Data Encryption Workshop

Best Practices

Issue	12
Date	2025-11-11

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2025. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

Trademarks and Permissions

HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: https://www.huaweicloud.com/intl/en-us/

Contents

1 Key Management Service.. 1

1.1 Using KMS to Encrypt Offline Data.. 1

1.1.1 Symmetric Encryption and Decryption..1

1.1.1.1 Small-Size Sensitive Data Encryption and Decryption.. 1

1.1.1.2 Large-Size Envelope Encryption and Decryption...4

1.1.2 Asymmetric Encryption and Decryption.. 10

1.1.2.1 Solution Overview..10

1.1.2.2 RSA Asymmetric Data Encryption and Decryption...12

1.2 Using KMS to Encrypt and Decrypt Data for Cloud Services...13

1.2.1 Overview...14

1.2.2 Encrypting Data in ECS..16

1.2.3 Encrypting Data in EVS..16

1.2.4 Encrypting Data in IMS... 20

1.2.5 Encrypting Data in OBS..23

1.2.6 Encrypting an RDS DB Instance...30

1.2.7 Encrypting a DDS DB Instance..30

1.3 Using the Encryption SDK to Encrypt and Decrypt Local Files... 31

1.4 Encrypting and Decrypting Data Through Cross-region DR.. 34

1.5 Using KMS to Protect File Integrity.. 37

2 Cloud Secret Management Service.. 41

2.1 Using CSMS to Change Hard-coded Database Account Passwords.....................................41

2.2 Using CSMS to Prevent AK/SK Leakage.. 45

2.3 Services Using CSMS...50

2.3.1 CCE Servers Using CSMS...51

2.3.2 Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials.............................52

3 General...56

3.1 Retrying Failed DEW Requests by Using Exponential Backoff... 56

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

1 Key Management Service

1.1 Using KMS to Encrypt Offline Data

1.1.1 Symmetric Encryption and Decryption

1.1.1.1 Small-Size Sensitive Data Encryption and Decryption

KMS is mainly used to encrypt sensitive data that is smaller than 4 KB, such as keys, certificates, and configurations, into ciphertext using root keys (CMKs). The data is decrypted only in the memory and transmitted using HTTPS. Plaintext data is not transmitted out of the memory, stored in disks, or recorded in logs, ensuring sensitive data security.

To use KMS to perform high-performance encryption and decryption for massive data, see **Large-Size Envelope Encryption and Decryption**.

Sensitive Data Example

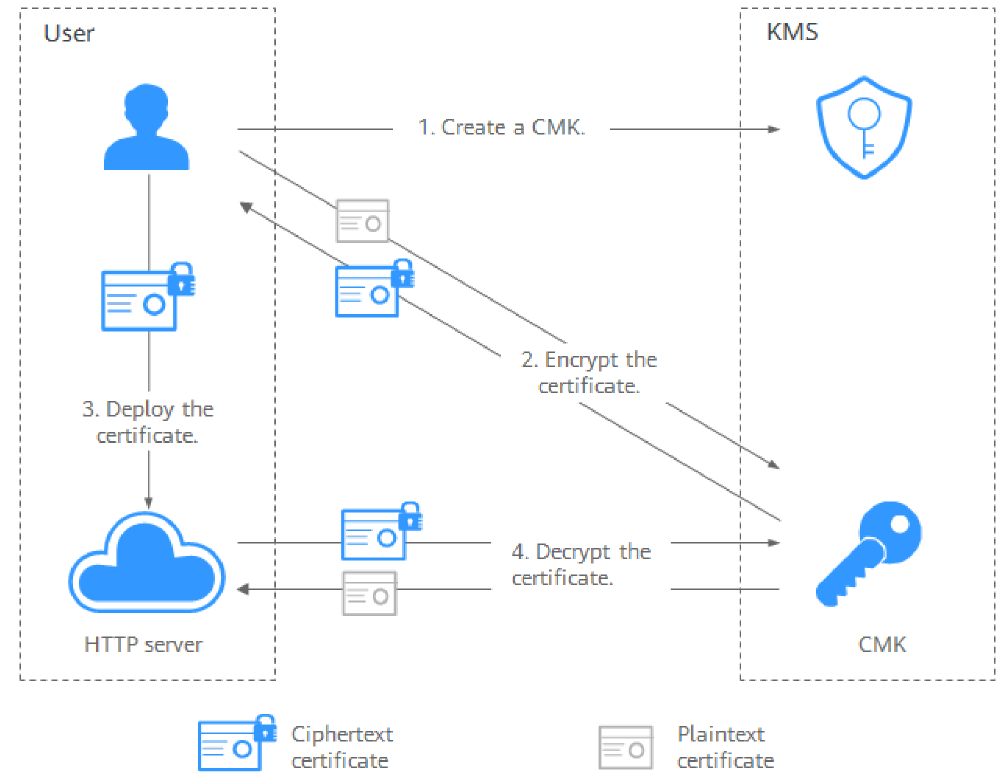
Sensitive Information	Usage	Loss Risk
Key certificate	Encrypt service data, communication channels, and digital signatures.	Confidential information will be stolen, encrypted channels will be intercepted, and signatures will be forged.
Backend configuration file	Store system architecture and other service information, such as database IP addresses and passwords.	Service data breach will occur and will be used to attack other systems.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Encryption and Decryption Principles

Figure 1-1 shows an example about how to call KMS APIs to encrypt and decrypt an HTTPS certificate.

Figure 1-1 Encrypting and decrypting an HTTPS certificate



Preparations

- You have obtained the Huawei Cloud SDK and installed it.
- You have obtained a Huawei Cloud tenant account and its AK/SK. You can create or view the AK/SK on the **My Credentials** > **Access Keys** page of the Huawei Cloud console.

Procedure

Step 1 Create a CMK with the AES_256 algorithm. For details, see **Creating a Key**.

Step 2 Encrypt and decrypt sensitive information.

1. **Console mode**

   You can use the online tool to perform one-time or non-batch encryption and decryption, for example, generating a key ciphertext for the first time. You do not need to develop additional tools for non-batch encryption and decryption. Instead, you can focus on core service capabilities. For details, see **Encrypting and Decrypting Small-size Data Online Using a Custom Key**.

2. **GO mode**

   Encrypt data that is smaller than 4 KB. It can be used to encrypt database passwords, RSA keys, and other small-size sensitive information. This

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

following uses Go as an example. You can also use other supported programming languages.

Call the KMS API for **encrypting a data key** and use the specified CMK to encrypt the plaintext certificate.

Go code example

```
package main

import (
  "log"
  "os"

  "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/auth/basic"
  "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/config"
  "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/region"
  kms "github.com/huaweicloud/huaweicloud-sdk-go-v3/services/kms/v2"
  kmsModel "github.com/huaweicloud/huaweicloud-sdk-go-v3/services/kms/v2/model"
)

const plainText = "Hello World!"

// There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt
the AK/SK in the configuration file or environment variables for storage.
// In this example, the AK and SK are stored in environment variables. Before running this example,
set environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK.
var ak = os.Getenv("HUAWEICLOUD_SDK_AK")
var sk = os.Getenv("HUAWEICLOUD_SDK_SK")
var iamEndpoint = "https://<IAM_ENDPOINT>"
var endpoint = "<ENDPOINT>"
var regionID = "<REGION_ID>"

func main() {

  // CMK ID
  keyId := "xxxxxxxx-xxx-xxx-xxxx-xxxxxxxxxxx1"

  // 1. Prepare the authentication information.
  auth := basic.NewCredentialsBuilder().
    WithAk(ak).
    WithSk(sk).
    WithIamEndpointOverride(iamEndpoint).
    Build()

  httpConfig := config.DefaultHttpConfig()
  httpConfig.WithIgnoreSSLVerification(true)

  // 2. Initialize the SDK and import the authentication information and KMS endpoint information.
  client := kms.NewKmsClient(
    kms.KmsClientBuilder().
      WithRegion(region.NewRegion(regionID, endpoint)).
      WithHttpConfig(httpConfig).
      WithCredential(auth).
      Build())

  // 3. Encrypt data.
  encryptDataRequest := kmsModel.EncryptDataRequest{
    Body: &kmsModel.EncryptDataRequestBody{KeyId: keyId, PlainText: plainText},
  }
  encryptDataResponse, err := client.EncryptData(&encryptDataRequest)
  if err != nil {
    log.Fatal("Encrypt data occur error.")
  }

  // 4. Decrypt data.
  decryptDataRequest := kmsModel.DecryptDataRequest{
    Body: &kmsModel.DecryptDataRequestBody{KeyId: &keyId, CipherText:
*encryptDataResponse.CipherText},
  }
```

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

```
    decryptDataResponse, err := client.DecryptData(&decryptDataRequest)
    if err != nil {
      log.Fatal("Decrypt data occur error.")
    }

    // 5. Compare the decryption results.
    log.Printf("result is %t", *decryptDataResponse.PlainText == plainText)
}
```

----End

1.1.1.2 Large-Size Envelope Encryption and Decryption

Envelope encryption is a high-performance encryption and decryption solution for massive data. A one-time DEK, which is randomly generated locally, is used to encrypt and decrypt massive data in seconds. The DEK is immediately encrypted by the CMK and stored on the disk. You can call the KMS API to decrypt the DEK ciphertext into plaintext and store the plaintext in the memory. The encryption and decryption performance is close to that of the local AES. The key is securely managed in the KMS.

Encryption Solutions

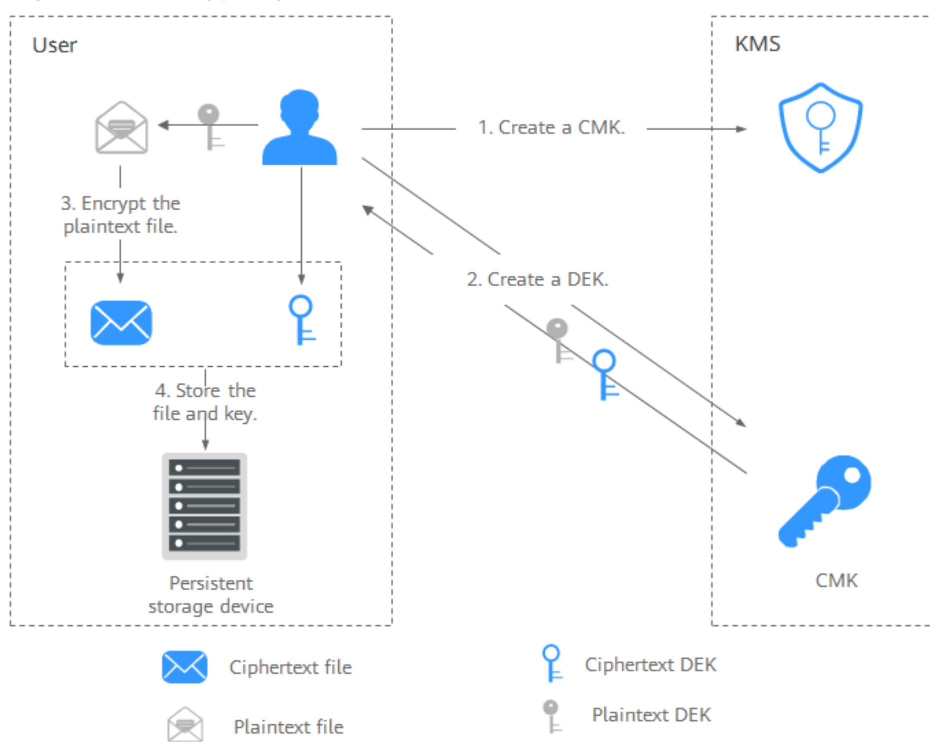
Item	Sensitive Information Encryption	Envelope Encryption
Key	CMK	CMK and DEK
Performance	Symmetric encryption, remote calling	Remote symmetric encryption for small-size data and local symmetric encryption for large-size data
Scenario	Keys, certificates, and small-size data, which are applicable to scenarios where the calling frequency is low.	Large-size data, which is applicable to scenarios where high performance is required.
Cloud API calling	Cloud APIs are called each time encryption or decryption is performed.	An API is called once after the process is started to decrypt the DEK ciphertext.

Encryption and Decryption Principles

- Large-size data encryption

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-2 Encrypting a local file



The process is as follows:

a. Create a CMK on KMS.

b. Call the **create-datakey** API of KMS to create a DEK. Then you get a plaintext DEK and a ciphertext DEK. The ciphertext DEK is generated when you use a CMK to encrypt the plaintext DEK.

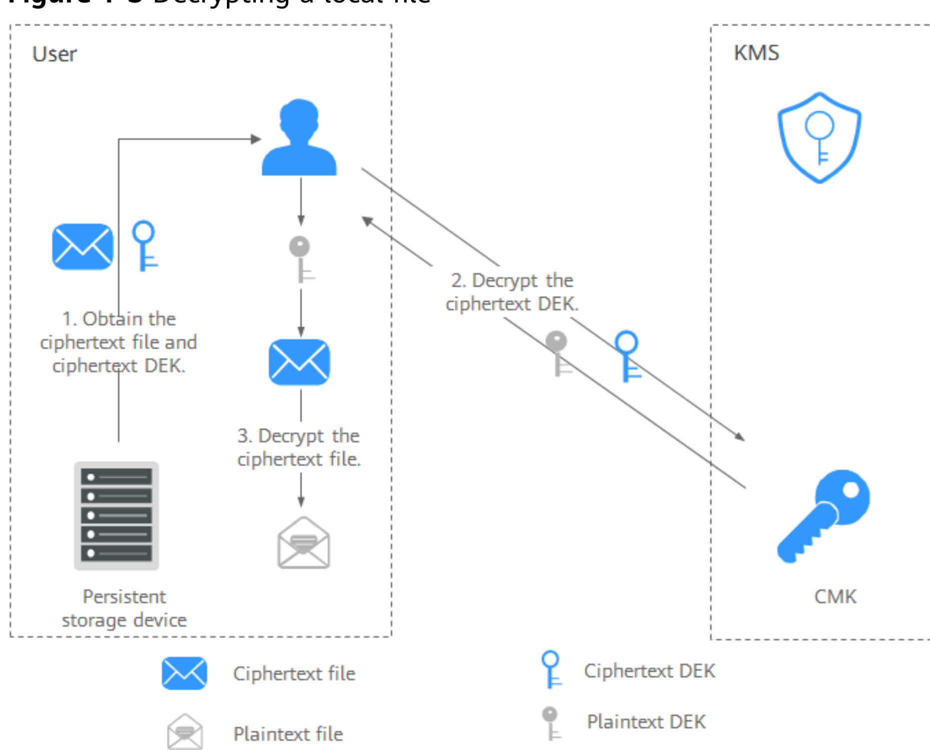
c. Use the plaintext DEK to encrypt the file. A ciphertext file is generated.

d. Save the ciphertext DEK and the ciphertext file together in a persistent storage device or a storage service.

- Large-size data decryption

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-3 Decrypting a local file



The process is as follows:

a. Obtain the ciphertext DEK and file from the persistent storage device or the storage service.

b. Call the **decrypt-datakey** API of KMS and use the corresponding CMK (the one used for encrypting the DEK) to decrypt the ciphertext DEK. Then you get the plaintext DEK.

   If the CMK is deleted, the decryption fails. Therefore, properly keep your CMKs.

c. Use the plaintext DEK to decrypt the ciphertext file.

APIs Related to Envelope Encryption

You can use the APIs listed in the following table to encrypt and decrypt data.

API	Description
Creating a DEK	Create a DEK.
Encrypting a DEK	Encrypt a DEK with the specified master key.
Decrypting a DEK	Decrypt a DEK with the specified master key.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Procedure

Step 1 Create a CMK on the management console. For details, see **Creating a Key**.

Step 2 Prepare basic authentication information.

- **ACCESS_KEY**: Access key of the Huawei account
- **SECRET_ACCESS_KEY**: Secret access key of the Huawei account
- **PROJECT_ID**: project ID of a Huawei Cloud site. For details, see **Obtaining Account, IAM User, Group, Project, Region, and Agency Information**.
- **KMS_ENDPOINT**: endpoint for accessing KMS.
- There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
- In this example, the AK/SK stored in the environment variables are used for identity authentication. Configure the environment variables **HUAWEICLOUD_SDK_AK** and **HUAWEICLOUD_SDK_SK** in the local environment first.

Step 3 Encrypt and decrypt local files.

Go code example

```
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/sha256"
    "encoding/hex"
    "io"
    "io/ioutil"
    "log"
    "math/rand"
    "os"
    "time"

    "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/auth/basic"
    "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/config"
    "github.com/huaweicloud/huaweicloud-sdk-go-v3/core/region"
    kms "github.com/huaweicloud/huaweicloud-sdk-go-v3/services/kms/v2"
    kmsModel "github.com/huaweicloud/huaweicloud-sdk-go-v3/services/kms/v2/model"
)

const (
    AadData = "Demo for aad"
)

// There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
// In this example, the AK and SK are stored in environment variables. Before running this example, set environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK.
var ak = os.Getenv("HUAWEICLOUD_SDK_AK")
var sk = os.Getenv("HUAWEICLOUD_SDK_SK")
var iamEndpoint = "https://<IAM_ENDPOINT>"
var endpoint = "<ENDPOINT>"
var regionID = "<REGION_ID>"

func main() {

    // CMK ID
    keyId := "xxxxxxxx-xxx-xxx-xxxx-xxxxxxxxxxx3"

    // 1. Prepare the authentication information.
```

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

```
    auth := basic.NewCredentialsBuilder().
        WithAk(ak).
        WithSk(sk).
        WithIamEndpointOverride(iamEndpoint).
        Build()

    httpConfig := config.DefaultHttpConfig()
    httpConfig.WithIgnoreSSLVerification(true)

    // 2. Initialize the SDK and import the authentication information and KMS device information.
    client := kms.NewKmsClient(
        kms.KmsClientBuilder().
            WithRegion(region.NewRegion(regionID, endpoint)).
            WithHttpConfig(httpConfig).
            WithCredential(auth).
            Build())

    // 3. Create a DEK.
    aesDataKeyLength := "256"
    createDataKeyRequest := kmsModel.CreateDatakeyRequest{
        Body: &kmsModel.CreateDatakeyRequestBody{KeyId: keyId, DatakeyLength: &aesDataKeyLength},
    }
    createDataKeyResponse, err := client.CreateDatakey(&createDataKeyRequest)
    if err != nil {
        log.Fatal("Create data key occur error.")
    }

    // Read the file to be encrypted and store it after GCM encryption.
    readData, err := FileReader("FirstPlainFile.jpg")
    if err != nil {
        log.Fatal("Read file occur error.")
        return
    }
    nonce := GenRandomBytes(12)
    cryptData, err := GcmCrypt(readData, *createDataKeyResponse.PlainText, nonce, 0)
    err = FileWriter(cryptData, "SecondEncryptFile.jpg")
    if err != nil {
        log.Fatal("Write encrypt file occur error.")
        return
    }

    // 4. Decrypt the DEK.
    decryptDataKeyRequest := kmsModel.DecryptDatakeyRequest{
        Body: &kmsModel.DecryptDatakeyRequestBody{KeyId: keyId, CipherText:
*createDataKeyResponse.CipherText, DatakeyCipherLength: "32"},
    }
    decryptDataKeyResponse, err := client.DecryptDatakey(&decryptDataKeyRequest)
    if err != nil {
        log.Fatal("Decrypt data key occur error.")
    }

    // 5. Read the encrypted file, and use the decrypted DEK for GCM decryption and storage.
    readData, err = FileReader("SecondEncryptFile.jpg")
    if err != nil {
        log.Fatal("Read file occur error.")
        return
    }

    decryptData, err := GcmCrypt(readData, *decryptDataKeyResponse.DataKey, nonce, 1)
    err = FileWriter(decryptData, "ThirdDecryptFile.jpg")
    if err != nil {
        log.Fatal("Write encrypt file occur error.")
        return
    }

    // 6. For example, check whether the digest of the source file is the same as that of the encrypted and
decrypted file.
    log.Printf("result is %t", FileSha256("FirstPlainFile.jpg") == FileSha256("ThirdDecryptFile.jpg"))
}
```

```
// GcmCrypt AES-GCM encryption and decryption data stream
// data: file to be encrypted or decrypted
// plainKey: plaintext key
// nonce: initialized vector
// mode: 0 indicates encryption and 1 indicates decryption.
func GcmCrypt(data []byte, plainKey string, nonce []byte, mode int) ([]byte, error) {
    plainKeyByte, err := hex.DecodeString(plainKey)
    if err != nil {
        log.Fatal("Invalid aes key.")
        return nil, err
    }
    block, err := aes.NewCipher(plainKeyByte)
    if err != nil {
        log.Fatal("Invalid init aes.")
        return nil, err
    }

    gcm, err := cipher.NewGCM(block)
    if err != nil {
        log.Fatal("Invalid init gcm.")
        return nil, err
    }

    if mode == 0 {
        return gcm.Seal(nil, nonce, data, []byte(AadData)), nil
    } else {
        return gcm.Open(nil, nonce, data, []byte(AadData))
    }
}

func FileReader(path string) ([]byte, error) {
    file, err := ioutil.ReadFile(path)
    if err != nil {
        log.Fatal("Read file failed.")
        return nil, err
    }
    return file, nil
}

func FileWriter(data []byte, path string) error {
    err := ioutil.WriteFile(path, data, 0600)
    if err != nil {
        log.Fatal("Write file failed.")
        return err
    }
    return nil
}

func GenRandomBytes(size int) (randomBytes []byte) {
    rand.Seed(time.Now().UnixNano())
    randomBytes = make([]byte, size)
    _, err := rand.Read(randomBytes)
    if err != nil {
        log.Fatal("Read data failed.")
        return
    }
    return randomBytes
}

func FileSha256(filePath string) string {
    file, err := os.Open(filePath)
    if err != nil {
        return ""
    }

    defer file.Close()

    hash := sha256.New()
```

```
    _, err = io.Copy(hash, file)
    if err != nil {
        log.Fatal("Sha256 file data failed.")
        return ""
    }
    return hex.EncodeToString(hash.Sum(nil))
}
```

----End

1.1.2 Asymmetric Encryption and Decryption

1.1.2.1 Solution Overview

For asymmetric encryption and decryption, a public key and a private key are required. In cryptography, these two keys are a pair of bidirectional keys. That is, either the public key or the private key can be used for encryption, and only the other one can be used for decryption. The public key can be set to public, while the private key must be kept securely.

Asymmetric encryption does not require reliable key distribution channels. It is usually used between systems with different trust levels to implement encrypted transmission of sensitive data or digital signature verification.

Asymmetric Key Type

Table 1-1 lists the asymmetric key algorithms supported by KMS.

Table 1-1 Asymmetric key algorithms supported by KMS

Key Type	Algorithm Type	Key Specifications	Description	Application
Asymmetric key	RSA	• RSA_2048 • RSA_3072 • RSA_4096	RSA asymmetric key	• Digital signature and signature verification • Data encryption and decryption **NOTE** Asymmetric keys are applicable to signature and signature verification scenarios. Asymmetric keys are not efficient enough for data encryption. Symmetric keys are suitable for encrypting and decrypting data.
Asymmetric key	ECC	• EC_P256 • EC_P384	Elliptic curve recommended by NIST	Digital signature and signature verification

Typical Scenarios of Asymmetric Encryption

Asymmetric encryption and decryption can be used for encrypted communication and digital signature.

Encrypted Communication

Encrypted communication is a typical application of asymmetric encryption algorithms. Encrypted communication is similar to symmetric encryption. The difference is that the public key is used for encryption and the private key is used for decryption.

The principles of encrypted communication are as follows:

1. The information receiver creates a public-private key pair and sends the public key to one or more information senders.

2. The information sender uses the public key to encrypt sensitive information and sends the encrypted ciphertext to the information receiver through the transmission medium.

3. After obtaining the data, the information receiver decrypts the information using the private key to restore the original information.

The ciphertext can be decrypted only using the private key, which is not public. Therefore, even if information is leaked due to the low security of the transmission medium, the ciphertext cannot be decrypted, ensuring the security of sensitive information.

Digital Signature

Digital signature is another typical application of asymmetric encryption algorithms. Digital signature includes signature signing and verification. A private key is used for signature, and a public key is used for verification. The process is opposite to that of encrypted communication.

The principles of digital signature are as follows:

1. The information sender creates a public-private key pair and sends the public key to one or more information receivers.

2. The information sender uses a hash function to generate a message digest from the message, and then uses the private key to encrypt the digest. The encrypted digest is the digital signature corresponding to the message.

3. The information sender sends the message and the digital signature to the information receiver.

4. After receiving the message and the digital signature, the information receiver uses the same hash function to generate digest A from the message, and uses the public key provided by the information sender to decrypt the digital signature to obtain digest B. The information receiver then compares digest A with digest B to check whether the message has been tampered with.

The signature is generated using a non-public private key, ensuring signature uniqueness. Digital signatures ensure that data is not tampered with during transmission, verify the identity of the information sender, and prevent repudiation in transactions.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

1.1.2.2 RSA Asymmetric Data Encryption and Decryption

Sensitive data needs to be encrypted during transmission, for example, key exchange. To use the asymmetric key encryption and decryption, perform the following operations:

1. Create an asymmetric encryption key using the RSA_3072 algorithm. For details, see **Creating a Key**.
2. Obtain the public key. For details, see **Querying a Public Key**.
3. The information receiver distributes the public key to the sender.
4. The information sender uses the public key to locally encrypt sensitive data and sends the encrypted ciphertext to the receiver.
5. After receiving the ciphertext, the information receiver calls the decryption function of KMS to decrypt the ciphertext. For details, see **Decrypting Data**.

During the transmission of sensitive data, the data is encrypted into ciphertext. The only key that can decrypt the ciphertext is stored in KMS and cannot be obtained by anyone, greatly improving the security of encrypted transmission of sensitive data.

Example

The following shows an example:

An RSA_3072 CMK is used for **ENCRYPT_DECRYPT**. After a public key is used to encrypt "*hello world!*" offline, **decrypt-data** is called to decrypt the message using a private key. The RSA/ECB/OAEPWithSHA-256AndMGF1Padding algorithm is used.

```
public class RsaEncryptDataExample {
    /**
     * Basic authentication information:
     * - ACCESS_KEY: Access key of the Huawei Cloud account. For details, see How Do I Obtain an Access Key (AK/SK)?
     * - SECRET_ACCESS_KEY: Secret access key of the Huawei Cloud account. This is sensitive information. Store it in ciphertext. For details, see How Do I Obtain an Access Key (AK/SK)?
     * - IAM_ENDPOINT: Endpoint for accessing IAM.
     * - KMS_REGION_ID: Regions supported by KMS.
     * - KMS_ENDPOINT: Endpoint for accessing KMS.
     * - There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
     * - In this example, the AK/SK stored in the environment variables are used for identity authentication. Configure the environment variables HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK in the local environment first.
     */
    private static final String ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_AK");
    private static final String SECRET_ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_SK");
    private static final String IAM_ENDPOINT = "https://<IamEndpoint>";
    private static final String KMS_REGION_ID = "<RegionId>";
    private static final String KMS_ENDPOINT = "https://<KmsEndpoint>";

    private static final String RSA_PUBLIC_KEY_BEGIN = "-----BEGIN PUBLIC KEY-----\n";
    private static final String RSA_PUBLIC_KEY_END = "-----END PUBLIC KEY-----";

    private static final String RSAES_OAEP_SHA_256 = "RSA/ECB/OAEPWithSHA-256AndMGF1Padding";

    private static final String SHA_256 = "SHA-256";

    private static final String MGF1 = "MGF1";

    private static final String HELLO_WORLD = "hello world!";
```

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

```
    public static void main(String[] args) throws Exception {

        final String keyId = args[0];

        publicKeyEncrypt(keyId);
    }

    private static void publicKeyEncrypt(String keyId) throws NoSuchAlgorithmException,
InvalidKeySpecException,
            NoSuchPaddingException, InvalidAlgorithmParameterException, InvalidKeyException,
            IllegalBlockSizeException, BadPaddingException {

        // 1. Prepare the authentication information for accessing Huawei Cloud.
        final BasicCredentials auth = new BasicCredentials()
                .withIamEndpoint(IAM_ENDPOINT).withAk(ACCESS_KEY).withSk(SECRET_ACCESS_KEY);

        // 2. Initialize the SDK and import the authentication information and address for KMS to access the
client.
        final KmsClient kmsClient = KmsClient.newBuilder()
                .withRegion(new Region(KMS_REGION_ID, KMS_ENDPOINT)).withHttpConfig(new HttpConfig()
                        .withIgnoreSSLVerification(true)).withCredential(auth).build();

        // 3. Obtain the public key information. The returned information is in PKCS8 format.
        final ShowPublicKeyRequest showPublicKeyRequest = new ShowPublicKeyRequest()
                .withBody(new OperateKeyRequestBody().withKeyId(keyId));
        final ShowPublicKeyResponse showPublicKeyResponse =
kmsClient.showPublicKey(showPublicKeyRequest);

        // 4. Obtain the public key string.
        final String publicKeyStr = showPublicKeyResponse.getPublicKey().replace(RSA_PUBLIC_KEY_BEGIN, "")
                .replaceAll("\n", "").replace(RSA_PUBLIC_KEY_END, "");

        // 5. Obtain the binary public key.
        final X509EncodedKeySpec keySpec = new
X509EncodedKeySpec(Base64.getDecoder().decode(publicKeyStr));
        final KeyFactory keyFactory = KeyFactory.getInstance("RSA", new BouncyCastleProvider());
        final PublicKey publicKey = keyFactory.generatePublic(keySpec);

        // 6. Encrypt the string "hello world!" offline using a public key.
        final Cipher cipher = Cipher.getInstance(RSAES_OAEP_SHA_256);
        final OAEPParameterSpec oaepParameterSpec = new OAEPParameterSpec(SHA_256, MGF1,
                new MGF1ParameterSpec(SHA_256), PSource.PSpecified.DEFAULT);
        cipher.init(Cipher.ENCRYPT_MODE, publicKey, oaepParameterSpec);
        final byte[] cipherData = cipher.doFinal(HELLO_WORLD.getBytes(StandardCharsets.UTF_8));

        // 7. Decrypt the ciphertext online using a private key.
        final DecryptDataRequest decryptDataRequest = new DecryptDataRequest()
                .withBody(new DecryptDataRequestBody().withKeyId(keyId)
                        .withEncryptionAlgorithm(DecryptDataRequestBody.EncryptionAlgorithmEnum.RSAES_OAEP
_SHA_256)
                        .withCipherText(Base64.getEncoder().encodeToString(cipherData)));

        final DecryptDataResponse decryptDataResponse = kmsClient.decryptData(decryptDataRequest);

        assert HELLO_WORLD.equals(decryptDataResponse.getPlainText());
    }

}
```

1.2 Using KMS to Encrypt and Decrypt Data for Cloud Services

1.2.1 Overview

After your cloud services are integrated with KMS, to encrypt data on cloud, you simply need to select a CMK managed by KMS for encryption.

You can select a default key automatically created by a cloud service through KMS, or a key you created or imported to KMS. For details, see **What Is a Customer Master Key?**

This section describes how to use KMS for encryption.

Table 1-2 Cloud services integrated with KMS

Category	Service	Encryption Mode
Computing	Elastic Cloud Server (ECS)	You can encrypt an image or EVS disk in ECS. • When creating an ECS, if you select an encrypted image, the system disk of the created ECS automatically has encryption enabled, with its encryption mode same as the image encryption mode. • When creating an ECS, you can encrypt added data disks.
	Image Management Service (IMS)	**Encrypting Data in IMS**
Storage	Object Storage Service (OBS)	**Encrypting Data in OBS**
	Elastic Volume Service (EVS)	**Encrypting Data in EVS**
	Volume Backup Service (VBS)	VBS generally creates online backups for a single EVS disk (system or data disk) of the server. If it is encrypted, its backup data will be stored in encrypted mode.
	Cloud Server Backup Service (CSBS)	CSBS mainly creates consistency backups online for all EVS disks of the server. CSBS backups will also be displayed on the VBS page. If it is encrypted, its backup data will be stored in encrypted mode.
Database	RDS for MySQL	**Encrypting an RDS DB Instance**
	RDS for PostgreSQL	
	RDS for SQL Server	
	Document Database Service (DDS)	**Encrypting a DDS DB Instance**

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Encryption Process

Huawei Cloud services use the envelope encryption technology and call KMS APIs to encrypt service resources. Your CMKs are under your own management. With your grant, Huawei Cloud services use a specific CMK of yours to encrypt data.

Figure 1-4 How Huawei Cloud uses KMS for encryption

The encryption process is as follows:

1. Create a CMK on KMS.
2. A Huawei Cloud service calls the **create-datakey** API of the KMS to create a DEK. A plaintext DEK and a ciphertext DEK are generated.

   NOTE

   Ciphertext DEKs are generated when you use a CMK to encrypt the plaintext DEKs.
3. The Huawei Cloud service uses the plaintext DEK to encrypt a plaintext file, generating a ciphertext file.
4. The Huawei Cloud service saves the ciphertext DEK and the ciphertext file together in a permanent storage device or a storage service.

NOTE

When users download the data from the Huawei Cloud service, the service uses the CMK specified by KMS to decrypt the ciphertext DEK, uses the decrypted DEK to decrypt data, and then provides the decrypted data for users to download.

1.2.2 Encrypting Data in ECS

Overview

KMS supports one-click encryption for ECS. The images and data disks of ECS can be encrypted.

- When creating an ECS, if you select an encrypted image, the system disk of the created ECS automatically has encryption enabled, with its encryption mode same as the image encryption mode.
- When creating an ECS, you can encrypt added data disks.

For details about how to encrypt an image, see **Encrypting Data in IMS**.

For details about how to encrypt a data disk, see **Encrypting Data in EVS**.

1.2.3 Encrypting Data in EVS

KMS encrypts created cloud disks to ensure data security.

NOTE

- The encryption attribute of a disk cannot be changed after the disk is created.
- For details about how to create an encrypted disk, see **Purchasing an EVS Disk**.
- Disk encryption is used for data disks only. System disk encryption relies on the image. For details, see **Encrypting Data in IMS**.

Scenario

You can use the key provided by KMS to encrypt data on the disk as required during EVS disk creation. You do not need to build or maintain the key management infrastructure, ensuring security and convenience.

KMS keys include default keys, custom keys, and shared keys.

- **Default key**: The key that is automatically created by EVS through KMS and named **evs/default**.

  The default key cannot be disabled and does not support scheduled deletion.
- **Custom key**: Keys created by users. You can select an existing key or create one. For details, see "Key Management Service" > "Creating a Key" in *Data Encryption Workshop (DEW) User Guide*.

  NOTE

  You will be billed for the custom keys you use. If pay-per-use keys are used, ensure that you have sufficient account balance. If yearly/monthly keys are used, renew your order timely. Or, your services may be interrupted and data may never be restored if encrypted disks become inaccessible.
- **Shared key**: You can create KMS resources using DEW to share your keys with other accounts. For details, see "Permission Management" > "Sharing" > "Shared KMS" in *Data Encryption Workshop (DEW) User Guide*.

When an encrypted disk is attached, EVS accesses KMS, and KMS sends the data key (DK) to the host memory for use. The disk uses the DK plaintext to encrypt and decrypt disk I/Os. The DK plaintext is only stored in the memory of the host

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

housing the ECS and is not stored persistently on the media. If the custom key is deleted or disabled in KMS, the disk encrypted using the key can still use the DK plaintext stored in the host memory. However, if the disk is detached, the DK plaintext will be deleted from the memory, and the disk cannot be read or written. Before you re-attach this encrypted disk, ensure that the key is enabled.

If disks are encrypted using a custom key, which is then disabled or scheduled for deletion, the disks can no longer be read or written, and data on these disks may never be restored. For details, see **Table 1-3**.

Table 1-3 Impact on encrypted disks after a custom key becomes unavailable

<table>
<tr><th>Custom Key Status</th><th>Impact on Encrypted Disks</th><th>Restoration Method</th></tr>
<tr><td>Disabled</td><td rowspan="3">• If an encrypted disk is then attached to an ECS, the disk can still be used, but normal read/write operations are not guaranteed permanently.
• If an encrypted disk is then detached, re-attaching the disk will fail.</td><td>Enable the CMK. For details, see Enabling a Key.</td></tr>
<tr><td>Scheduled deletion</td><td>Cancel the scheduled deletion for the CMK. For details, see Canceling the Scheduled Deletion of One or More CMKs.</td></tr>
<tr><td>Deleted</td><td>Data on the disks can never be restored.</td></tr>
</table>

Resource and Cost Planning

Table 1-4 Resources and costs

Resource	Description	Monthly Fee
EVS	• Billing mode: Pay-per-use • Purchase method: A data disk can be purchased along with the server or separately.	For details about billing rules, see **Billing for Disks**.
KMS	• Billing mode: Pay-per-use • Key type: Default key. In this case, **ims/default** is used.	For details about billing rules, see **Billing Items**.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

User Permissions

- Security administrators (users having Security Administrator rights) can grant the KMS access rights to EVS for using disk encryption.
- When a common user who does not have the Security Administrator rights needs to use the disk encryption feature, the condition varies depending on whether the user is the first one ever in the current region or project to use this feature.
  - If the user is the first, the user must contact a user having the Security Administrator rights to grant the KMS access rights to EVS. Then, the user can use the disk encryption feature.
  - If the user is not the first, the user can use the disk encryption function directly.

From the perspective of a tenant, as long as the KMS access rights have been granted to EVS in a region, all users in the same region can directly use the disk encryption feature.

If there are multiple projects in the current region, the KMS access permissions need to be granted to each project in this region.

Using KMS to Encrypt a Disk (on the Console)

Step 1 Log in to the EVS console.

Step 2 Click **Buy Disk** in the upper right corner of the EVS console.

Step 3 Select the **Encryption** check box.

1. Click **More**. The **Encryption** check box is displayed.

Figure 1-5 More

2. Create an agency.

   Select **Encryption**. If EVS is not authorized to access KMS, the **Create Agency** dialog box is displayed. In this case, click **Yes** to authorize it. After the authorization, EVS can obtain KMS keys to encrypt and decrypt disks.

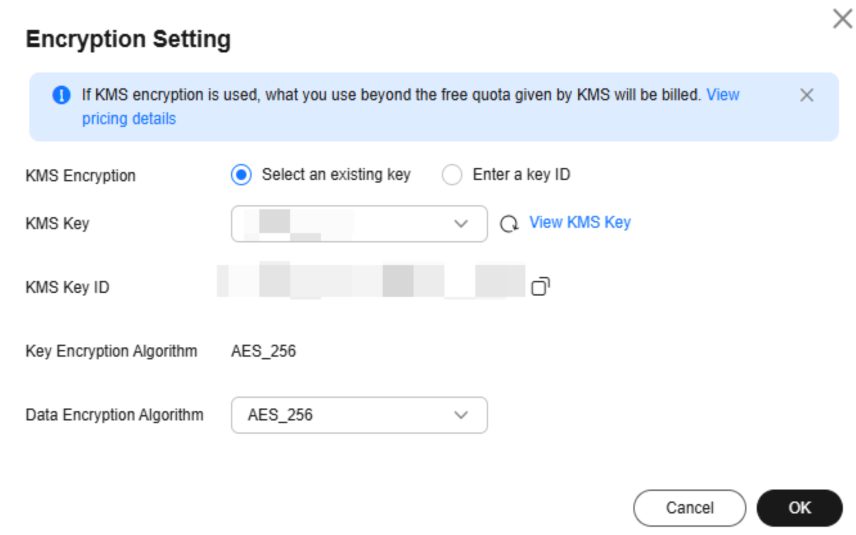
   NOTE

   Before you use the disk encryption function, KMS access rights need to be granted to EVS. If you have the right for granting, grant the KMS access rights to EVS directly. If you do not have the permission, contact a user with the Security Administrator permission for authorization, and then try again.

3. Select **Encryption**. The **Encryption Settings** dialog box is displayed.

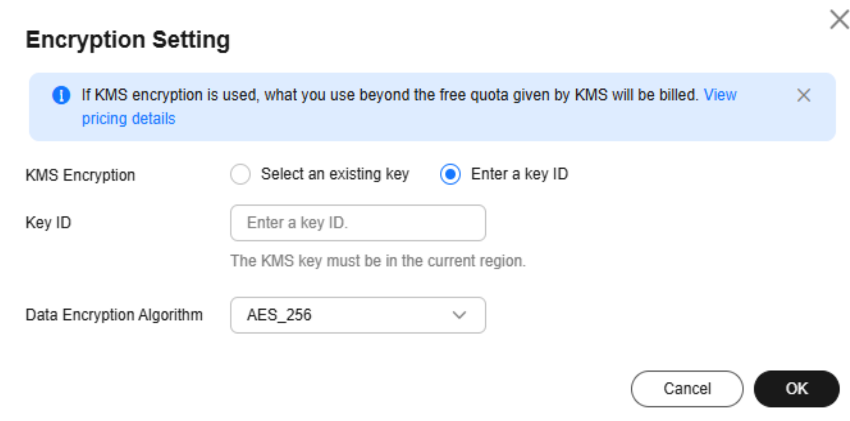
Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-6 Encryption Settings dialog box



4. Set **KMS Encryption**.
   a. Select an existing key.
      i. Click and select the key used for encryption.
         The key name identifies a key. You can select the following keys:
         - Default keys: After the KMS access permission is granted to EVS, the system automatically creates a default key **evs/default**.
         - Custom keys: Keys you already have or just created. For details, see **Creating a Key**.
      ii. Click **View KMS Key** to view all keys.
      iii. Click **OK**.
   b. Enter a key ID.

Figure 1-7 Entering a key ID



      i. Enter the ID of the key used for encryption.
      ii. Click **OK**.

Step 4 Configure other parameters and click **Buy now**.

----End

Using KMS to Encrypt a Disk (Through an API)

You can call the required API of EVS to purchase an encrypted EVS disk. For details, see *Elastic Volume Service API Reference*.

1.2.4 Encrypting Data in IMS

You can use KMS encryption to create private images in Image Management Service (IMS) to securely store data.

Scenario

The IMS server (image) is a template used to create servers or disks, including public images, private images, shared images, and KooGallery images. When you create a private image in IMS, you can use KMS encryption to ensure data security.

You can create an encrypted image in either of the following ways:

- Method 1: **Create an encrypted image using an external image file.**

  When you register an image file as a private image, select **KMS encryption** and select a key.

- Method 2: **Create an encrypted image using an encrypted ECS.**

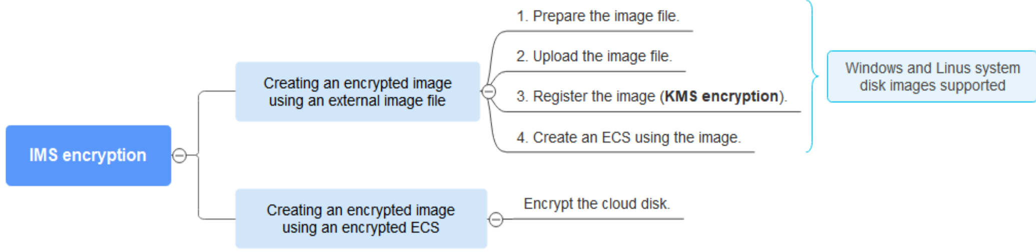
  When you use an ECS to create a private image, if the system disk of the ECS is encrypted, the private image created using the ECS is also encrypted. The key used for encrypting the image must be the same as that used for encrypting the system disk.

This section describes how to use default KMS keys to encrypt IMS image files.

Solution Architecture

Figure 1-8 describes how to use KMS to encrypt an IMS image file.

Figure 1-8 Encrypting IMS



 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Resource and Cost Planning

Table 1-5 Resources and costs

Resource	Description	Monthly Fee
OBS buckets	• Billing mode: Yearly/Monthly • Resource package type: Standard storage (multi-AZ) • Specifications: 100 GB • Quantity: 1	For details about billing rules, see **Billing Items**.
IMS	• Image type: System disk image • Billing Mode: Free	Free. For details about billing rules, see **Billing**.
KMS	• Billing mode: Pay-per-use • Key type: Default key. In this case, **ims/default** is used.	For details about billing rules, see **Billing Items**.

Restrictions

- An encrypted image cannot be shared with other users.
- An encrypted image cannot be published in the Marketplace.
- The key used for encrypting an image cannot be changed.
- If the key used for encrypting an image is disabled or deleted, the image is unavailable.
- The system disk of an ECS created using an encrypted image is also encrypted, and its key is the same as the image key.

Method 1: Creating an Encrypted Image Using an External Image File

Step 1 Prepare an external image file.

- For Windows, prepare an image by referring to **Windows Private Images**.
- For Linux, prepare an image by referring to **Linux Private Images**.

Step 2 Upload the external image file to the OBS bucket. For details, see **Creating a Windows System Disk Image from an External Image File**.

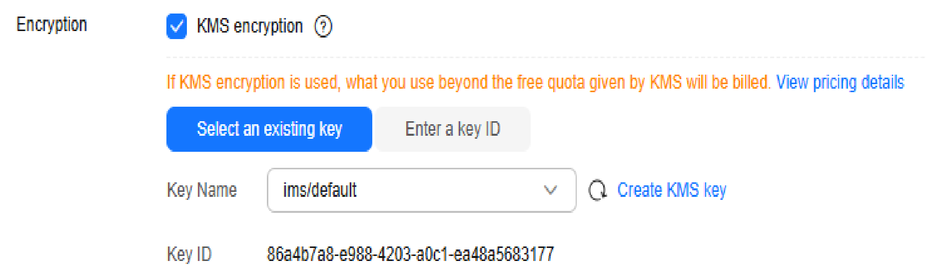
Step 3 Create a private image. Log in to the IMS console. Click the **Private Images** tab and click **Create Image** in the upper right corner.

- **Type**: Select **Import Image**.
- **Image Type**: Select **System disk image**.
- **Select Image File**: Select the bucket that stores the image file in **Step 2**.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

- **Encryption**: Select **KMS encryption**. **Select an existing key** is selected by default. The default key name is **ims/default**.
- For details about other parameters, see **Creating a Windows System Disk Image from an External Image File**.

Figure 1-9 Encryption configuration



Step 4 Create an ECS using an image.

For details, see **Creating an ECS from an Image**.

Note for setting the parameters:

- **Region**: Select the region where the private image is located.
- **Specifications**: Select a flavor based on the OS type in the image and the OS versions described in **OSs Supported by Different Types of ECSs**.
- **Image**: Select **Private image** and then choose the image created in **Step 3** from the drop-down list.
- (Optional) **Data Disk**: Add a data disk, which is created using the image created with the system disk image. In this way, the system disk and data disk data of the VM on the original platform can be migrated to the current cloud platform.

----End

Method 2: Creating an Encrypted Image Using an Encrypted ECS

When you use an ECS to create a private image, if the system disk of the ECS is encrypted, the private image created using the ECS is also encrypted. The key used for encrypting the image is the one used for creating the system disk.

Step 1 Encrypt the EVS system disk. For details, see **Encrypting Data in EVS**.

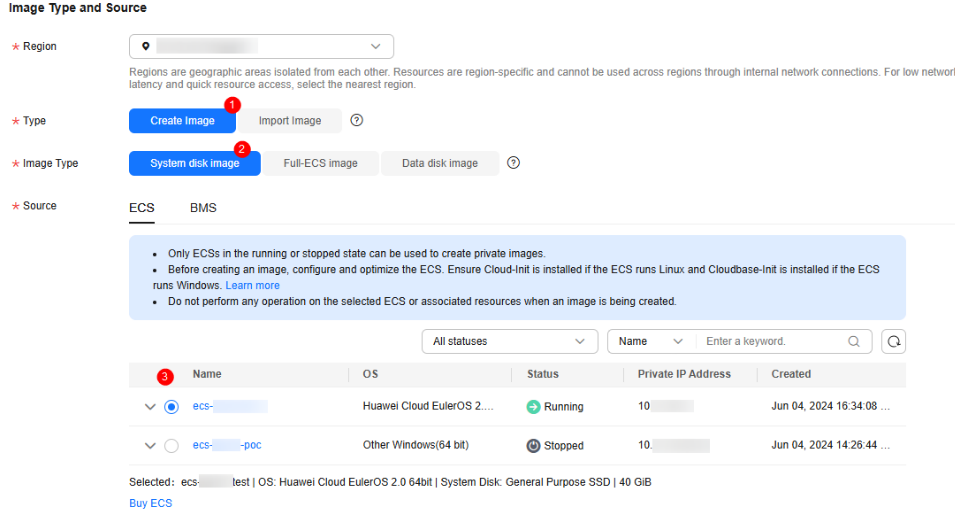
Step 2 When purchasing an ECS, set **Disk Type** to the encrypted system disk in **Step 1**.

Step 3 Create a private image. Log in to the IMS console. Click the **Private Images** tab and click **Create Image** in the upper right corner.

- **Type**: Select **Create Image**.
- **Image Type**: Select **System disk image**.
- **Source**: Select the ECS purchased in **Step 2** from the ECS list.
- For details about other parameters, see **Creating a Windows System Disk Image from an External Image File**.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-10 Creating a private image



Step 4 Click **Next**.

----End

Related Operations

Using KMS to encrypt a private image (API): You can call IMS APIs to create an encrypted image. For details, see *Image Management Service API Reference*.

1.2.5 Encrypting Data in OBS

Scenario

You can use KMS to encrypt all or certain objects in an OBS bucket. When you use KMS encryption in OBS, KMS envelope encryption ensures data encryption and decryption without transmitting a large amount of data over the network. Envelope encryption ensures the confidentiality of data transmission, the efficiency and convenience of data decryption, and information security during object upload and download.

- Full encryption: Encrypt all objects uploaded to an OBS bucket.

  In this case, you only need to encrypt the OBS bucket, as the objects in the bucket inherit the bucket encryption configurations by default. For details, see **Enabling Server-Side Encryption When Creating an OBS Bucket** or **Enabling Encryption for a Created OBS Bucket**.

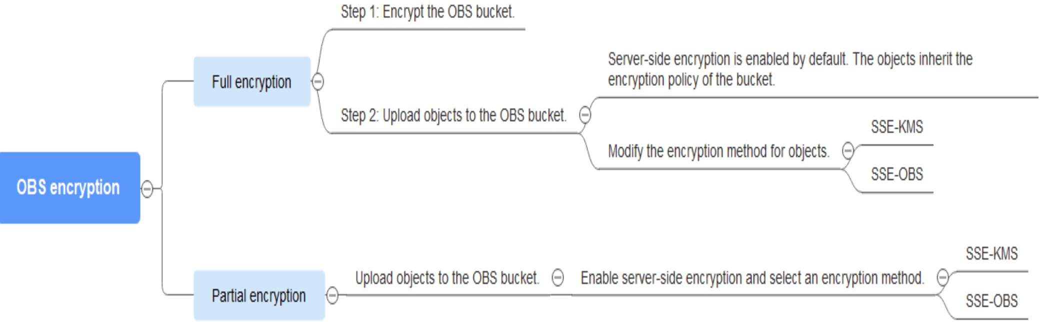
  After an OBS bucket is encrypted, **Inherit from bucket** is enabled by default when you upload objects to the bucket. In this case, the OBS bucket and its objects share the same encryption method. To change the encryption method for the objects, disable **Inherit from bucket** when you upload the objects, and modify the encryption method. For details, see **Uploading Objects to an OBS Bucket**.

- Partial encryption: Encrypt only certain objects uploaded to an OBS bucket.

  In this case, you do not need to encrypt the OBS bucket. Instead, you can directly upload objects to the OBS bucket and configure the encryption method. For details, see **Uploading Objects to an OBS Bucket**.

Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-11 OBS encryption

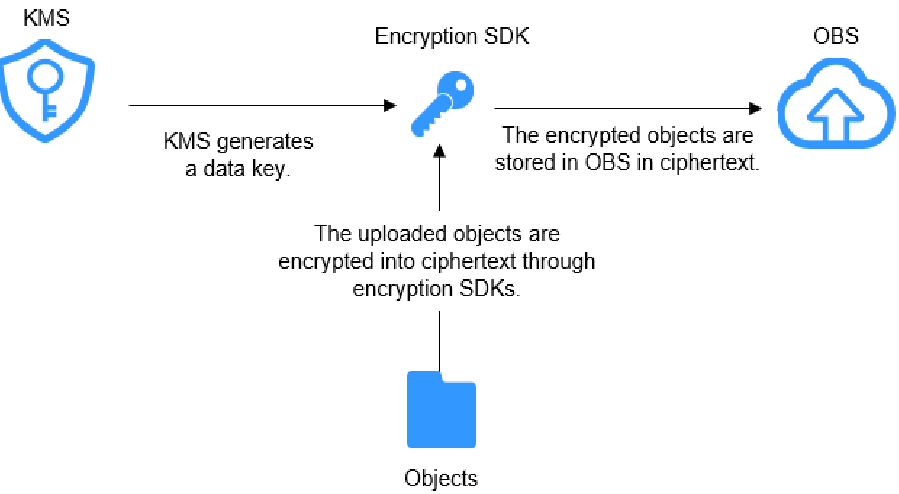


Solution Architecture

The following figures show how objects uploaded to OBS are encrypted and decrypted.

- Encryption principle

Figure 1-12 Encryption principle



a. Obtain the encryption key.

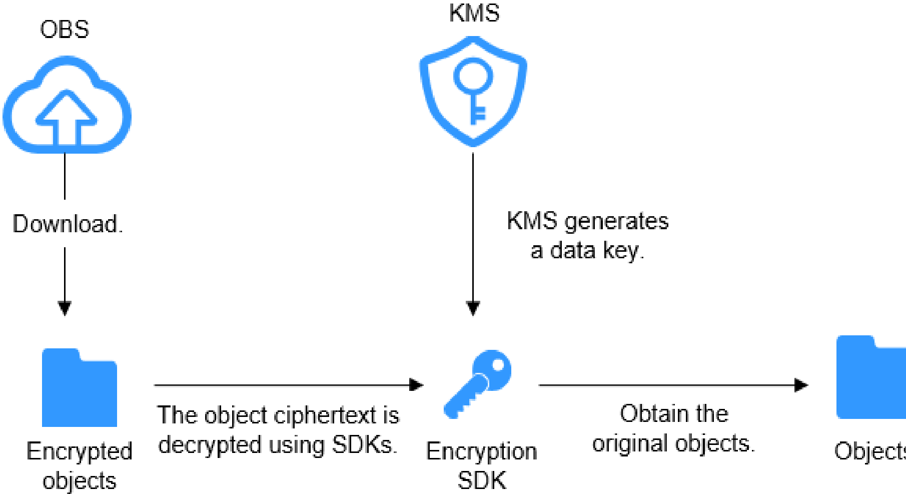
   Generate a data encryption key (DEK) on KMS to encrypt objects in an OBS bucket.

b. Upload encrypted data to the OBS bucket.

   The encryption SDKs encrypt the uploaded data plaintext using the obtained DEK and store the encrypted object ciphertext to OBS.

- Decryption principle

Figure 1-13 Decryption principle



a. Download the objects.

   Download the encrypted object data from OBS.

b. Decrypt the objects.

   The encrypted objects obtain the corresponding ciphertext DEK using the encryption SDKs, and decrypt the ciphertext DEK using KMS to obtain the decrypted original objects.

Constraints

A key in use cannot be deleted. Otherwise, the object encrypted with this key cannot be downloaded.

Enabling Server-Side Encryption When Creating an OBS Bucket

Step 1 Log in to the **OBS console**.

Step 2 In the navigation pane on the left, choose **Buckets**. On the displayed page, click **Create Bucket** in the upper right corner.

Step 3 Under **Properties**, enable **Server-Side Encryption**, select **SSE-KMS** for **Encryption Method**, and select an encryption key type.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-14 Encrypting data in OBS

> **NOTE**
>
> OBS uses the encryption key provided by KMS. You can select any of the following keys:
>
> - Default key **obs/default**. If you do not have a default key, OBS automatically creates one when you upload an object for the first time.
> - Custom keys created on KMS. For details, see **Creating a Key**.
> - Keys using the SM4 cryptographic algorithm, which is supported only in CN North-Ulanqab 1.

Step 4 Configure other parameters and click **Create Now**.

----End

Enabling Encryption for a Created OBS Bucket

Step 1 In the navigation pane on the left, choose **Buckets**. Click the target bucket and access the **Objects** page.

Step 2 In the navigation pane on the left, choose **Overview**.

Step 3 In the **Basic Configurations** area, click **Server-Side Encryption**.

Step 4 In the displayed dialog box, enable server-side encryption, set **Encryption Method** to **SSE-KMS**, and select an encryption key type.

Figure 1-15 Enabling server-side encryption

NOTE

OBS uses the encryption key provided by KMS. You can select any of the following keys:

- Default key **obs/default**. If you do not have a default key, OBS automatically creates one when you upload an object for the first time.
- Custom keys created on KMS. For details, see **Creating a Key**.
- Keys using the SM4 cryptographic algorithm, which is supported only in CN North-Ulanqab 1.

Step 5 Configure other parameters and click **OK**.

----End

Uploading Objects to an OBS Bucket

Step 1 Click the target bucket in the list on the OBS console.

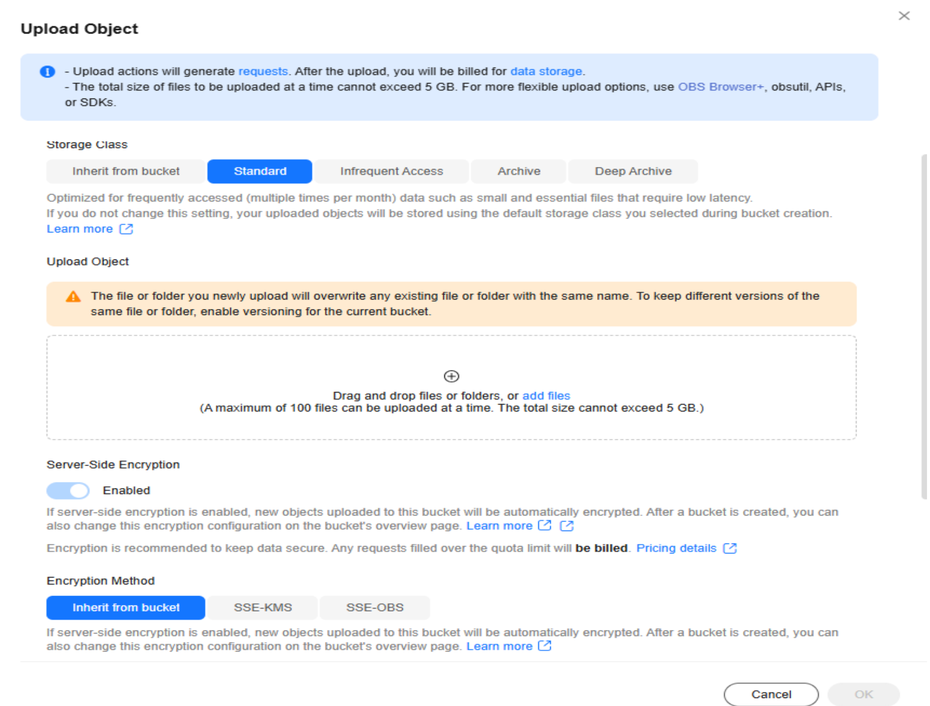
Step 2 In the navigation pane on the left, choose **Objects**.

Step 3 Click **Upload Object**.

Step 4 In the displayed dialog box, add files to be uploaded.

Step 5 For **Server-Side Encryption**, select an encryption method, and select a default key or custom key from the drop-down list, as shown in **Figure 1-16**.

Figure 1-16 Uploading an object with server-side encryption enabled (OBS bucket encryption enabled)

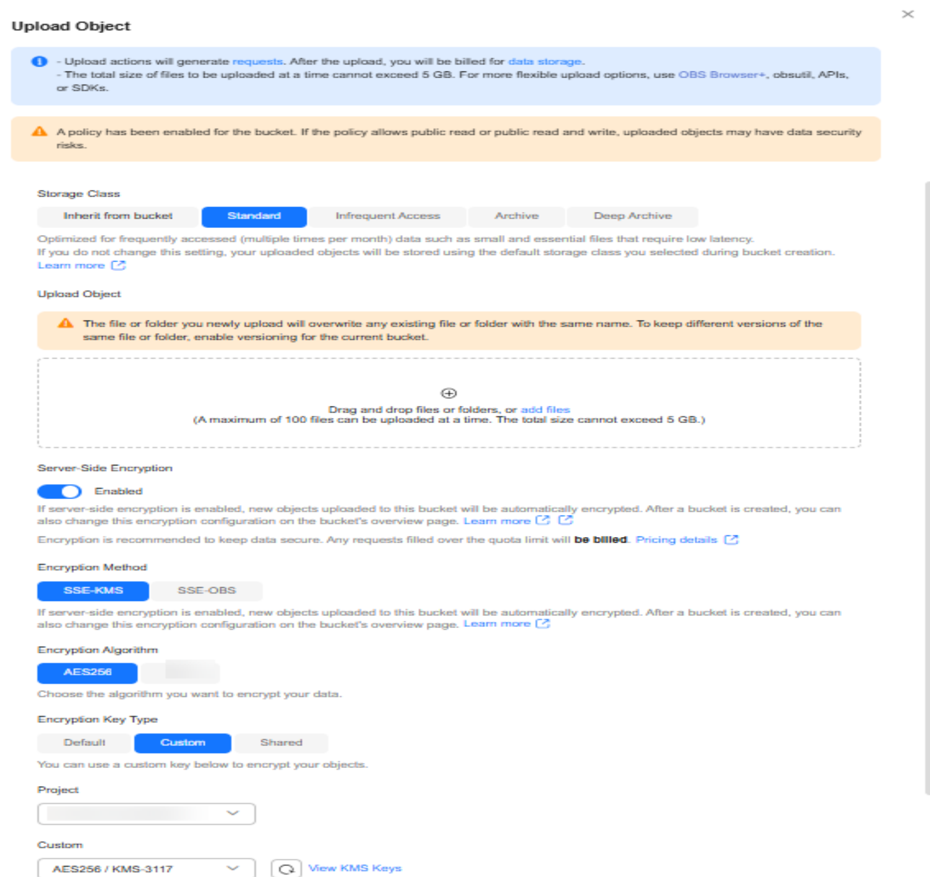


NOTE

- After server-side encryption is enabled for the OBS bucket, the encryption configuration is inherited by default when an object is uploaded.
- To modify the encryption configuration, you need to disable **Inherit from bucket** and select **SSE-KMS** or **SSE-OBS** as required.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-17 Uploading an object with server-side encryption enabled (OBS bucket encryption disabled)



NOTE

If OBS bucket encryption is not enabled, you need to enable server-side encryption when uploading objects.

Step 6 After uploading the object, click it to view its encryption status.

NOTE

- The object encryption status cannot be changed.
- A key in use cannot be deleted. Otherwise, the object encrypted with this key cannot be downloaded.

----End

Related Operations

Alternatively, you can call OBS APIs to upload a file with server-side encryption using KMS-managed keys (SSE-KMS). For details, see **Configuring Bucket Encryption**.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

1.2.6 Encrypting an RDS DB Instance

Overview

Relational Database Service (RDS) supports MySQL and PostgreSQL engines.

After encryption is enabled, disk data will be encrypted and stored on the server when you create a DB instance or expand disk capacity. When you download encrypted objects, the encrypted data will be decrypted on the server and displayed in plaintext.

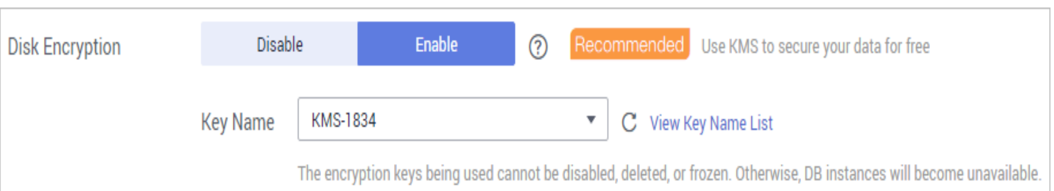
Restrictions

- The KMS Administrator right must be granted to the user in the region of RDS by using Identity and Access Management (IAM). For details about how to assign permissions to user groups, see "How Do I Manage User Groups and Grant Permissions to Them?" in *Identity and Access Management User Guide*.
- To use a user-defined key to encrypt objects to be uploaded, create a key using DEW.
- Once the disk encryption function is enabled, you cannot disable it or change the key after a DB instance is created. The backup data stored in OBS will not be encrypted.
- After an RDS DB instance is created, do not disable or delete the key that is being used. Otherwise, RDS will be unavailable and data cannot be restored.
- If you scale up a DB instance with disks encrypted, the expanded storage space will be encrypted using the original encryption key.

Using KMS to Encrypt a DB Instance (on the Console)

When purchasing a DB instance on the RDS console, you can enable disk encryption to use KMS-provided keys to encrypt DB instance disks.

Figure 1-18 Encrypting data in RDS



Using KMS to Encrypt a DB Instance (Through an API)

You can also call the required API of RDS to purchase encrypted DB instances. For details, see *Relational Database Service API Reference*.

1.2.7 Encrypting a DDS DB Instance

Overview

After encryption is enabled, disk data will be encrypted and stored on the server when you create a DB instance or expand disk capacity. When you download

Copyright © Huawei Cloud Computing Technologies Co., Ltd.

encrypted objects, the encrypted data will be decrypted on the server and displayed in plaintext.

Restrictions

- The KMS Administrator right must be added in the region of RDS using IAM. For details about how to assign permissions to user groups, see "How Do I Manage User Groups and Grant Permissions to Them?" in *Identity and Access Management User Guide*.
- To use a user-defined key to encrypt objects to be uploaded, create a key using DEW. For details, see **Creating a Key**.
- Once the disk encryption function is enabled, you cannot disable it or change the key after a DB instance is created. The backup data stored in OBS will not be encrypted.
- After a Document Database Service (DDS) DB instance is created, do not disable or delete the key that is being used. Otherwise, DDS will be unavailable and data cannot be restored.
- If you scale up a DB instance with disks encrypted, the expanded storage space will be encrypted using the original encryption key.

Using KMS to Encrypt a DB Instance (on the Console)

When you purchase a DB instance in DDS, you can set **Disk Encryption** to **Enable** and use the key provided by KMS to encrypt the disk of the DB instance. For more information, see **Buying a Cluster Instance**.

Figure 1-19 Encrypting data in DDS

Using KMS to Encrypt a DB Instance (Through an API)

You can also call the required API of DDS to purchase encrypted DB instances. For details, see *Document Database Service API Reference*.

1.3 Using the Encryption SDK to Encrypt and Decrypt Local Files

You can use certain algorithms to encrypt your files, protecting them from being breached or tampered with.

Encryption SDK is a client password library that can encrypt and decrypt data and file streams. You can easily encrypt and decrypt massive amounts of data simply by calling APIs. It allows you to focus on developing the core functions of your applications without being distracted by the data encryption and decryption processes.

Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Scenario

If large files and images are sent to KMS through HTTPS for encryption, a large number of network resources will be consumed and the encryption will be slow. This section describes how to quickly encrypt a large amount of data.

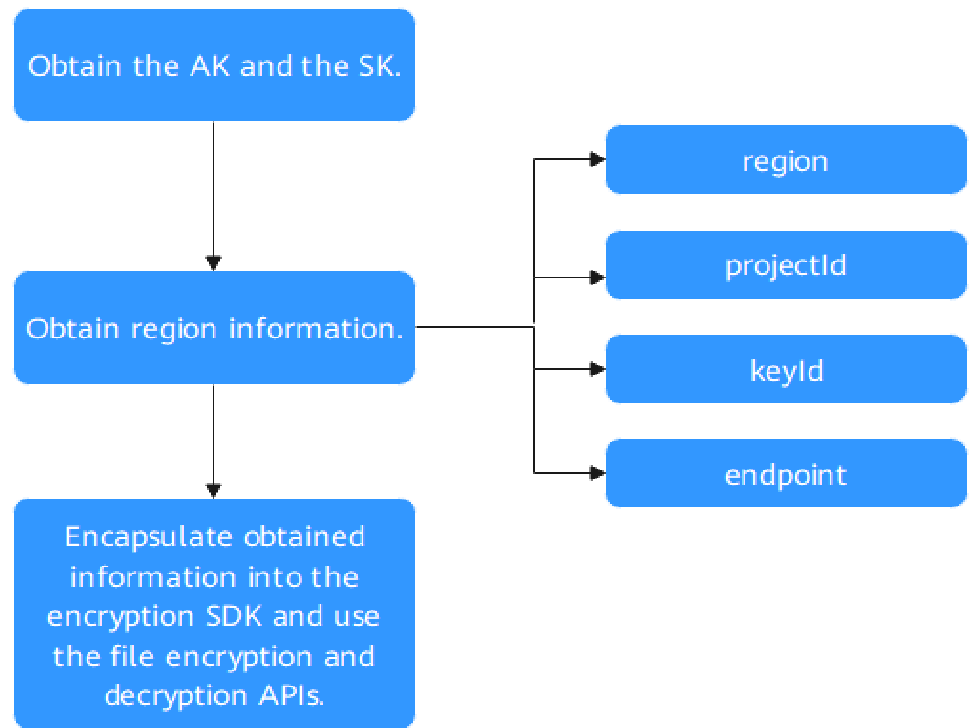
Solution

Encryption SDK performs envelope encryption on file streams segment by segment.

Data is encrypted within the SDK by using the DEK generated by KMS. Segmented encryption of files in the memory ensures the security and correctness of file encryption, because it does not require file transfer over the network.

The SDK loads a file to memory and processes it segment by segment. The next segment will not be read before the encryption or decryption of the current segment completes.

Process



Procedure

Step 1 Obtain the AK and the SK.

- **ACCESS_KEY**: Access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**
- **SECRET_ACCESS_KEY**: Secret access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**

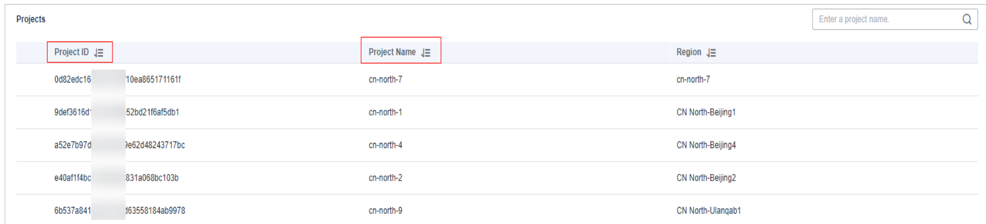
Copyright © Huawei Cloud Computing Technologies Co., Ltd.

- **PROJECT_ID**: site project ID. For details, see **Obtaining a Project ID**.
- **KMS_ENDPOINT**: endpoint for accessing KMS.
- There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
- In this example, the AK/SK stored in the environment variables are used for identity authentication. Configure the environment variables **HUAWEICLOUD_SDK_AK** and **HUAWEICLOUD_SDK_SK** in the local environment first.

Step 2 Obtain region information.

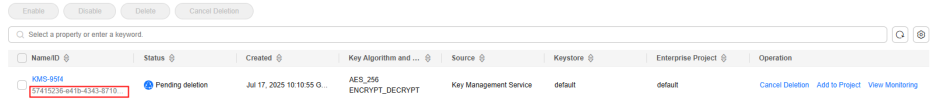
1. Log in to the **DEW console**.
2. Hover over the username in the upper right corner and choose **My Credentials** from the drop-down list.
3. Obtain the **Project ID** and **Project Name**.

Figure 1-20 Obtaining the project ID and project name



4. Click on the left and choose **Security** > **Data Encryption Workshop**.
5. Obtain the ID of the CMK (**KEYID**) to be used in the current region.

Figure 1-21 Obtaining the CMK ID



6. Obtain the endpoint (**ENDPOINT**) required by the current region.

Step 3 Encrypt and decrypt a file.

```
public class KmsEncryptFileExample {

    private static final String ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_AK");
    private static final String SECRET_ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_SK");
    private static final String PROJECT_ID = "<projectId>";
    private static final String REGION = "<region>";
    private static final String KEYID = "<keyId>";
    public static final String ENDPOINT = "<endpoint>";

    public static void main(String[] args) throws IOException {
        // Source file path
        String encryptFileInPutPath = args[0];
        // Path of the encrypted ciphertext file
        String encryptFileOutPutPath = args[1];
        // Path of the decrypted file
        String decryptFileOutPutPath = args[2];
        // Encryption context
```


Copyright © Huawei Cloud Computing Technologies Co., Ltd.

```
        Map<String, String> encryptContextMap = new HashMap<>();
        encryptContextMap.put("encryption", "context");
        encryptContextMap.put("simple", "test");
        encryptContextMap.put("caching", "encrypt");
        // Construct the encryption configuration
        HuaweiConfig config = HuaweiConfig.builder().buildSk(SECRET_ACCESS_KEY)
                .buildAk(ACCESS_KEY)
                .buildKmsConfig(Collections.singletonList(new KMSConfig(REGION, KEYID, PROJECT_ID,
ENDPOINT)))
                .buildCryptoAlgorithm(CryptoAlgorithm.AES_256_GCM_NOPADDING)
                .build();
        HuaweiCrypto huaweiCrypto = new HuaweiCrypto(config);
        // Set the key ring.
        huaweiCrypto.withKeyring(new
KmsKeyringFactory().getKeyring(KeyringTypeEnum.KMS_MULTI_REGION.getType()));
        // Encrypt the file.
        encryptFile(encryptContextMap, huaweiCrypto, encryptFileInPutPath, encryptFileOutPutPath);
        // Decrypt the file.
        decryptFile(huaweiCrypto, encryptFileOutPutPath, decryptFileOutPutPath);
    }

    private static void encryptFile(Map<String, String> encryptContextMap, HuaweiCrypto huaweiCrypto,
                                    String encryptFileInPutPath, String encryptFileOutPutPath) throws IOException {
        // fileInputStream: input stream corresponding to the encrypted file
        FileInputStream fileInputStream = new FileInputStream(encryptFileInPutPath);
        // fileOutputStream: output stream corresponding to the source file
        FileOutputStream fileOutputStream = new FileOutputStream(encryptFileOutPutPath);
        // Encryption
        huaweiCrypto.encrypt(fileInputStream, fileOutputStream, encryptContextMap);
        fileInputStream.close();
        fileOutputStream.close();
    }

    private static void decryptFile(HuaweiCrypto huaweiCrypto, String decryptFileInPutPath, String
decryptFileOutPutPath) throws IOException {
        // in: input stream corresponding to the source file
        FileInputStream fileInputStream = new FileInputStream(decryptFileInPutPath);
        // out: output stream corresponding to the encrypted file
        FileOutputStream fileOutputStream = new FileOutputStream(decryptFileOutPutPath);
        // Decryption
        huaweiCrypto.decrypt(fileInputStream, fileOutputStream);
        fileInputStream.close();
        fileOutputStream.close();
    }
}
```

----End

1.4 Encrypting and Decrypting Data Through Cross-region DR

Scenario

If a fault occurs during encryption or decryption in a region, you can use KMS to implement cross-region DR encryption and decryption, ensuring service continuity.

Solution

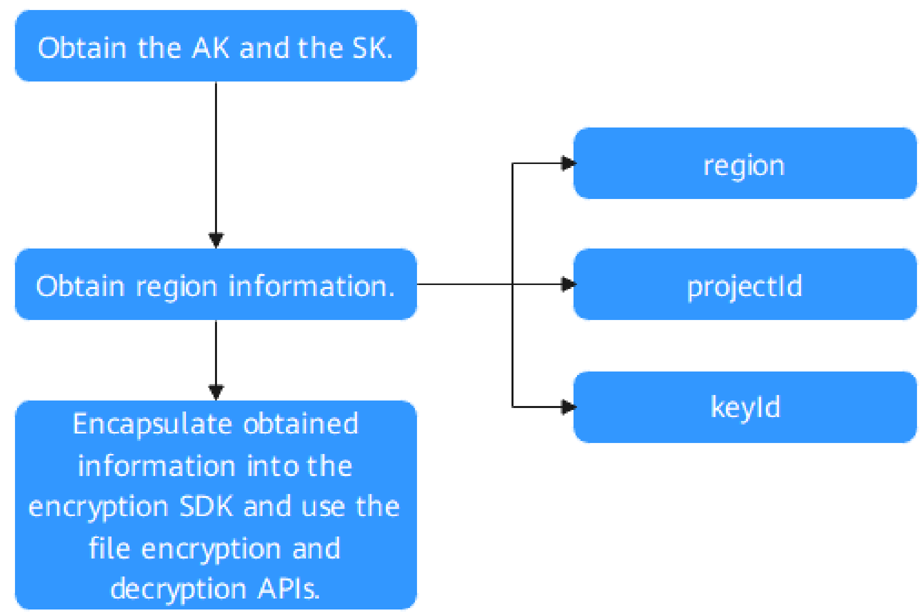
If KMS is faulty in one or multiple regions, encryption and decryption can be completed as long as a key in the key ring is available.

A cross-region key can use the CMKs of multiple regions to encrypt a piece of data and generate unique data ciphertext. To decrypt the data, you simply need to use

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

a key ring that contains one or more available CMKs that were used for encrypting the data.

Process



Procedure

Step 1 Obtain the AK and the SK.

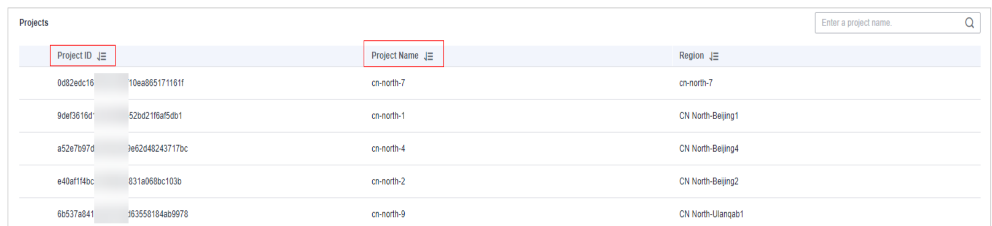
- **ACCESS_KEY**: Access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**
- **SECRET_ACCESS_KEY**: Secret access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**
- **PROJECT_ID**: site project ID. For details, see **Obtaining a Project ID**.
- **KMS_ENDPOINT**: endpoint for accessing KMS.
- There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
- In this example, the AK/SK stored in the environment variables are used for identity authentication. Configure the environment variables **HUAWEICLOUD_SDK_AK** and **HUAWEICLOUD_SDK_SK** in the local environment first.

Step 2 Obtain region information.

1. Log in to the **DEW console**.
2. Hover over the username in the upper right corner and choose **My Credentials** from the drop-down list.
3. Obtain the **Project ID** and **Project Name**.

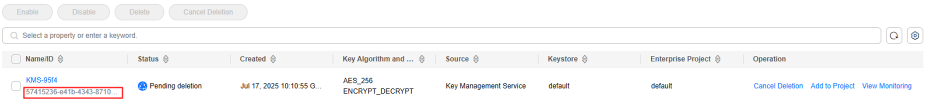
 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-22 Obtaining the project ID and project name



4. Click ☰ on the left and choose **Security** > **Data Encryption Workshop**.
5. Obtain the ID of the CMK (**KEYID**) to be used in the current region.

Figure 1-23 Obtaining the CMK ID



Step 3 Use the key ring for encryption and decryption.

```
public class KmsEncryptionExample {
    private static final String ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_AK");
    private static final String SECRET_ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_SK");

    private static final String PROJECT_ID_1 = "<projectId1>";
    private static final String REGION_1 = "<region1>";
    private static final String KEYID_1 = "<keyId1>";

    public static final String PROJECT_ID_2 = "<projectId2>";
    public static final String REGION_2 = "<region2>";
    public static final String KEYID_2 = "<keyId2>";

    // Data to be encrypted
    private static final String PLAIN_TEXT = "Hello World!";

    public static void main(String[] args) {
        // CMK list
        List<KMSConfig> kmsConfigList = new ArrayList<>();
        kmsConfigList.add(new KMSConfig(REGION_1, KEYID_1, PROJECT_ID_1));
        kmsConfigList.add(new KMSConfig(REGION_2, KEYID_2, PROJECT_ID_2));
        // Construct encryption-related information.
        HuaweiConfig multiConfig = HuaweiConfig.builder().buildSk(SECRET_ACCESS_KEY)
                .buildAk(ACCESS_KEY)
                .buildKmsConfig(kmsConfigList)
                .buildCryptoAlgorithm(CryptoAlgorithm.AES_256_GCM_NOPADDING)
                .build();
        // Select a key ring.
        KMSKeyring keyring = new
KmsKeyringFactory().getKeyring(KeyringTypeEnum.KMS_MULTI_REGION.getType());
        HuaweiCrypto huaweiCrypto = new HuaweiCrypto(multiConfig).withKeyring(keyring);
        // Encryption context
        Map<String, String> encryptContextMap = new HashMap<>();
        encryptContextMap.put("key", "value");
        encryptContextMap.put("context", "encrypt");
        // Encryption
        CryptoResult<byte[]> encryptResult = huaweiCrypto.encrypt(new EncryptRequest(encryptContextMap,
PLAIN_TEXT.getBytes(StandardCharsets.UTF_8)));
        // Decryption
        CryptoResult<byte[]> decryptResult = huaweiCrypto.decrypt(encryptResult.getResult());
        Assert.assertEquals(PLAIN_TEXT, new String(decryptResult.getResult()));
```

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

```
  }
}
```

----End

1.5 Using KMS to Protect File Integrity

Scenario

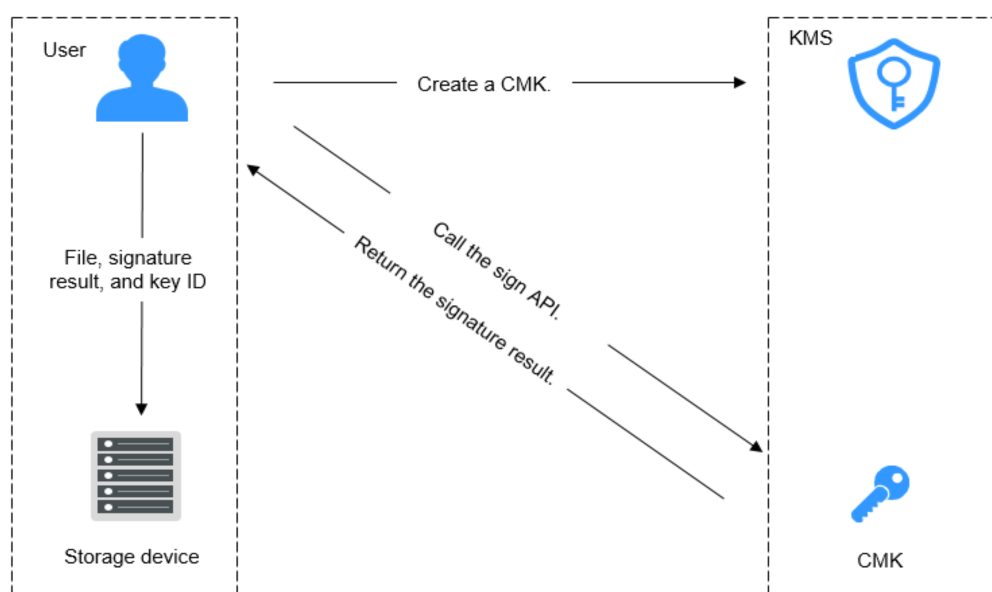
When a large amount of files (such as images, electronic insurance policies, and important files) need to be transmitted or stored securely, you can use KMS to sign the file digest. When the files are used again, you can recalculate the digest for signature verification. Ensure that files are not tampered with during transmission or storage.

Solution

Create a CMK on KMS.

Calculate the file digest and call the sign API of KMS to sign the digest. The signature result of the digest is obtained. Transmit or store the digest signature result, key ID, and the file together. The following figure shows the signature process.

Figure 1-24 Signature process

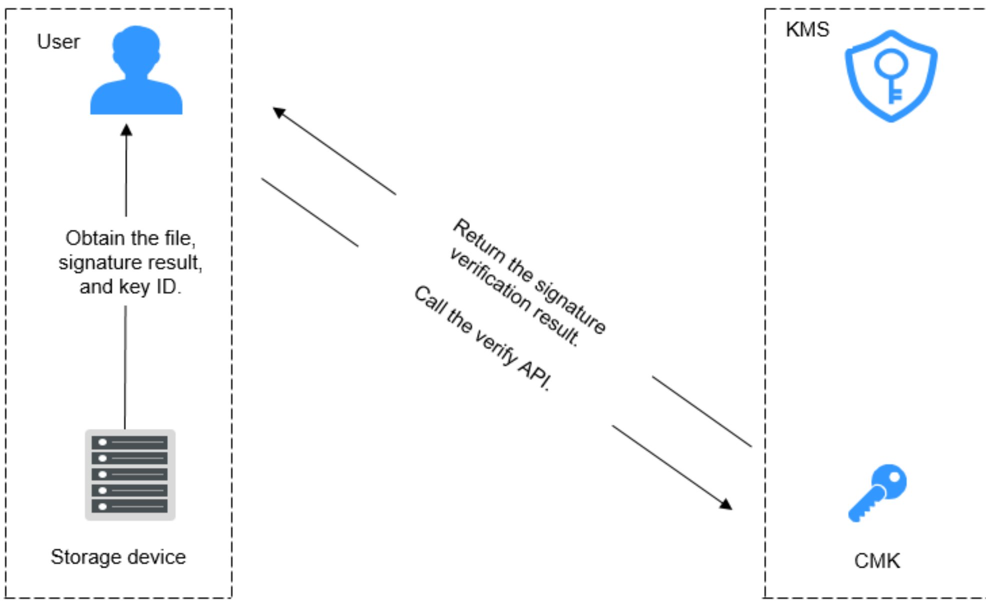


Before using a file, you need to check the integrity of the file to ensure that the file is not tampered with.

Recalculate the file digest and call the verify API of KMS with the signature value to verify the signature for the digest. The signature verification result is obtained. If the signature is verified, the file has not been tampered with. The following figure shows the signature verification process.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 1-25 Signature verification process.



Procedure

Step 1 Obtain the AK and the SK.

- **ACCESS_KEY**: Access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**
- **SECRET_ACCESS_KEY**: Secret access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**
- **PROJECT_ID**: site project ID. For details, see **Obtaining a Project ID**.
- **KMS_ENDPOINT**: endpoint for accessing KMS.
- There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
- In this example, the AK/SK stored in the environment variables are used for identity authentication. Configure the environment variables **HUAWEICLOUD_SDK_AK** and **HUAWEICLOUD_SDK_SK** in the local environment first.

Step 2 Use KMS to sign the file and verify the signature.

```
public class FileStreamSignVerifyExample {

    /**
     * Basic authentication information:
     * - ACCESS_KEY: access key of the Huawei Cloud account
     * - SECRET_ACCESS_KEY: secret access key of the Huawei Cloud account, which is sensitive information.
Store this in ciphertext.
     * - IAM_ENDPOINT: endpoint for accessing IAM.
     * - KMS_REGION_ID: regions supported by KMS.
     * - KMS_ENDPOINT: endpoint for accessing KMS.
     */
    private static final String ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_AK");
    private static final String SECRET_ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_SK");
    private static final String IAM_ENDPOINT = "https://<IamEndpoint>";
    private static final String KMS_REGION_ID = "<RegionId>";
    private static final String KMS_ENDPOINT = "https://<KmsEndpoint>";
```

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

```
    public static void main(String[] args) {
        // CMK ID. Select a key whose usage contains SIGN_VERIFY.
        final String keyId = args[0];

        signAndVerifyFile(keyId);
    }

    /**
     * Use KMS to sign the file and verify the signature.
     *
     * @param keyId: CMK ID
     */
    static void signAndVerifyFile(String keyId) {
        // 1. Prepare the authentication information for accessing Huawei Cloud.
        final BasicCredentials auth = new BasicCredentials()
                .withIamEndpoint(IAM_ENDPOINT).withAk(ACCESS_KEY).withSk(SECRET_ACCESS_KEY);

        // 2. Initialize the SDK and transfer the authentication information and the address for the KMS to 
access the client.
        final KmsClient kmsClient = KmsClient.newBuilder()
                .withRegion(new Region(KMS_REGION_ID, KMS_ENDPOINT)).withCredential(auth).build();

        // 3. Prepare the file to be signed.
        // inFile File to be signed
        final File inFile = new File("FirstSignFile.iso");
        final String fileSha256Sum = getFileSha256Sum(inFile);

        // 4. Calculate the digest and select a proper signature algorithm based on the key type.
        final SignRequest signRequest = new SignRequest().withBody(
                new 
SignRequestBody().withKeyId(keyId).withSigningAlgorithm(SignRequestBody.SigningAlgorithmEnum.RSASSA
_PSS_SHA_256)
                        .withMessageType(SignRequestBody.MessageTypeEnum.DIGEST).withMessage(fileSha256Su
m));

        final SignResponse signResponse = kmsClient.sign(signRequest);

        // 5. Verify the digest.
        final ValidateSignatureRequest validateSignatureRequest = new ValidateSignatureRequest().withBody(
                new 
VerifyRequestBody().withKeyId(keyId).withMessage(fileSha256Sum).withSignature(signResponse.getSignatur
e())
                        .withSigningAlgorithm(VerifyRequestBody.SigningAlgorithmEnum.RSASSA_PSS_SHA_256)
                        .withMessageType(VerifyRequestBody.MessageTypeEnum.DIGEST));
        final ValidateSignatureResponse validateSignatureResponse = 
kmsClient.validateSignature(validateSignatureRequest);

        // 6. Compare the digest result.
        assert validateSignatureResponse.getSignatureValid().equalsIgnoreCase("true");

    }

    /**
     * Calculate the SHA256 digest of the file.
     *
     * @param file
     * @return SHA256 digest in Base64 format
     */
    static String getFileSha256Sum(File file) {
        int length;
        MessageDigest sha256;
        byte[] buffer = new byte[1024];
        try {
            sha256 = MessageDigest.getInstance("SHA-256");
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e.getMessage());
        }
        try (FileInputStream inputStream = new FileInputStream(file)) {
```

```
            while ((length = inputStream.read(buffer)) != -1) {
                sha256.update(buffer, 0, length);
            }
            return Base64.getEncoder().encodeToString(sha256.digest());
        } catch (IOException e) {
            throw new RuntimeException(e.getMessage());
        }
    }

}
```

----End

2 Cloud Secret Management Service

2.1 Using CSMS to Change Hard-coded Database Account Passwords

Generally, the secrets used for access are embedded in applications. To update a secret, you need to create a secret and spend time updating your applications. If you have multiple applications using the same secret, you have to update all of them, or the applications you forgot to update will be unable to use the secret for login.

An easy-to-use, effective, and secure secret management tool will be helpful.

Cloud Secret Management Service (CSMS) has the following advantages:

- You can host your secrets instead of using hardcoded secrets, improving the security of data and assets.
- Secure SDK access allows you to dynamically call your secrets.
- You can store many types of secrets. You can store service accounts, passwords, and database information, including but not limited to database names, IP addresses, and port numbers.

Logging In to a Database Using Secrets

You can create a secret and log in to your database by calling the secret via an API.

Ensure your account has the KMS Administrator or KMS CMKFullAccess permission. For details, see .

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 2-1 Secret-based login process

Administrator ① CSMS ② ③ Third-party application ④ Database

The process is as follows:

Step 1 Create a secret on the or via an **API** to store database information (such as the database address, port, and password).

Step 2 Use an application to access the database. CSMS will query the secret created in **1**.

Step 3 CSMS retrieves and decrypts the secret ciphertext and securely returns the information stored in the secret to the application through the secret management API.

Step 4 The application obtains the decrypted plaintext secret and uses it to access the database.

----End

Secret Creation and Query APIs

You can call the following APIs to create secrets, save their content, and query secret information.

API	Description
Creating a Secret	This API is used to create a secret and store the secret value in the initial secret version.
Querying a Secret	This API is used to query a secret.

Creating and Querying Secrets via APIs

1. Prepare basic authentication information.
   - **ACCESS_KEY**: Access key of the Huawei account
   - **SECRET_ACCESS_KEY**: Secret access key of the Huawei account
   - **PROJECT_ID**: project ID of a Huawei Cloud site. For details, see **Obtaining Account, IAM User, Group, Project, Region, and Agency Information**.
   - **CSMS_ENDPOINT**: endpoint for accessing CSMS.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

– There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
– In this example, the AK/SK stored in the environment variables are used for identity authentication. Configure the environment variables **HUAWEICLOUD_SDK_AK** and **HUAWEICLOUD_SDK_SK** in the local environment first.

2. Create and query secret information.

   Secret name: **secretName**

   Secret value: **secretString**

   Secret version value: **LATEST_SECRET**

   Secret version: **versionId**

```
import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.csms.v1.CsmsClient;
import com.huaweicloud.sdk.csms.v1.model.CreateSecretRequest;
import com.huaweicloud.sdk.csms.v1.model.CreateSecretRequestBody;
import com.huaweicloud.sdk.csms.v1.model.CreateSecretResponse;
import com.huaweicloud.sdk.csms.v1.model.ShowSecretVersionRequest;
import com.huaweicloud.sdk.csms.v1.model.ShowSecretVersionResponse;

public class CsmsCreateSecretExample {
    /**
     * Basic authentication information:
     * - ACCESS_KEY: Access key of the Huawei account
     * - SECRET_ACCESS_KEY: Secret access key of the Huawei account
     * - PROJECT_ID: Huawei Cloud project ID. For details, see https://support.huaweicloud.com/eu/
productdesc-iam/iam_01_0023.html
* - CSMS_ENDPOINT: endpoint address for accessing CSMS.
     * - There will be security risks if the AK/SK used for authentication is directly written into code.
Encrypt the AK/SK in the configuration file or environment variables for storage.
     * - In this example, the AK/SK stored in the environment variables are used for identity
authentication. Configure the environment variables HUAWEICLOUD_SDK_AK and
HUAWEICLOUD_SDK_SK in the local environment first.
     */
    private static final String ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_AK");
    private static final String SECRET_ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_SK");
    private static final String PROJECT_ID = "<ProjectID>";
    private static final String CSMS_ENDPOINT = "<CsmsEndpoint>";

    //Version ID used to query the latest secret version details
    private static final String LATEST_SECRET = "latest";

    public static void main(String[] args) {
        String secretName = args[0];
        String secretString = args[1];

        //Create a secret.
        createSecret(secretName, secretString);

        //Query the content of the new secret based on the secret version latest or v1.
        ShowSecretVersionResponse latestVersion = showSecretVersion(secretName, LATEST_SECRET);
        ShowSecretVersionResponse firstVersion = showSecretVersion(secretName, "v1");

        assert latestVersion.equals(firstVersion);
        assert latestVersion.getVersion().getSecretString().equalsIgnoreCase(secretString);
    }

    /**
     * Create a secret.
     * @param secretName
     * @param secretString
     */
    private static void createSecret(String secretName, String secretString) {
```

```
        CreateSecretRequest secret = new CreateSecretRequest().withBody(
                new CreateSecretRequestBody().withName(secretName).withSecretString(secretString));

        CsmsClient csmsClient = getCsmsClient();

        CreateSecretResponse createdSecret = csmsClient.createSecret(secret);

        System.out.printf("Created secret success, secret detail:%s", createdSecret);
    }
    /**
     * Query secret version details based on the secret version ID.
     * @param secretName
     * @param versionId
     * @return
     */
    private static ShowSecretVersionResponse showSecretVersion(String secretName, String versionId) {
        ShowSecretVersionRequest showSecretVersionRequest = new
ShowSecretVersionRequest().withSecretName(secretName)
                .withVersionId(versionId);

        CsmsClient csmsClient = getCsmsClient();

        ShowSecretVersionResponse version = csmsClient.showSecretVersion(showSecretVersionRequest);

        System.out.printf("Query secret success. version id:%s",
version.getVersion().getVersionMetadata().getId());

        return version;
    }

    /**
     * Obtain the CSMS client.
     * @return
     */
    private static CsmsClient getCsmsClient() {
        BasicCredentials auth = new BasicCredentials()
                .withAk(ACCESS_KEY)
                .withSk(SECRET_ACCESS_KEY)
                .withProjectId(PROJECT_ID);

        return CsmsClient.newBuilder().withCredential(auth).withEndpoint(CSMS_ENDPOINT).build();
    }
}
```

Obtaining the Database Account Through an Application

1. Obtain the dependency statement of the CSMS SDK.

   Example:

```
<dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>XXX</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.8.9</version>
        </dependency>
        <dependency>
            <groupId>com.huaweicloud.sdk</groupId>
            <artifactId>huaweicloud-sdk-csms</artifactId>
            <version>3.0.79</version>
        </dependency>
```

2. Establish a database connection and obtain the account.

   Example:

```
import com.google.gson.Gson;
import com.google.gson.JsonObject;
```

```
import com.huaweicloud.sdk.csms.v1.model.ShowSecretVersionRequest;
import com.huaweicloud.sdk.csms.v1.model.ShowSecretVersionResponse;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

 // Obtain the specified database account based on the secret information.
    public static Connection getMySQLConnectionBySecret(String secretName, String jdbcUrl) throws
ClassNotFoundException, SQLException{
        Class.forName(MYSQL_JDBC_DRIVER);
        ShowSecretVersionResponse latestVersionValue = getCsmsClient().showSecretVersion(new
ShowSecretVersionRequest().withSecretName(secretName).withVersionId("latest"));
        String secretString = latestVersionValue.getVersion().getSecretString();
        JsonObject jsonObject = new Gson().fromJson(secretString, JsonObject.class);
        return DriverManager.getConnection(jdbcUrl, jsonObject.get("username").getAsString(),
jsonObject.get("password").getAsString());
    }
```

2.2 Using CSMS to Prevent AK/SK Leakage

CSMS is a secure, reliable, and easy-to-use credential hosting service. Users or applications can use CSMS to create, retrieve, update, and delete credentials in a unified manner throughout the credential lifecycle. CSMS can help you eliminate risks incurred by hardcoding, plaintext configuration, and permission abuse.

Scenario

Application secrets are stored and can be accessed temporarily to prevent AK/SK leakage.

How It Works

You can configure an agency for elastic cloud server (ECS) on Identity and Access Management (IAM) to obtain the temporary access key (AK), thereby protecting the AK and secret key (SK).

Access secrets can be classified into permanent secrets and temporary secrets based on their validity periods. Permanent access secrets include usernames and passwords. Temporary access keys have a shorter validity period, are updated frequently, thus are more secure. You can assign an IAM agency to an ECS instance, so that applications in the ECS instance can use the temporary AK, SK, and security token to access CSMS. The temporary access keys are dynamically obtained every time they are required. They can also be cached in the memory and updated periodically.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Process Flow

Figure 2-2 ECS agency configuration process

Constraint

Only the administrator or an IAM user with the ECS permission can configure an agency for an ECS instance.

Procedure

Step 1 Create an ECS agency on IAM.

1. Log in to the **DEW console**.
2. Click ☰ on the left of the page and choose **Management & Governance** > **Identity and Access Management**.
3. In the navigation pane on the left, choose **Agencies**.
4. Click **Create Agency** in the upper right corner.
5. Configure the parameters on the displayed page. For details about the parameters, see **Table 2-1**.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 2-3 Creating an agency

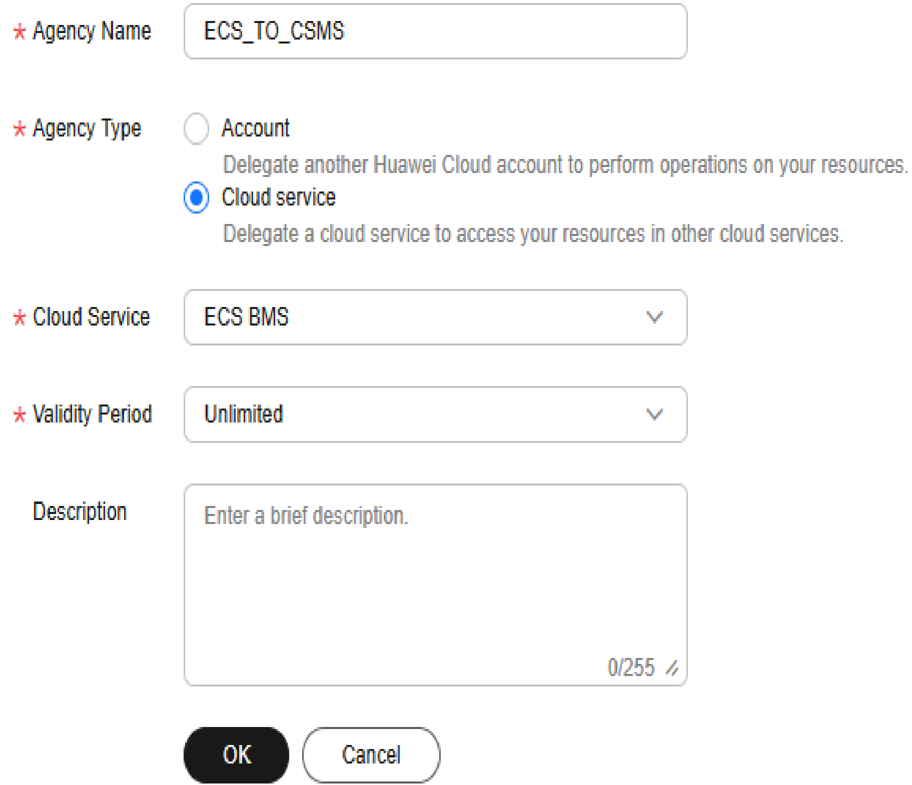


Table 2-1 Agency parameters

Parameter Name	Description
Agency Name	Enter an agency name, for example, **ECS_TO_CSMS**.
Agency Type	Select **Cloud service**.
Cloud Service	Select **ECS BMS**.
Validity Period	Select a duration. The value can be **Unlimited**, **1 day**, or **Custom**.
Description	(Optional) Enter agency description.

6. Click **OK**. In the displayed dialog box, click **Authorize**.
7. Click **Create Policy** in the upper right corner. If you already have a policy, skip this step.
   a. Configure policy parameters. For details about the parameters, see **Table 2-2**.

Copyright © Huawei Cloud Computing Technologies Co., Ltd.

Figure 2-4 Creating a policy

Table 2-2 Policy parameters

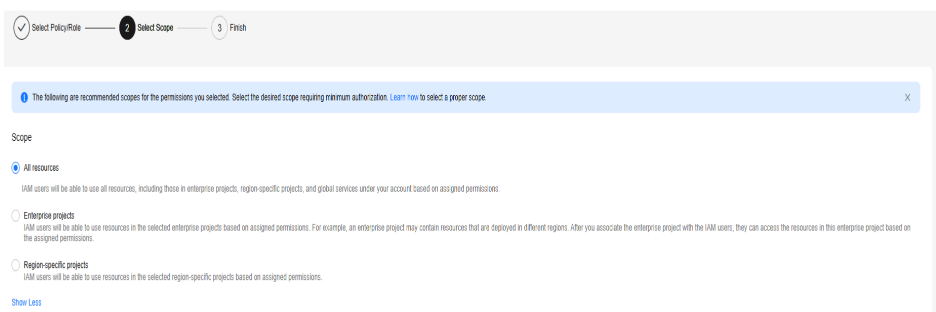
Parameter Name	Description
Policy Name	Enter a policy name.
Policy View	Select **Visual editor**.
Policy Content	■ **Allow**: Select **Allow**. ■ **Select service**: Select **Cloud Secret Management Service (CSMS)**. NOTE ○ If only the CSMS service permission is added, the KMS API may fail to be called. ○ You need to add policies for services one by one. After the policies are configured for a service, click **Add Permissions** to add policies for other services. ■ **Select action**: Select read and write permissions as required. ■ **(Optional) Select resource**: Select the scope of resources. ○ **Specific**: Access specific secrets. NOTE You can select **Specify resource path**, and then click **Add Resource Path** to specify an accessible secret. ○ **All**: Access all secrets. ■ **(Optional) Add request condition**: Click **Add Request Condition**, select a condition key and an operator, and enter values as required.
Description	(Optional) Enter policy description.

8. Click **Next** and select a policy for the agency. Click **Next**.
9. Select an authorization scope. You are advised to select **All resources**.
   – **All resources**: IAM users will be able to use all resources, including those in enterprise projects, region-specific projects, and global services under your account based on assigned permissions.

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

- **Enterprise projects**: The selected permissions will be applied to resources in the enterprise projects you select.
- **Region-specific projects**: The selected permissions will be applied to resources in the region-specific projects you select.

Figure 2-5 Selecting a scope

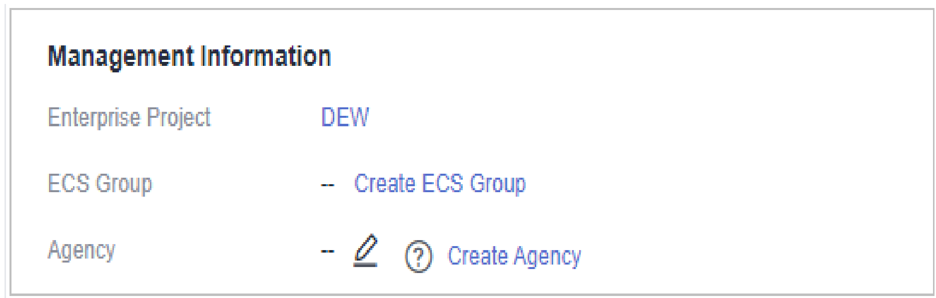


10. Click **OK**. Confirm the information and click **Finish**.

Step 2 Assign an agency (for example, **ECS_TO_CSMS**) to an ECS instance.

- If no ECS has been created, create an agency, for example, **ECS_TO_CSMS** for **Agency** under **Advanced Settings**. For details, see **Purchasing an ECS**.
- To use an existing ECS instance, perform the following steps:
  a. Click ☰ on the left and choose **Computing** > **Elastic Cloud Server**.
  b. Click the name of an ECS instance to go to the **Summary** page.
  c. In the **Management Information** area, click ✎ and select an agency (for example, **ECS_TO_CSMS**).

Figure 2-6 Selecting an agency



Step 3 In an application running on the ECS instance, call an API to obtain the temporary agency secrets, including the temporary AK, SK, and security token, to access CSMS.

1. Obtain the temporary AK and SK (in the **Security Key** directory). For details, see **Obtaining Metadata**.
   - URI
     http://169.254.169.254/openstack/latest/securitykey
   - Method
     GET request

– The following data is returned:

```
{
     "credential":{
        "access": "LDHZK30XXXXXXXXXXXXXV",
        "secret":"gyqcdzVXXXXXXXXXXXXXXXXXXXXXXXMl6",
        "securitytoken": "El9FI2C65qXXXXXXXXXXXXXXXXXXXXXXnkcaoV",
        "expires_at": "2022-07-14T12:09:24.147000Z"
             }
}
```

NOTE

- Extract the values of **access**, **secret**, and **securitytoken** to access CSMS.
- ECS automatically rotates temporary secrets to ensure that they are secure and valid.

2. Use the temporary AK/SK and security token to access CSMS.

```
package com.huaweicloud.sdk.test;

import com.huaweicloud.sdk.core.auth.ICredential;
import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.core.exception.ConnectionException;
import com.huaweicloud.sdk.core.exception.RequestTimeoutException;
import com.huaweicloud.sdk.core.exception.ServiceResponseException;
import com.huaweicloud.sdk.csms.v1.region.CsmsRegion;
import com.huaweicloud.sdk.csms.v1.*;
import com.huaweicloud.sdk.csms.v1.model.*;

public class ListSecretsSolution {
    public static void main(String[] args) {
        String ak = "<access>";
        String sk = "<secret>";
        String securitytoken = "<securitytoken>";
        ICredential auth = new BasicCredentials()
                .withAk(ak)
                .withSk(sk)
                .withSecurityToken(securitytoken);
         CsmsClient client = CsmsClient.newBuilder()
                 .withCredential(auth)
                 .withRegion(CsmsRegion.valueOf("cn-north-1"))
                 .build();
         ListSecretsRequest request = new ListSecretsRequest();
         try {
             ListSecretsResponse response = client.listSecrets(request);
             System.out.println(response.toString());
         } catch (ConnectionException e) {
             e.getMessage();
         } catch (RequestTimeoutException e) {
             e.getMessage();
         } catch (ServiceResponseException e) {
             e.getMessage();
             System.out.println(e.getHttpStatusCode());
             System.out.println(e.getErrorCode());
             System.out.println(e.getErrorMsg());
         }
     }
}
```

----End

2.3 Services Using CSMS

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

2.3.1 CCE Servers Using CSMS

Overview

CCE provides multiple types of plug-ins to extend cluster functions. The dew-provider plug-in of CCE interconnects with CSMS and mounts secrets to service pods. In this way, sensitive information is decoupled from the cluster environment, preventing sensitive information leakage caused by hard coding or plaintext configuration.

Constraints

- Supported cluster versions: v1.19 and later
- Supported cluster types: CCE Standard and CCE Turbo

Components

Table 2-3 dew-provider components

Component	Description	Resource Type
dew-provider	A component that obtains specified secrets from CSMS and mounts them to the pods.	Daemon Set
secrets-store-csi-driver	A component responsible for maintaining two CRDs: SecretProviderClass (SPC) and SecretProviderClassPodStatus (spcPodStatus). **SPC** is used to describe the secret that users are interested in (such as the secret version and name). It is created by users and will be referenced in pods. **spcPodStatus** is used to trace the binding relationships between pods and secrets. It is automatically created by csi-driver and requires no manual operation. One pod corresponds to one **spcPodStatus**. After a pod is started, a **spcPodStatus** is generated for the pod. When the pod lifecycle ends, the **spcPodStatus** is deleted accordingly.	Daemon Set

Installing the Plug-in On the Console

Step 1 Log in to the CCE console. Click the cluster name to access its details page. In the navigation pane on the left, choose **Add-ons**. Locate dew-provider on the right and click **Install**.

Step 2 On the **Install Add-on** page, configure parameters as required. **Table 2-4** describes the parameters.

Table 2-4 Parameters

Parameter	Description
rotation_poll_interval	Rotation interval, in unit of minutes (m, not min). The rotation interval indicates the interval for sending a request to CSMS and obtaining the latest secret. The proper interval range is [1m, 1440m]. The default value is **2m**.

Step 3 Click **Install**. After the plug-in is installed, select the cluster and click **Add-ons** from the navigation pane. On the displayed page, view the plug-in in the **Add-ons Installed** area.

Step 4 The plug-in can be used only if the secret created in DEW is used. Otherwise, the pod cannot run. For details about how to create a secret, see **Creating a Shared Secret**.

Step 5 Use the plug-in after it is installed. For details, see **CCE Secrets Manager for DEW**.

----End

2.3.2 Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials

When DLI writes the output data of Flink jobs to MySQL or DWS, you need to set sensitive parameters such as the username and password in the connector. Storing this information in plaintext poses significant security risks. To safeguard user data privacy, you are advised to encrypt these credentials.

Data Encryption Workshop (DEW) and Cloud Secret Management Service (CSMS) offer a secure, reliable, and easy-to-use solution for encrypting and decrypting sensitive data. By hosting database account details (for example, username and password) as managed secrets within CSMS, you can securely reference these credentials in your Flink jobs. This ensures that sensitive information is retrieved through a secure channel during runtime.

Additionally, CSMS offers comprehensive lifecycle management for credentials, enhancing both security and efficiency. It effectively mitigates risks associated with hardcoding sensitive information or storing it in plaintext configurations, thereby preventing unauthorized access and potential business disruptions.

This section walks you through on how to use DEW to manage access credentials for Flink OpenSource SQL jobs.

Constraints

DEW can be used to manage access credentials only in Flink 1.15. When creating a Flink job, select version 1.15 and configure the information of the agency that allows DLI to access DEW for the job.

Prerequisites

- A shared secret has been created on the DEW console and the secret value has been stored. For details, see **Creating a Secret**.
- An agency has been created and authorized for DLI to access DEW. The agency must have been granted the following permissions:
  - Permission of the **ShowSecretVersion** interface for querying secret versions and secret values in DEW: **csms:secretVersion:get**.
  - Permission of the **ListSecretVersions** interface for listing secret versions in DEW: **csms:secretVersion:list**.
  - Permission to decrypt DEW secrets: **kms:dek:decrypt**

  For details about agency permission examples, see **Creating a Custom DLI Agency** and **Agency Permission Policies in Common Scenarios**.
- On the DLI management console, create an enhanced datasource connection and configure the network connection between DLI and the data source.

  For details, see **Overview of Enhanced Datasource Connections**.

Syntax Format

```
create table tableName(
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression) )
 with (
   ...
 'dew.endpoint'='',
 'dew.csms.secretName'='',
 'dew.csms.decrypt.fields'='',
 'dew.projectId'='',
 'dew.csms.version'=''
  );
```

Parameters

Table 2-5 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
dew.endpoint	Yes	None	String	Endpoint of the DEW service to be used. Example: **'dew.endpoint'='kms.cn-xxxx.myhuaweicloud.com'**
dew.projectId	No	Yes	String	ID of the project DEW belongs to. The default value is the ID of the project where the Flink job is. **Obtain the project ID**.

Copyright © Huawei Cloud Computing Technologies Co., Ltd.

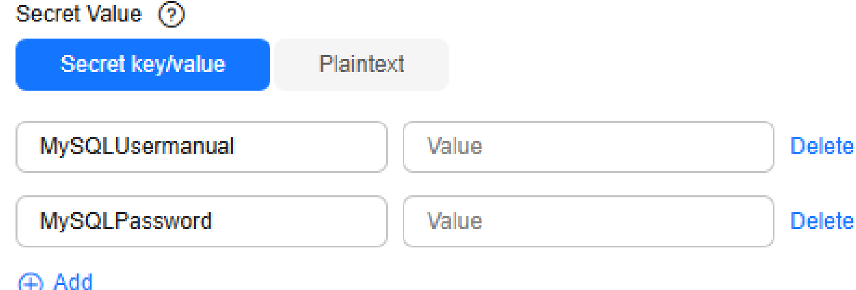
Parameter	Mandatory	Default Value	Data Type	Description
dew.csms.secretName	Yes	None	String	Name of the shared secret created in DEW CSMS. Example: **'dew.csms.secretName'='***secretInfo***'**
dew.csms.decrypt.fields	Yes	None	String	Specify which fields in the **connector with** attribute need to be decrypted using DEW CSMS. Separate the field attributes with commas, for example, **'dew.csms.decrypt.fields'='field1,field2,field3'**
dew.csms.version	No	Latest version	String	Version number (certificate version identifier) of the shared secret in DEW CSMS. If not specified, the latest version of the shared secret is obtained by default. Example: **'dew.csms.version'='v1'**

Example

This example demonstrates how to configure Flink OpenSource SQL to manage access credentials using DEW by generating random data through a DataGen table and outputting it to a MySQL result table.

1. Create a shared secret on DEW.
    a. Log in to the DEW console.
    b. In the navigation pane on the left, choose **Cloud Secret Management Service** > **Secrets**.
    c. Click **Create Secret**. On the displayed page, configure basic secret information.
        - **Secret Name**: Enter a secret name. In this example, the name is **secretInfo**.
        - **Secret Value**: Enter the username and password for logging in to the RDS for MySQL DB instance.
            - The key in the first line is **MySQLUsername**, and the value is the username for logging in to the DB instance.
            - The key in the second line is **MySQLPassword**, and the value is the password for logging in to the DB instance.

Figure 2-7 Setting the secret values



d. Set other parameters as required and click **OK**.

For details, see **Creating a Secret**.

2. Configure DEW to manage access credentials in the Flink OpenSource SQL job.

Ensure that an enhanced datasource connection between DLI and MySQL has been created. For details, see **Creating an Enhanced Datasource Connection**.

Ensure that an agency has been created for DLI to access DEW and authorization has been completed. For details, see **Creating a Custom DLI Agency**.

The following shows an example configuration for a Flink OpenSource SQL job:

```
create table dataGenSource(
   user_id string,
   amount int) with (
   'connector' = 'datagen',
   'rows-per-second' = '1', --Generate a piece of data per second.
  'fields.user_id.kind' = 'random', --Specify a random generator for the user_id field.
  'fields.user_id.length' = '3' --Limit the length of user_id to 3.
);
  CREATE TABLE jdbcSink (
    user_id string,
   amount int
 )
WITH (
   'connector' = 'jdbc',
   'url? = 'jdbc:mysql://MySQLAddress:MySQLPort/flink',--flink is the MySQL database where the
orders table locates.
   'table-name' = 'orders',
   'username' = 'MySQLUsername',  -- Shared secret in DEW whose name is secretInfo and version is
v1. The key MySQLUsername defines the secret value. The value is the user's sensitive information.
 'password' = 'MySQLPassword',  -- Shared secret in DEW whose name is secretInfo and version is
v1. The key MySQLPassword defines the secret value. The value is the user's sensitive information.
  'sink.buffer-flush.max-rows' = '1',
 'dew.endpoint'='endpoint', --Endpoint information for the DEW service being used
  'dew.csms.secretName'='secretInfo', --Name of the DEW shared secret
  'dew.csms.decrypt.fields'='username,password', --The password field value must be decrypted and
replaced using DEW CSMS.
  'dew.csms.version'='v1'
);
  insert into jdbcSink select * from dataGenSOurce;
```

3 General

3.1 Retrying Failed DEW Requests by Using Exponential Backoff

Scenario

If you receive an error message when calling an API, you can use exponential backoff to retry the request.

NOTICE

When interconnecting with KMS, retry is required. Error code such as 504, 502, 500, and 429 are included. Retry three to five times. For error codes 502 and 504, the timeout interval should be 5 to 8 seconds. Do not configure a long timeout interval. Otherwise, the client cannot respond.

How It Works

If consecutive errors (such as traffic limiting errors) are reported by the service side, continuous access will keep causing conflicts. Exponential backoff can help you avoid such errors.

Constraints

The current account has an enabled key.

Example

1. Prepare basic authentication information.
   - **ACCESS_KEY**: Access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**
   - **SECRET_ACCESS_KEY**: Secret access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

- **PROJECT_ID**: site project ID. For details, see **Obtaining a Project ID**.
- **KMS_ENDPOINT**: endpoint for accessing KMS.
- There will be security risks if the AK/SK used for authentication is directly written into code. Encrypt the AK/SK in the configuration file or environment variables for storage.
- In this example, the AK/SK stored in the environment variables are used for identity authentication. Configure the environment variables **HUAWEICLOUD_SDK_AK** and **HUAWEICLOUD_SDK_SK** in the local environment first.

2. Code for exponential backoff:

```
import com.huaweicloud.sdk.core.auth.BasicCredentials;
 import com.huaweicloud.sdk.core.auth.ICredential;
 import com.huaweicloud.sdk.core.exception.ClientRequestException;
 import com.huaweicloud.sdk.kms.v2.model.EncryptDataRequest;
 import com.huaweicloud.sdk.kms.v2.model.EncryptDataRequestBody;
 import com.huaweicloud.sdk.kms.v2.KmsClient;

 public class KmsEncryptExample {

     private static final String ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_AK");

     private static final String SECRET_ACCESS_KEY = System.getenv("HUAWEICLOUD_SDK_SK");

     private static final String KMS_ENDPOINT = "xxxx";

     private static final String KEY_ID = "xxxx";

     private static final String PROJECT_ID = "xxxx";


     private static KmsClient KmsClientInit() {
         ICredential auth = new BasicCredentials()
                 .withAk(ACCESS_KEY)
                 .withSk(SECRET_ACCESS_KEY)
                 .withProjectId(PROJECT_ID);
         return KmsClient.newBuilder()
                 .withCredential(auth)
                 .withEndpoint(KMS_ENDPOINT)
                 .build();
     }

     public static long getWaitTime(int retryCount) {
         long initialDelay = 200L;
         return (long) (Math.pow(2, retryCount) * initialDelay);
     }

     public static void encryptData(KmsClient client, String plaintext) {
         EncryptDataRequest request = new EncryptDataRequest().withBody(
                 new EncryptDataRequestBody()
                         .withKeyId(KEY_ID)
                         .withPlainText(plaintext));
         client.encryptData(request);
     }

     public static void main(String[] args) {
         int maxRetryTimes = 6;
         String plaintext = "plaintext";
         String errorMsg = "The throttling threshold has been reached";

         KmsClient client = KmsClientInit();
         for (int i = 0; i < maxRetryTimes; i++) {
             try {
                 encryptData(client, plaintext);
                 return;
             } catch (ClientRequestException e) {
```

 Copyright © Huawei Cloud Computing Technologies Co., Ltd.

```
                if (e.getErrorMsg().contains(errorMsg)) {
                    try {
                        Thread.sleep(getWaitTime(i));
                    } catch (InterruptedException ex) {
                        throw new RuntimeException(ex);
                    }
                }
            }
        }
    }
}
```