
```

- **将公钥 PublicKeyPem 转成 pem 格式，并存入文件 public_key.pem：**

```
echo "-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEFLlge0vtct949CwtadHODzisgXJa
hujq+PvM**************bBs/f3axWbvgvHx8Jmqw==
-----END PUBLIC KEY-----" > public_key.pem
```

> **注意**
> 您也可以登录 KMS 控制台，单击**用户密钥 > 密钥 ID/密钥名称**进入密钥信息页面，直接下载非对称密钥公钥。

步骤3：创建信息的明文文件

创建测试明文文件。

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

> **注意**
> 当生成的文件内容中，存在不可见的字符情况下（如换行符等），需对文件进行 truncate 操作（如 truncate -s -1 test_verify.txt），从而保证签名准确。

步骤4：计算消息摘要

> **注意**
> - 如果待签名的消息的长度不超过4096字节，可以跳过本步骤，直接进入 步骤5 。
> - 如果待签名的消息的长度超过4096字节，则需先在用户端本地计算消息摘要。

使用 openssl 对 test_verity.txt 文件内容进行摘要计算。

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

步骤5：通过 KMS 签名接口生成签名

调用 KMS 的 签名 接口计算信息的签名。

1. 消息原文或消息摘要进行计算签名之前，需先进行 base64 编码。

```
//消息摘要进行base64编码。
openssl enc -e -base64 -A -in digest.bin -out encoded.base64
//消息原文进行base64编码。
openssl enc -e -base64 -A -in test_verify.txt -out encoded.base64
```

2. 进行签名的计算。
   - 请求：

```
// 将上述 encoded.base64 的文件内容作为 SignByAsymmetricKey 的 Message 参数，以消息摘要的形式进行签名。
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm ECC_P256_R1 --Message "qJQj83hSyOuU7Tn0SRReGCk4yuuVWaeZ44BP******==" --MessageType DIGEST
// 以消息原文的形式进行签名（原文要进行 Base64 编码）。
tccli kms SignByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --Algorithm ECC_P256_R1 --Message "dG***Ao=" --MessageType RAW
```

   - 返回结果：

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

   - 将签名内容 Signature 存入文件 signContent.sign：

```
echo "U7Tn0SRReGCk4yuuVWaeZ4******" | base64 -d > signContent.bin
```

步骤6：验证签名

1. 通过 KMS 验证签名接口校验。（建议使用该方法进行验证签名）
   - 请求：

```
// 对消息摘要进行验证（将步骤4 encoded.base64 文件内容作为 VerifyByAsymmetricKey 的 Message 参数，以消息摘要的形式进行验签）。
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "QUuAcNFr1Jl5+3GDbCxU7te7Uekq+oTxZ**********=" --Algorithm ECC_P256_R1 --MessageType DIGEST
// 对消息原文进行验证（原文要进行Base64编码）。
tccli kms VerifyByAsymmetricKey --KeyId 22d79428-61d9-11ea-a3c8-525400****** --SignatureValue "U7Tn0SRReGCk4yuuVWaeZ4******" --Message "dG***Ao=" --Algorithm ECC_P256_R1 --MessageType RAW
```

- **返回结果：**

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

> **注意**
> 签名接口和验签接口中使用的参数 Message 和 MessageType 的取值要保持一致。

2. 通过 KMS 公钥和签名内容在本地进行验证。
   - **请求：**

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

   - **返回结果：**

```
Verified OK
```

KMS 后量子密码实践

最近更新时间：2025-02-24 16:45:53

概述

随着量子计算机的快速发展，传统密码学面临严峻挑战：基于质数分解（RSA），离散对数（DH），椭圆曲线（ECC）的公钥密码方案，都可被量子计算机使用 Shor 算法破解。面对量子威胁，后量子密码（PQC）被设计用于抵抗量子计算机的破解，密钥管理系统（KMS）支持以下两种 PQC 密码算法：

- 基于 Kyber 的 PQC 加密解密算法，保护数据的机密性。
- 基于 Dilithium 的 PQC 签名验签算法，确保数据的完整性。

数据加密算法

Kyber 算法基于 Module Learning-With-Error（MLWE）难题，提供一个基础的 IND-CPA 安全的公钥加密方案（PKE）。通过 Fujisaki-Okamoto（FO） transform 可以获得一个 IND-CCA2安全的密钥封装方案（KEM）。KMS 将 Kyber-KEM 集成 AES-256实现数据封装方案（KEM-DEM），为用户提供 IND-CCA2安全的高效加密方案。

操作步骤

1. 登录 密钥管理系统（合规） 控制台。
2. 参考文档 创建密钥，密钥用途选择非对称加解密，加密算法选择 Kyber_AES。
3. 参考文档 后量子密码算法加密 和 后量子密码算法解密，使用腾讯云 SDK 调用相关 API 执行加解密操作。

数据签名算法

Dilithium 算法的安全性基于在格中寻找最短向量的 NP 问题，算法设计上兼顾了公钥与签名的大小，NIST Level 3可保证很高的安全强度。Dilithium 支持 DET 和 Random 的签名，使用场景也较灵活，可以通过 KMS 的 SDK 来调用相关的签名验签算法。

操作步骤

1. 登录 密钥管理系统（合规） 控制台。
2. 参考文档 创建密钥，密钥用途选择非对称签名验签，加密算法选择 Dilithium。
3. 参考文档 后量子密码算法签名 和 后量子密码算法验签，使用腾讯云 SDK 调用相关 API 执行签名验签操作。

外部密钥导入
概述

最近更新时间：2024-09-10 17:42:32

用户主密钥（CMK，Customer Master Key）是 KMS 服务的基本元素，它包含密钥 ID、密钥元数据（别名、描述、状态等）和用于加解密数据的密钥材料。
默认情况下，通过 KMS 服务创建 CMK 密钥时，由 KMS 服务底层加密机生成安全的密钥材料。当您希望使用自己的密钥材料时，也就是希望实施 BYOK（Bring Your Own Key）方案时，可通过 KMS 服务生成一个密钥材料为空的 CMK，并将自己的密钥材料导入到该用户主密钥中，形成一个外部密钥 CMK（EXTERNAL CMK），再由 KMS 服务进行该外部密钥的分发管理。

CMK 结构示意图

功能特点

- 在腾讯云上实施 BYOK（Bring Your Own Key）方案，即允许您在腾讯云架构上使用您自有的密钥材料进行敏感数据加解密服务。
- 完全掌控并管理您在腾讯云上使用的密钥服务，包括按需导入或删除密钥材料。
- 您可以在本地密钥管理基础设施中备份一份密钥材料，作为腾讯云密钥管理系统的额外灾备措施。
- 通过支持在云上使用您自有的密钥材料进行加解密操作，满足相关行业合规要求。

注意事项

- 需要确保导入密钥材料的安全性：
  - 在使用密钥导入功能时，您需要确保自己生成密钥材料的随机源的安全可靠性。目前 KMS 国密版本仅允许导入128位对称密钥，FIPS 版本仅允许导入256位对称密钥。
- 需要确保导入密钥材料的可用性：
  - KMS 服务提供自身服务的高可用及备份恢复能力，但导入的密钥材料的可用性需由用户来管理。强烈建议您采用安全可靠的方式保存密钥材料的原始备份，以便在意外删除密钥材料或密钥材料过期时，能及时将备份的密钥材料重新导入KMS服务。
- 需注意密钥导入操作的规范性：
  - 当您将密钥材料导入CMK 时，该 CMK 与该密钥材料永久关联，即不能将其他密钥材料导入该外部密钥 CMK 中。当使用该外部密钥 CMK 加密数据时，加密后的数据必须使用加密时采用的 CMK（即 CMK 的元数据及密钥材料与导入的密钥匹配）才能解密数据，否则解密将失败。请谨慎处理密钥材料、CMK 的删除操作。
- 需要注意密钥导入的状态：
  待导入状态的密钥属于启用状态的密钥，该启用状态密钥需付费使用。

操作指南

最近更新时间：2024-09-10 17:42:32

操作流程

创建外部密钥 CMK，您可以通过以下4个步骤完成操作。

1. 通过控制台或 API 创建一个密钥来源为“外部”的用户主密钥 CMK，即创建外部密钥 CMK。
2. 通过 API 操作获取密钥材料导入的参数，包括一个用于加密密钥材料的公钥，以及一个导入令牌。
3. 在本地通过加密机或其他安全的加密措施，利用步骤2获取的加密公钥对您的密钥材料进行加密。
4. 通过 API 操作，将加密后的密钥材料及步骤2获取的导入令牌，导入创建的外部密钥 CMK 中，至此导入外部密钥完成。

操作步骤

步骤1：创建外部密钥 CMK

创建外部密钥 CMK 有两种方式，控制台方式和调用 API 的方式。

- **控制台方式**

  （1）登录 密钥管理系统（合规） 控制台。

  （2）选择需要创建密钥的区域，单击**新建**开始创建密钥。

  （3）在新建密钥窗口，输入密钥名称，选择密钥材料来源为外部，阅读导入外部密钥材料的方法及注意事项并勾选确认框。

  （4）单击**确定**，即可创建外部密钥 CMK。您可在控制台看到已创建的外部密钥 CMK，“密钥来源”显示为“外部”。

- **调用 API 方式**

  本文示例使用腾讯云 命令行工具 TCCLI ，后续您可以使用任何受支持的编程语言调用。

  请求 CreateKey API 时指定参数 Type为2，执行命令如下。

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

CreateKey 函数源码示例：

```
def create_external_key(client, alias):
    """
    生成 BYOK 密钥，
    :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
```

步骤2：获取导入密钥材料参数

为确保密钥材料的安全性，需要先对您的密钥材料进行加密后再导入。您可以通过 API 获取导入密钥材料的参数，其中包括一个用于加密密钥材料的公钥，以及一个导入令牌。
通过 TCCLI 执行命令如下：

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

GetParametersForImport 函数源码示例：

```
def get_parameters_for_import(client, keyid):
    """
    获取导入主密钥（CMK）材料的参数，
    返回的Token作为执行ImportKeyMaterial的参数之一，
    返回的PublicKey用于对自主导入密钥材料进行加密。
    返回的Token和PublicKey 24小时后失效，失效后如需重新导入，需要再次调用该接口获取新的 Token
和 PublicKey。
    WrappingAlgorithm 指定加密密钥材料的算法，目前支持 RSAES_PKCS1_V1_5、
RSAES_OAEP_SHA_1、RSAES_OAEP_SHA_256。
    WrappingKeySpec 指定加密密钥材料的类型，目前只支持 RSA_2048。
    """
    try:
        req = models.GetParametersForImportRequest()
        req.KeyId = keyid
        req.WrappingAlgorithm = 'RSAES_PKCS1_V1_5' # RSAES_PKCS1_V1_5 |
RSAES_OAEP_SHA_1 | RSAES_OAEP_SHA_256
        req.WrappingKeySpec =  'RSA_2048' # RSA_2048
```

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

步骤3：本地加密您的密钥材料

在本地利用 步骤2 获取的加密公钥对您的密钥材料进行加密。加密公钥是一个2048比特的 RSA 公钥，使用的加密算法需要与获取导入密钥材料参数时指定的一致。由于 API 返回的加密公钥经过 base64 编码，因此在使用时需要先进行 base64 解码。目前 KMS 支持的加密算法有 RSAES_OAEP_SHA_1、RSAES_OAEP_SHA_256 与 RSAES_PKCS1_V1_5。
以下为通过 openssl 加密密钥材料的测试示例。在实际使用中，建议您通过加密机或其他更为安全的加密措施对密钥材料进行加密。

（1）调用 GetParametersForImport 接口获取Token 和 PublicKey。将 PublicKey 写入文件 public_key.base64。

（2）使用 openssl 生成随机数。

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

您也可以通过 GenerateRandom 生成随机数进行 base64 解码。

> **注意**
> 国密版本 key material 长度必须为 128， FIPS 版本为 256。

（3）Decode Public Key 获取公钥原文。

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

（4）使用公钥对 key material 进行加密。

```
# RSAES_OAEP_SHA_1 对应的命令行如下
openssl pkeyutl -in raw_material.bin -out encrypted_key_material.bin -inkey
public_key.bin -keyform DER -pubin -encrypt -pkeyopt rsa_padding_mode:oaep -
pkeyopt rsa_oaep_md:sha1

# RSAES_PKCS1_V1_5 对应的命令行如下
openssl pkeyutl -in raw_material.bin -out encrypted_key_material.bin -inkey
public_key.bin -keyform DER -pubin -encrypt -pkeyopt rsa_padding_mode:pkcs1

# RSAES_OAEP_SHA_256 对应的命令行如下
openssl pkeyutl -in raw_material.bin -out encrypted_key_material.bin -inkey
public_key.bin -keyform DER -pubin -encrypt -pkeyopt rsa_padding_mode:oaep -
pkeyopt rsa_oaep_md:sha256
```

（5）将密文编码后作为参数可以导入 KMS。

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

encrypted_material.base64 为最终输出，作为 EncryptedKeyMaterial 导入 KMS。

步骤4：导入密钥材料

最后，将加密后的密钥材料及 步骤2 获取的导入令牌，通过 API 操作，一起导入至 步骤1 创建的外部密钥 CMK中。

- 导入令牌与加密密钥材料的公钥具有绑定关系，同时一个令牌只能为其生成时指定的主密钥导入密钥材料。导入令牌的有效期为24小时，在有效期内可以重复使用，失效以后需要获取新的导入令牌和加密公钥。
- 如果多次调用 GetParametersForImport 获取导入材料，只有最后一次调用的 token 和 publicKey 有效，历史调用返回的将自动过期。
- 可以为从未导入过密钥材料的外部密钥导入密钥材料，也可以重新导入已经过期和已被删除的密钥材料，或者重置密钥材料的过期时间。

请求通过 ImportKeyMaterial 来导入，命令示例如下：

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

ImportKeyMaterial 函数源码示例：

```
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
```

至此，导入外部密钥操作完成。您可以像使用普通密钥一样使用外部密钥 CMK。

更多操作

删除外部密钥 CMK

外部密钥 CMK 的删除操作涉及到两类操作，一类操作为计划删除 CMK，一类操作为删除密钥材料，两类操作将会有不同的操作效果。

计划删除 CMK

通过计划删除功能，删除外部密钥 CMK。计划删除的操作强制要求有7 - 30天的等待期，当达到到期时间后，外部密钥CMK 将被彻底删除。已经删除的 CMK 将无法恢复，通过其加密的数据也将无法解密，请谨慎操作。

删除密钥材料

您可以通过两种方式删除密钥材料。当密钥材料过期或者被删除以后，外部密钥 CMK 将无法继续使用，由该 CMK 加密的密文也无法被解密，除非您重新导入相同的密钥材料。

- 通过 API 操作 DeleteImportedKeyMaterial 完成。当操作删除密钥材料后，密钥状态将变为等待导入（PendingImport）。
- 在导入密钥材料 API 操作中，将 ImportKeyMaterial 设置 ValidTo 输入参数设置过期时间完成，KMS 服务将自动删除到达过期时间的密钥材料。

> **说明**
> 等待密钥材料到期失效与手动删除密钥材料所达到的效果是一样的。

删除密钥材料，执行命令如下：

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

DeleteImportedKeyMaterial 函数源码示例：

```
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
```

> **注意**
> - 当您将密钥材料导入 CMK 时，该 CMK 与该密钥材料永久关联，即不能将其他密钥材料导入该外部密钥 CMK 中。当您删除密钥材料后，若需要重新导入密钥材料，导入的密钥材料必须与删除的密钥材料完全相同，才能导入成功。
> - 当使用外部密钥 CMK 加密数据时，加密后的数据必须使用加密时采用的 CMK（即 CMK 的元数据及密钥材料与导入的密钥匹配）才能解密数据，否则解密会失败。请谨慎处理密钥材料、CMK 的删除操作。

白盒密钥管理

概述

最近更新时间：2025-05-30 18:02:02

腾讯云密钥管理系统 KMS 提供了白盒密钥管理的解决方案，白盒密码技术是一项能够抵抗白盒攻击的密码技术，即在攻击者对加密设备终端拥有完全的控制能力，能够观测和更改程序运行时的内部数据，并进行密码运算过程的逆向分析时，有效保障加密数据和密钥的安全性。相对传统加密技术，白盒密码技术使得密钥在白盒环境中也难以被提取出来，同时白盒密钥管理也支持设备绑定认证能力，确保敏感密钥信息及加密数据的安全性保护。

白盒密钥用于保护端上的敏感根密钥信息，例如 API SecretKey，用户内部系统使用的鉴权密钥或 token，其它本地敏感根密钥信息等，实现端到端的全链路数据安全解决方案。白盒密钥管理解决方案将密钥和算法进行融合，并通过引入随机化因子，有效的将密钥隐藏起来，大大提高了密钥被嗅探、被破解的难度，从而保护了密钥这一极其敏感的信息。

功能特点

高安全性

基于高强度混淆加固算法及多重安全防护技术，保障在不可信终端下密码运算的密钥 Key 安全性保护。

动态白盒

原始密钥经过同样的白盒密码技术转化为白盒密钥，支持在白盒库不变的情况下实现灵活的密钥的动态轮换。

算法支持

支持国际算法 AES，以及国产密码算法 SM4，满足不同场景下密码算法合规要求。

多平台支持

适用于 Windows，Linux 等平台系统。

支持设备指纹绑定

绑定设备指纹信息，实现对解密密钥的加固保护，密钥只有在指定设备上才能生效，否则无法进行解密操作。

系统要求

目前白盒密钥管理服务支持以下操作系统。使用非以下列举的操作系统，服务将无法正常运行。

- 支持的 Linux 版本：CentOS6、CentOS7、 Ubuntu。注意：glibc 版本需大于等于2.14。
- 支持的 Windows 版本：Windows 7、Windows 10、Windows Server 2012、Windows Server 2016。

如何使用

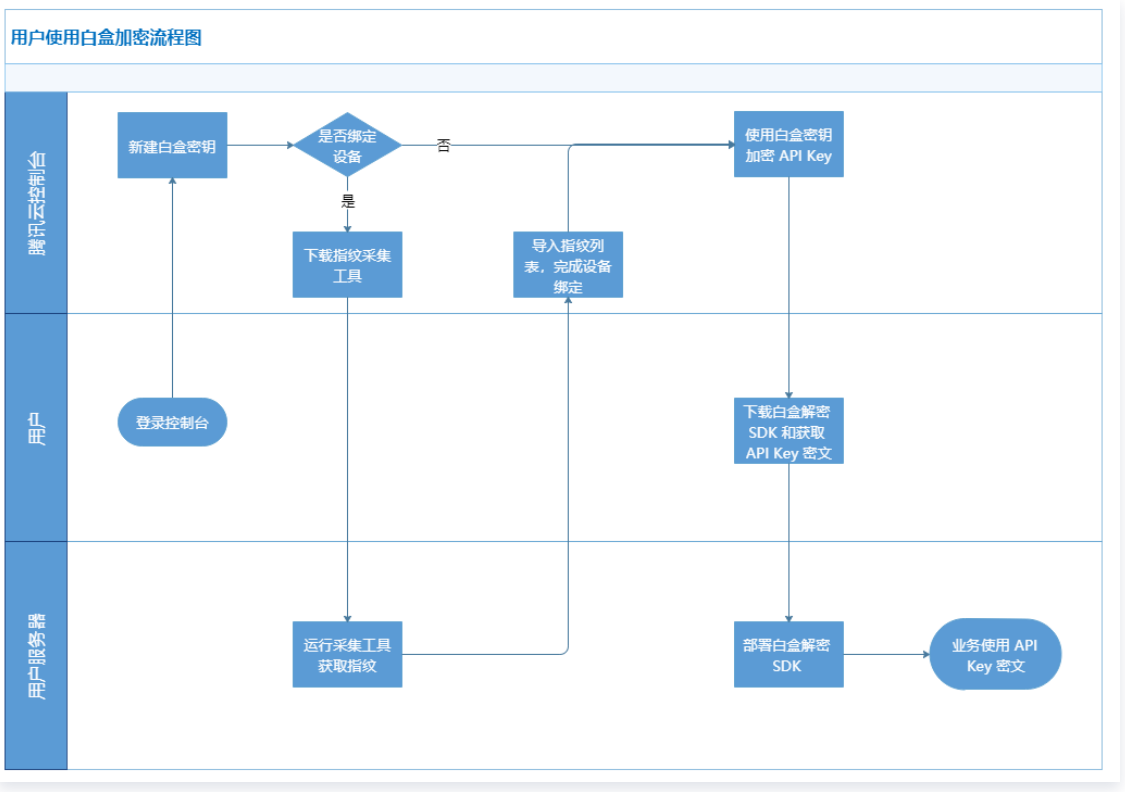
登录 密钥管理系统（合规）控制台 白盒密钥管理页面，单击**立即开通**进入白盒密钥购买页面，支付开通后即可使用。详细操作指导请参见白盒密钥 操作指南 。

操作指南

最近更新时间：2025-05-30 18:02:02

操作流程

API Key 是常用的一类用于客户端应用与后台服务器建立可信连接和数据通信的密钥。下面就以 API Key 为例，介绍腾讯云提供的白盒密钥保护解决方案。
用户使用白盒密钥加密的流程如下图所示：



从上述流程可以看出，API Key 并没有暴露在开放环境中，在实际的用户业务的服务器上只保存了 API Key 的密文，以及需要的白盒解密密钥，白盒解密算法和密钥混淆后无需额外的明文密钥进行解密，这就确保了整个链路的安全性。针对 GitHub 上源代码的 API Key 泄露事件，如果采用白盒密钥管理的解决方案，就可以彻底杜绝 API Key 的明文暴露在源代码中，从而防止 API Key 被恶意利用。

操作步骤

步骤1：创建白盒密钥

> **注意：**
> 白盒密钥为收费项，详情请参见 计费概述 和 购买方式 。

创建白盒密钥对是通过调用白盒服务来实现的，支持控制台方式和 API 方式。
本文示例使用腾讯云 命令行工具 TCCLI ，后续您可以使用任何受支持的编程语言调用。 请求命令：

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

```
'this is test for gz key' --Algorithm 'SM4'
```

返回结果示例：


```
{
    "Response": {
        "RequestId": "55f7ca05-17af-4bfb-87a5-80a80a4b0761",
        "EncryptKey":
"BA4AADZhzTBr7vmCDyHwQRCKTSCXm8pnGs38mJfA1Pw+VaP8/MW+lSCTYIi/0AUsido39JWw2XkvrnmauX
sU3cemHQ+bcpofelLB2nS2ShCINi9MTSS/KgAKhvQPD5jXtbbmUSiBcKLTMXxH6MZXcPyGvEmqe3Srhvqfl
Qk4bL/uUxAW1IC4p7E07pn6RQr2zMZAdkK0IciuwV23+cp25x2G/fEGtZi46YZR48tb2ZciGQ5kzS8se/CJ
M+lTqdBwAHV6yvPVQaXS16NFp4UhAtGCRZVfMQCp56hkhEryGjtxvIz4qfgCZ4bEo3Xp2DnNtzwBmtRBnCX
l2PbLABRfGZ9ZX7uN3IMv3rJ1mRJWJh2eBHR1N+ujO7bGTqYHIb9VxxafQ/D9PHhGO3LeQRrNi1LS31XClv
VNqNR8p/gc4GQCs987q0vdXC2PfinEV0fkyAKzis6096Jo7gfK+QvjYgVNRPr0nVuKKMI60zDKHG5jnN5kR
KuTYfhl2tppFQO2WpvCLPHQH3DcX2FZGAgmvV1yjMSp79dOgS3tnbHn8fPNIEYdziQgYV0+ce4r****",
        "DecryptKey":
"CA4AAEFLh2SUfMm1UIuxgTiD7g/rBKlxShZ/5jxRByvOqgtNyx42vDGACEtPYZwOXIhCn5wnpnisA+ZlDd
3oYnmbqO9cG9KzczsHy3x6ymlpsIfw46PB2ostz755947ykzbi71G0osTvqo9rJFrVjQpcfu7/P8iKbJhI5
Y/W2CIMuhJdbrQhXl6xyYILnhAV+o4D+c3MgzpkzB/zovcWr7EoDCDcLn4730e96ubN/+BbA9opz11jMymC
Jsx6PHsrfd6jzis4ZzDV+Uq+jYd5IJHGhmXaaQoeX2Shi+b9lOHuLi/MFqwjoskNMZ17xhc1e7l3lHpQQMI
k3dYfnK3UI+6t2D3QoRCz8ytRCfI2uRHtKGZUPYl2586iJ+48lJ7/cWLlRwsypYUv73Oas0KnuRqGCilG2d
wpIW7P/0CAqO09hN7ti+IMbiVK4dHF8aJB24A9SAHpN1ZejZbduHACPBDfb5jyMdBegswiip9qdtRzHfP9C
U1ozVRXt/8jt0iJdib6nEuvjHuFDauVuZk1ahd9MUqhdK3zKBBHdH9uP+EyiYu8w88+N0XOEMWv****",
        "KeyId": "1c820b96-73bd-11ea-a490-5254006d0810"
    }
}
```


> **注意**
> - 目前支持的加密算法有两种：SM4，AES_256。
> - 返回的 EncryptKey，DecryptKey 分别是加密密钥和解密密钥，且都做了 base64 编码。

步骤2：绑定设备指纹（可选操作）

1. 登录需要采集设备指纹的机器，运行指纹采集工具（getDeviceFingerprint），得到长度70位的字符串，即为设备指纹。
2. 通过 密钥管理系统控制台 或 API 的方式，将收集到的设备指纹列表上传并绑定到指定的白盒密钥上。详细说明和操作步骤请参见 设备绑定指南 。

> **说明：**
> 指纹采集工具（getDeviceFingerprint）请直接从 密钥管理系统控制台 上下载。

请求命令：

```
tccli kms OverwriteWhiteboxDeviceFingerprints --region ap-guangzhou --KeyId
1c820b96-73bd-11ea-a490-5254006d****  --DeviceFingerprints
```

```
'[{"Identity":"c19c024c-2ba1-11b2-a85c-96f970f4****","Description":"10.0.0.1 密钥管理系统设备"}]'
```

返回结果示例：

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

注意：
DeviceFingerprints 会完全替换现有指定密钥的指纹。如果 DeviceFingerprints 为空，则表示删除该密钥所关联的所有的设备指纹。

步骤3：对明文进行 base64 编码

准备需要使用白盒密钥管理的明文，并对该明文进行 base64 编码。例如，假设要加密的明文是：1234567890，使用 openssl 命令生成 base64 编码后的结果为：`MTIzNDU2Nzg5MAo=`。

```
echo 1234567890 | openssl base64
```

步骤4：使用白盒密钥加密 API Key

请求命令：

```
tccli kms EncryptByWhiteBox --region ap-guangzhou --KeyId a1a9376a-7261-11ea-a490-5254006d**** --PlainText MTIzNDU2Nzg5MAo=
```

返回结果：

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

注意：
返回的字段主要有两个：

- InitializationVector：随机生成的初始化向量（简称 Iv），用于增加密文被破解的难度。
- CipherText：加密后的密文，并进行了 base64 编码。

步骤5：白盒解密密钥和 API Key 密文分发

管理员将上述步骤中生成的 DecryptKey，InitializationVector，CipherText 分发给各业务系统的开发或运维人员。其中，DecryptKey 会被部署到相应业务系统的文件中，而 InitializationVector 和 CipherText 会作为 SDK 的传参。
其中：

1. 解密密钥（DecryptKey）需要以二进制文件的形式保存，供系统启动的时候读取。操作命令如下：

   1.1 将 DecryptKey 保存到文件中。

   ```
   echo
   "CA4AAEFLh2SUfMm1UIuxgTiD7g/rBKlxShZ/5jxRByvOqgtNyx42vDGACEtPYZwOXIhCn5wnpnis
   A+ZlDd3oYnmbqO9cG9KzczsHy3x6ymlpsIfw46PB2ostz755947ykzbi71G0osTvqo9rJFrVjQpcf
   u7/P8iKbJhI5Y/W2CIMuhJdbrQhXl6xyYILnhAV+o4D+c3MgzpkzB/zovcWr7EoDCDcLn4730e96u
   bN/+BbA9opz11jMymCJsx6PHsrfd6jzis4ZzDV+Uq+jYd5IJHGhmXaaQoeX2Shi+b9lOHuLi/MFqw
   joskNMZ17xhc1e7l3lHpQQMIk3dYfnK3UI+6t2D3QoRCz8ytRCfI2uRHtKGZUPYl2586iJ+48lJ7/
   cWLlRwsypYUv73Oas0KnuRqGCilG2dwpIW7P/0CAqO09hN7ti+IMbiVK4dHF8aJB24A9SAHpN1Zej
   ZbduHACPBDfb5jyMdBegswiip9qdtRzHfP9CU1ozVRXt/8jt0iJdib6nEuvjHuFDauVuZk1ahd9MU
   qhdK3zKBBHdH9uP+EyiYu8w88+N0XOEMWv****" > decrypt_key.base64
   ```

   1.2 base64 解码获取公钥实际内容：

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

   1.3 将生成的二进制文件 decrypt_key.bin 放在和业务系统（已经集成了解密 SDK）相同的服务器上的指定的目录 `decrypt_key_bin_dir` 中。

2. 将 InitializationVector，CipherText，decrypt_key_bin_dir，decrypt_key.bin 以字符串的形式，作为 SDK 中的 whitebox_decrypt 方法的传入参数。

步骤6：部署白盒解密密钥

各业务系统根据自身的编程语言，选择下载相应编程语言的解密 SDK，将 SDK 集成到业务系统中。

步骤7：使用 API Key 密文

在业务逻辑中调用 SDK 的解密函数（whitebox_decrypt），传入参数：decrypt_key_bin_dir（解密密钥文件所在的目录），decrypt_key.bin（解密密钥文件名），InitializationVector（初始化向量），CipherText（加密后的密文），algorithmType（指定生成密钥时使用的算法类型）从而获得解密后的明文。其中 algorithmType 是生成密钥时使用的算法类型。取值为0或1。0表示 AES_256，1表示 SM4。
关于白盒密钥如何进行解密，请参考 白盒密钥解密代码示例，各语言 SDK 均有详细的代码示例。

设备绑定指南

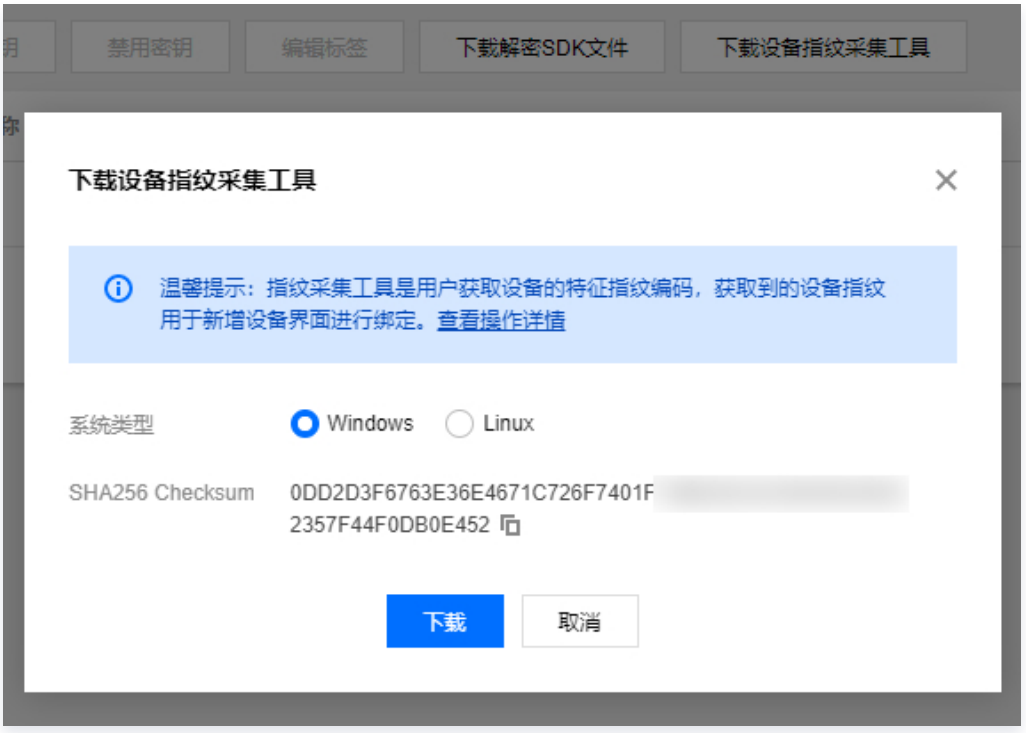
最近更新时间：2024-09-10 17:42:32

概述

设备指纹是从物理主机上提取的关于硬件设备的唯一性标识，如同人的指纹一样，不同的设备具备不同的设备指纹。
白盒密钥支持对设备指纹的集成，实现白盒密钥和指定的一个或一组设备的绑定，使得解密操作只能在已经建立绑定关系的设备上运行，从而进一步加固保护敏感信息的安全性。

操作步骤

1. 管理员进入 密钥管理系统 用户密钥管理页面。
2. 单击**下载设备指纹采集工具**，将弹出下载采集工具对话框，根据需进行绑定的设备操作系统的类型，单击**下载**，即可下载相应的设备指纹采集工具。



3. 管理员将设备指纹采集工具分发给系统运维人员，并告知需要采集指纹的设备。
4. 系统运维人员登录相应设备的操作系统，运行设备采集工具，进行设备指纹采集，如下图所示：

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

> **注意：**
> 设备指纹采集工具只支持 Windows，Linux 操作系统，并且需要在宿主操作系统上运行，目前支持物理机，腾讯云 CVM，不支持 Docker 环境。

5. 系统运维人员将采集到的设备指纹列表给到管理员。

> **说明：**
> 根据实际的应用场景和权限控制策略，系统运维人员和控制台管理员可以是同一个人。

6. 管理员在 密钥管理系统 用户密钥管理页面，选择要绑定指纹的白盒密钥。
   - 如果是新创建的白盒密钥或者未绑定过设备指纹，则单击**新增设备指纹**。
   - 如果已绑定过设备指纹，则单击**管理设备指纹**。

7. 在弹出的对话框中，输入搜集到的设备指纹，并提供每个设备指纹的描述信息。这里，有如下两种录入方式：
   - 单击**新增设备指纹**，直接在页面上录入：

   - 单击**下载批量上传导入模板**，在下载的 CSV 文件中录入，录入后，保存文件。单击**批量上传**，选择保存的 CSV 文件，将指纹信息批量上传。
     CSV 文本内容示例：

```
设备指纹,描述
123456,test description
```

8. 完成录入后，单击**确定**，即可实现白盒密钥与指定设备指纹列表的绑定。
9. 完成设备指纹绑定后，在非绑定设备上运行解密操作会出现报错，错误码为 01000016，如下图所示：

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

注意事项

1. 设备绑定是一种对白盒密钥功能的加强操作，属于可选操作。
   - 若未进行设备绑定，那么白盒解密操作可以在任意设备上执行。
   - 若进行了设备绑定，那么白盒解密操作只能在已绑设备上才能正确执行。
2. **设备指纹绑定操作必须在下载解密密钥操作之前执行**，否则设备指纹绑定不会生效。
3. 设备指纹采集工具只支持 Windows，Linux 操作系统，并且需要在宿主操作系统上运行，目前支持物理机，腾讯云CVM，不支持 Docker 环境。

使用 KMS 白盒密钥保护 SecretKey 实践教程

最近更新时间：2025-05-15 17:49:53

本文为您介绍对于 API SecretKey 进行白盒密钥加解密的操作示例，详情步骤如下：

密钥的管理和分发

步骤1：创建白盒密钥

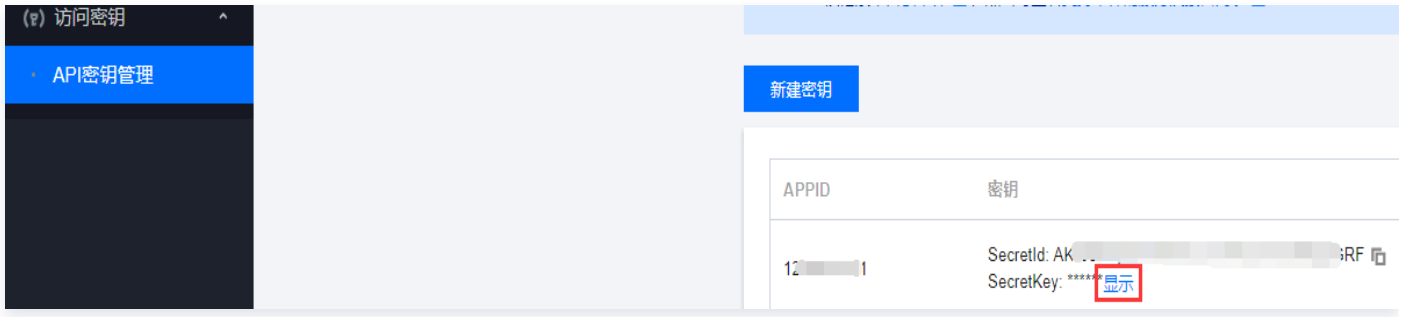
> **注意：**
> - 白盒密钥为收费项，详情请参见 计费概述 和 购买方式 。
> - 创建白盒密钥对是通过调用白盒服务来实现的，支持控制台方式和 API 方式，本文示例采用控制台方式。

1. 登录 密钥管理系统（合规）控制台，在左侧菜单栏选择**白盒密钥管理**页面，根据业务需求切换“地域”，单击**新建**。

2. 在弹出的对话框，填写白盒密钥名称，选择加密算法，描述信息及标签（两者选填），单击**确定**，即可完成白盒密钥的创建。

步骤2：控制台获取 API SecretKey

1. 使用主账号登录 API 密钥管理控制台，查看您的 API 密钥 。
2. 在密钥操作列中，单击**显示**，完成身份验证，获取并复制 SecretKey。



步骤3：对 SecretKey 明文进行 base64 编码

将步骤2中获取的 SecretKey 内容进行 base64 编码。例如，要加密的 SecretKey 明文是：lY9Ynrabcdj05YH1234LE370HOM，使用 openssl 命令生成 base64 编码后的结果为：`bFk5WW5yYWJjZGowNVlIMTIzNExFMzcwSE9NCg==` 。

> **注意：**
> 需使用 Linux 操作系统执行以下命令（使用 Windows 操作系统会给明文自动加上特殊字符 \r\n，导致明文数据不一致）。

```
echo lY9Ynrabcdj05YH1234LE370HOM | openssl base64
```

步骤4：使用白盒密钥加密 API SecretKey

1. 登录 密钥管理系统（合规）控制台，在白盒密钥列表，单击**白盒密钥ID/名称**或**操作**列的**加密**。

2. 在弹出的对话框，将步骤3中获取的编码内容填充至明文（base64）文本框中，单击**白盒加密**。

3. 加密成功之后，会返回随机生成的初始化向量（简称 IV）和加密后的密文，单击**下载IV**和**下载密文**，即可完成内容的下载。

> **说明：**

其中初始化向量（简称 IV）和加密后的密文均已进行 base64 编码。

步骤5：下载解密密钥

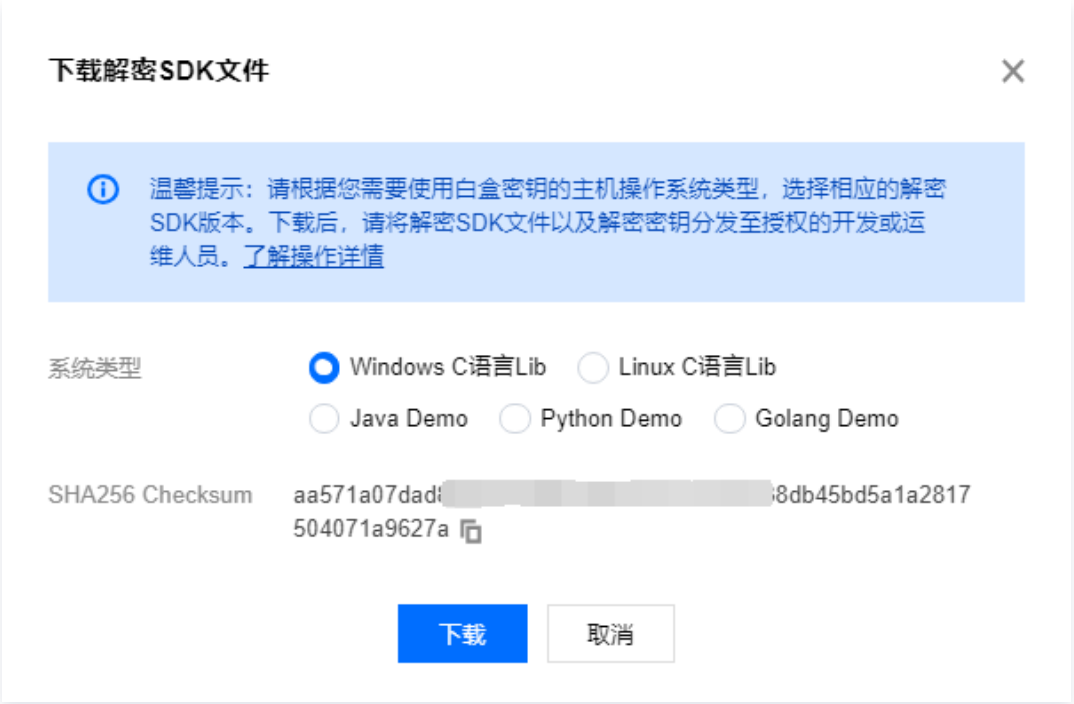
1. 登录 密钥管理系统（合规）控制台，在左侧菜单栏选择**白盒密钥管理**页面，单击**白盒密钥ID/名称**，进入密钥基本信息页面。

2. 在密钥基本信息页面，单击**下载解密密钥**，并命名为 decrypt_key_sm4.bin。

步骤6：下载解密 SDK 文件

1. 登录 密钥管理系统（合规）控制台，在左侧菜单栏选择**白盒密钥管理**页面，单击右侧的**下载解密SDK文件**。

2. 在弹出的对话框，根据各业务系统自身的编程语言，选择下载相应编程语言的解密 SDK，并将 SDK 集成到业务系统中。



步骤7：白盒解密密钥和 API SecretKey 密文分发

管理员将上述步骤中下载的解密密钥、IV 和密文三个文件，分发给各业务系统的开发或运维人员。其中，解密密钥部署到相应业务系统的文件中，而初始化向量 IV 和密文会作为 SDK 的传参。

> **注意**
> 下载的解密密钥是一个二进制 bin 文件，需要将该文件和可执行文件（已经集成了解密 SDK）放在相同的服务器上，文件路径将作为 SDK 的解密参数。
> 例如：代码示例 中指定目录为 ./data，表示放在和可执行文件相同父目录的 data 子目录中。

业务集成

使用 API SecretKey 密文

- 在业务逻辑中调用 SDK 的解密函数（whitebox_decrypt），传入参数：decrypt_key_bin_dir（步骤7中解密密钥存放的目录）、decrypt_key_sm4.bin（步骤5中下载的解密密钥，其对应的文件名）、InitializationVector（步骤4中下载的 IV）、CipherText（步骤4中用白盒加密后的 SecretKey 密文）和 algorithmType，从而获得解密后的明文。
- 其中 algorithmType 是生成密钥时使用的算法类型，取值为0或1。0表示 AES_256，1表示 SM4。
- 关于白盒密钥如何进行解密，请参考 白盒密钥解密代码示例 ，各语言 SDK 均有详细的代码示例。

白盒密钥解密代码示例

最近更新时间：2024-09-10 17:42:32

本文以 Golang、Python、C、Java 这四种语言为例，分别提供相应的示例代码。

基于 Golang 的解密代码示例

```
package main

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

func main() {
    fmt.Println("--------- test case for AES_256 ---------")
    // 解密密钥文件所在的目录
    decryptKeyFileDirectoryName := "./data"
    // 解密密钥文件名
    decryptKeyFileName := "decrypt_key_aes256.bin"
    // 初始化向量，base64 编码
    iv := "EUi3Vv7DiCf73D6XbVzMYg=="
    // 白盒密钥加密后的密文，并 base64 编码
    cipherText := "HKyXV1Xoodi1P/sdf/cYLw=="
    // 创建白盒密钥时用的加密算法，0: AES_256, 1: SM4
    algoType := 0
    fmt.Println(fmt.Sprintf("demo start for decryptKeyFileName=%s, iv=%s,
cipherText=%s, algoType=%d", decryptKeyFileName, iv, cipherText, algoType))

    whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv,
cipherText, algoType)

    fmt.Println("--------- test case for SM4 ---------")
    // 解密密钥文件所在的目录
    decryptKeyFileDirectoryName = "./data"
    // 解密密钥文件名
    decryptKeyFileName = "decrypt_key_sm4.bin"
    // 初始化向量，base64 编码
    iv = "9+COkyNOrT8mvWN6CgTjKw=="
    // 白盒密钥加密后的密文，并 base64 编码
    cipherText = "83ji4vKFwtVSAN1LSh1aOQ=="
    // 创建白盒密钥时用的加密算法，0: AES_256, 1: SM4
    algoType = 1
    fmt.Println(fmt.Sprintf("demo start for decryptKeyFileName=%s, iv=%s,
cipherText=%s, algoType=%d", decryptKeyFileName, iv, cipherText, algoType))
```

```
    whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv,
cipherText, algoType)

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

基于 Python 的解密代码示例

```
#!/usr/bin/python

import os
from ctypes import *
import base64

if __name__ == "__main__":
    print("--------- test case for AES_256 ---------")
    # 解密密钥文件所在的目录
    decryptKeyFileDirectoryName = "../data";
    # 解密密钥文件名
    decryptKeyFileName = "decrypt_key_aes256.bin"
    # 初始化向量，base64 编码
    iv = "EUi3Vv7DiCf73D6XbVzMYg=="
    # 白盒密钥加密后的密文，并 base64 编码
    cipherText = "HKyXV1Xoodi1P/sdf/cYLw=="
    # 创建白盒密钥时用的加密算法，0: AES_256, 1: SM4
    algoType = 0
    try:
        print("decryptKeyFileName=%s, iv=%s, cipherText=%s, algoType=%d" %
(decryptKeyFileName, iv, cipherText, algoType))
        plain = demo_clt_cbc_dec(decryptKeyFileDirectoryName, decryptKeyFileName,
base64.b64decode(cipherText), base64.b64decode(iv), algoType)
        print ("decrypt success")
        print ("plain: %s" % plain)
    except YdwbCryptoException as e:
        print (e.msg)

    print("--------- test case for SM4 ---------")
    # 解密密钥文件所在的目录
    decryptKeyFileDirectoryName = "../data";
    # 解密密钥文件名
    decryptKeyFileName = "decrypt_key_sm4.bin"
    # 初始化向量，base64 编码
    iv = "9+COkyNOrT8mvWN6CgTjKw=="
    # 白盒密钥加密后的密文，并 base64 编码
    cipherText = "83ji4vKFwtVSAN1LSh1aOQ=="
    # 创建白盒密钥时用的加密算法，0: AES_256, 1: SM4
    algoType = 1
    try:
```

```
        print("decryptKeyFileName=%s, iv=%s, cipherText=%s, algoType=%d" %
(decryptKeyFileName, iv, cipherText, algoType))
        plain = demo_clt_cbc_dec(decryptKeyFileDirectoryName, decryptKeyFileName,
base64.b64decode(cipherText), base64.b64decode(iv), algoType)
        print ("decrypt success")
        print ("plain: %s" % plain)
    except YdwbCryptoException as e:
        print (e.msg)

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

    pass
```

基于 C 的解密代码示例

分为 Windows 和 Linux 两个平台代码：

- Windows 平台：打开 vs 文件夹下面的 demo.sln，demo 使用的是 vs2017，有静态链接和动态链接两个 Demo，直接编译和运行即可。因为路径问题，如果在命令行中使用 exe，需要将 /data 目录拷贝到上一层目录。
- Linux 平台：需要将 lib 加入环境变量：

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

示例代码如下：

```
#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==";
//填入白盒密钥加密后的密文
        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==";  //填
入初始化向量
        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";    //填入解密密钥文件名
    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==";
//填入白盒密钥加密后的密文
        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==";  //填
入初始化向量
        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";    //填入解密密钥文件名
    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;
}
```

基于 Java 的解密代码示例

```
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 ---------");
        // 解密密钥文件所在的目录
        String decryptKeyFileDirectoryName = "../data";
        // 解密密钥文件名
        String decryptKeyFileName = "decrypt_key_aes256.bin";
        // 初始化向量，base64 编码
        String iv = "EUi3Vv7DiCf73D6XbVzMYg==";
        // 白盒密钥加密后的密文，并 base64 编码
        String cipherText = "HKyXV1Xoodi1P/sdf/cYLw==";
        // 创建白盒密钥时用的加密算法，0: AES_256, 1: SM4
        int algoType = 0;
        System.out.printf("demo start for decryptKeyFileName=%s, iv=%s,
cipherText=%s, algoType=%d \n", decryptKeyFileName, iv, cipherText, algoType);
        whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv,
cipherText, algoType);

        System.out.println("--------- test case for SM4 ---------");
        // 解密密钥文件所在的目录
        decryptKeyFileDirectoryName = "../data";
        // 解密密钥文件名
        decryptKeyFileName = "decrypt_key_sm4.bin";
        // 初始化向量，base64 编码
        iv = "9+COkyNOrT8mvWN6CgTjKw==";
        // 白盒密钥加密后的密文，并 base64 编码
        cipherText = "83ji4vKFwtVSAN1LSh1aOQ==";
        // 创建白盒密钥时用的加密算法，0: AES_256, 1: SM4
        algoType = 1;
        System.out.printf("demo start for decryptKeyFileName=%s, iv=%s,
cipherText=%s, algoType=%d \n", decryptKeyFileName, iv, cipherText, algoType);

        whitebox_decrypt(decryptKeyFileDirectoryName, decryptKeyFileName, iv,
cipherText, algoType);
```

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

    }

    public static void whitebox_decrypt(String decrypt_key_bin_dir, String
fileName, String Iv, String CipherText, int algorithmType) throws Exception {
        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 Exception {
        CipherWhiteBox instance = CipherWhiteBox.getInstance(SymAlgType.AES,
BlockMode.cbc_mode, PaddingMode.p5padding);
        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 Exception {
        CipherWhiteBox instance = CipherWhiteBox.getInstance(SymAlgType.SM4,
BlockMode.cbc_mode, PaddingMode.p5padding);
        File file = new File(keyFilePath);
        instance.init(file, iv, CryptMode.decrypt_mode);
        instance.update(data);
        return instance.doFinal();
    }

}
```

云产品集成 KMS 实现透明加密

最近更新时间：2024-10-23 09:36:11

概述

腾讯云密钥管理系统（ Key Management Service，KMS ）是一款安全、可靠、简单易用的密钥托管服务，帮助您轻松创建和管理密钥，保护密钥的安全。腾讯云 KMS 可以与大多数腾讯云上的云产品进行无缝集成，在已集成 KMS 的云产品中，仅需要选择一个从 KMS 托管的密钥，即可轻松实现对其云产品内的数据进行加解密。

云产品集成 KMS 加密为您带来如下益处：

- 云产品通过集成 KMS 实现对用户数据的加密储存，加密密钥由用户管控。KMS 底层使用国家密码管理局或 FIPS-140-2 认证的硬件安全模块 HSM 来生成和保护密钥，满足国内外合规审查标准。
- 为用户提供透明加密的解决方案，用户只需要开通已集成 KMS 的云产品加密服务，无需关心加密的细节，即可实现透明的云上数据加解密。
- 无需用户自行构建和维护密钥管理基础设施，降低了开发成本，用户使用安全便捷。

> **说明：**
> 由于其他云产品并不是密钥的托管者，在使用已集成 KMS 的云产品加密数据前，需要通过腾讯云访问管理 CAM ，完成 KMS 对云产品的角色授权操作。

支持的密钥类型

用户主密钥（Customer Master Key，CMK），是用户或云产品通过密钥管理系统创建的密钥，主要用于加密并保护数据加密密钥。一个用户主密钥可以加密多个数据密钥 DEK 。

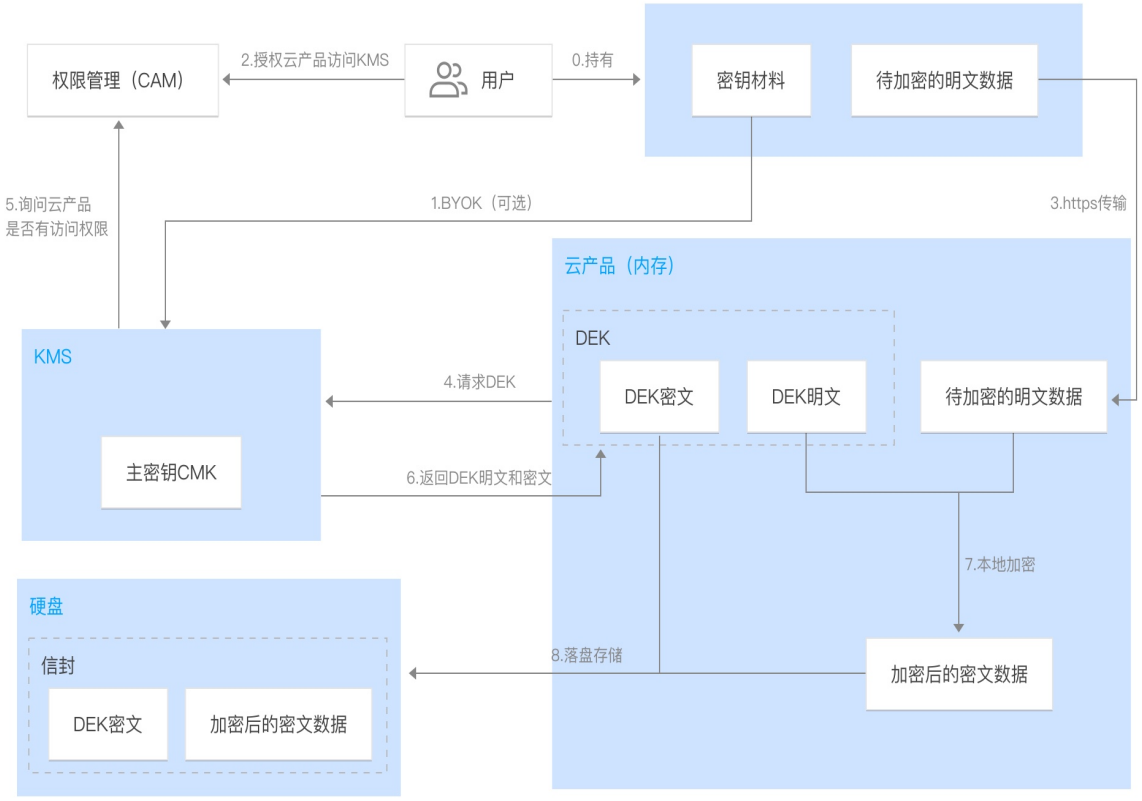
KMS 提供以下两种用户主密钥：

- 自定义密钥
  用户通过密钥管理系统自主创建的密钥。其密钥的来源方式有两种： **KMS 创建**或**自行导入** BYOK，更多信息请参见密钥管理系统 创建密钥 和 外部密钥导入 。
- 默认的云产品密钥
  在用户第一次通过对应云产品使用 KMS 加密时，云产品自动通过密钥管理系统为用户创建的密钥。云产品密钥可通过密钥管理系统控制台进行查询，但不支持禁用、计划删除操作。

加密原理

不同云产品的业务形态和客户需求，其加密的具体设计略有不同。通常情况下云产品采用 信封加密 的方式，通过调用 KMS 接口来实现对数据的加解密。

云产品使用 KMS 加密原理如下图所示：



加密流程说明如下：

1. 开通 KMS 服务并且完成 KMS 对云产品的角色授权。
2. 通过 KMS 创建用户主密钥 CMK，用户可选择默认的云产品密钥或自定义密钥。
3. 通过 CMK 调用 GenerateDataKey 接口生成数据加密密钥 DEK 密文和 DEK 明文，DEK 受 CMK 加密保护。
4. DEK 明文缓存在云产品后台的内存中，在本地对用户的数据进行加密，得到密文数据。
5. 云产品将 DEK 密文和加密后的密文数据进行落盘储存。

当前支持的云产品

- 云数据库 MySQL（TencentDB for MySQL）
- 云数据库 MariaDB（TencentDB for MariaDB）
- 云数据库 MongoDB（TencentDB for MongoDB）
- 云数据库 PostgreSQL（TencentDB for PostgreSQL）
- 对象存储（Cloud Object Storage，COS）
- 文件存储（Cloud File Storage，CFS）（需开白使用）
- 数据万象（Cloud Infinite，CI）
- 云硬盘（Cloud Block Storage，CBS）
- 凭据管理系统（Secrets Manager，SSM）
- 数据安全网关（云访问安全代理）（Data Security Gateway（Cloud Access Security Broker），CASB）

- 人脸核身

内测版 KMS 迁移指引

最近更新时间：2024-11-28 15:47:02

概述

腾讯云内测版密钥管理系统由于架构改进，计划进行 EOL（end-of-life）流程。建议用户尽快迁移到新版 KMS（产品名：密钥管理系统（合规））。
官网已正式上线全新 密钥管理系统（合规） 做服务替换。新版密钥管理系统 KMS 满足合规标准，提供了更为丰富的密钥管理功能，且大大提高了可靠性设计。

> **注意：**
> - 内测版密钥管理系统采用 API/SDK 2017 接口提供服务。
> - 内测版密钥管理系统已于2020年12月31日正式下线；内测版 KMS 不具备 SLA 服务保障，为了让您的系统服务更加安全，强烈建议您将服务升级为 商业版密钥管理系统 KMS 服务 。

价格说明

密钥管理系统 KMS 提供包年包月计费模式，详细请参见 计费概述 。

步骤说明

步骤1：内测旧 KMS 服务用户，可先在官网重新开通使用新的密钥管理系统（合规）。
步骤2：使用新版 KMS，创建用户主密钥 CMK。
步骤3：将内测旧版 KMS 服务加密的数据进行解密：按照 API/SDK 2017 接口规范，使用旧版 SDK，调用解密 Decrypt 接口，获取明文数据。
步骤4：通过新的密钥管理系统（合规）系统的 SDK 重新进行加密。

敏感数据加密迁移步骤

敏感信息加密是密钥管理系统 KMS 核心的能力，实际应用中主要用来保护服务器硬盘上敏感数据的安全（小于4KB），如密钥、证书、配置文件等，详情请参见 敏感信息加密 。

1. 开通 密钥管理系统（合规） 服务。
2. 按照业务需求，在密钥管理系统（合规）创建相应的用户主密钥 CMK。
3. 内测版密钥管理系统加密的数据进行解密：按照 API/SDK 2017 接口规范，使用旧版 SDK，调用解密 Decrypt 接口，获取明文数据，请参见 解密 API 文档。
4. 新版密钥管理系统（合规）敏感数据加密： 按照腾讯云 API 3.0 标准，使用新版 SDK，调用加密 Encrypt 接口进行加密，详情请参见 加密 API 文档。
5. 新版密钥管理系统（合规）敏感数据解密：按照腾讯云 API3.0 标准，使用新版 SDK，调用解密 Decrypt 接口进行解密，详情请参见 解密 API 文档。

信封加密迁移步骤

信封加密（Envelope Encryption）是一种应对海量数据的高性能加解密方案，详情请参见 信封加密 。

1. 开通 密钥管理系统（合规） 服务。
2. 按照业务需求，在密钥管理系统（合规）创建相应的用户主密钥 CMK。

3. 内测版密钥管理系统加密数据进行解密：只需要处理 DataKey 的迁移，按照 API/SDK 2017 接口规范，使用旧版本 SDK，调用解密 Decrypt 接口，获取明文 DataKey，详情请参见 解密 API 文档。
4. 新版密钥管理系统（合规）信封加密：按照腾讯云 API 3.0 标准，使用新版 SDK，调用加密 Encrypt 接口进行加密，详情请参见 加密 API 文档。
5. 新版密钥管理系统（合规）信封解密：按照腾讯云 API 3.0 标准，使用新版 SDK，调用解密 Decrypt 接口解密 DataKey 获取明文，使用 DataKey 明文对数据进行解密。详情请参见 解密 API 文档。