Data Encryption Workshop

Best Practices

Issue	10
Date	2025-09-15

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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 Encrypting or Decrypting a Small Amount of Data 1
1.1.2 Encrypting or Decrypting a Large Amount of Data 4
1.2 Using KMS to Encrypt and Decrypt Data for Cloud Services 12
1.2.1 Overview 12
1.2.2 Encrypting Data in ECS 14
1.2.3 Encrypting Data in EVS 15
1.2.4 Encrypting Data in IMS 19
1.2.5 Encrypting Data in OBS 22
1.2.6 Encrypting an RDS DB Instance 29
1.2.7 Encrypting a DDS DB Instance 29
1.3 Using the Encryption SDK to Encrypt and Decrypt Local Files 30
1.4 Encrypting and Decrypting Data Through Cross-region DR 33
1.5 Using KMS to Protect File Integrity 36
2 Cloud Secret Management Service 40
2.1 Using CSMS to Change Hard-coded Database Account Passwords 40
2.2 Using CSMS to Prevent AK/SK Leakage 44
2.3 Services Using CSMS 50
2.3.1 CCE Servers Using CSMS 50
3 General 52
3.1 Retrying Failed DEW Requests by Using Exponential Backoff 52

1 Key Management Service

1.1 Using KMS to Encrypt Offline Data

1.1.1 Encrypting or Decrypting a Small Amount of Data

Scenario

You can use online tools on the Key Management Service (KMS) console or call the necessary KMS APIs to directly encrypt or decrypt small-size data with a customer master key (CMK), such as passwords, certificates, or phone numbers.

Restrictions

Currently, a maximum of 4 KB of data can be encrypted or decrypted in this way.

Encryption and Decryption Using Online Tools

- **Encrypting data**

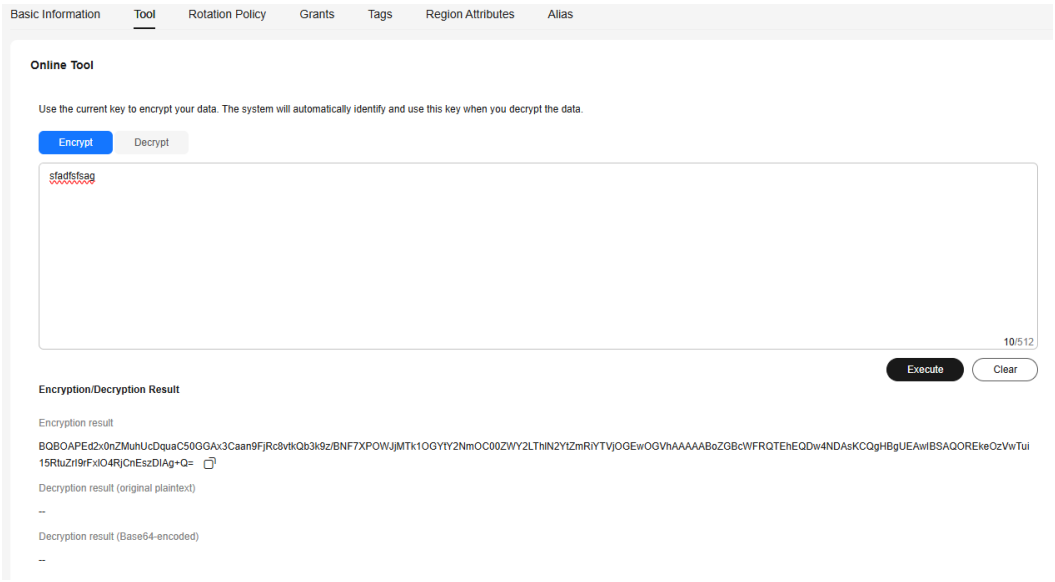
Step 1 Log in to the **DEW console**.

Step 2 Click in the upper left corner and select a region or project.

Step 3 Click the name of the target custom key to access the key details page. Click the **Tool** tab.

Step 4 Click **Encrypt**. In the text box on the left, enter the data to be encrypted, as shown in **Figure 1-1**.

Figure 1-1 Encrypting data



Step 5 Click **Execute**. The encrypted data is displayed in the **Encryption/Decryption Result** area.

NOTE

- Use the current CMK to encrypt the data.
- To clear your input, click **Clear**.
- In the **Encryption result** area, click to copy the encrypted data and save it to a local file.

----End

- **Decrypting data**

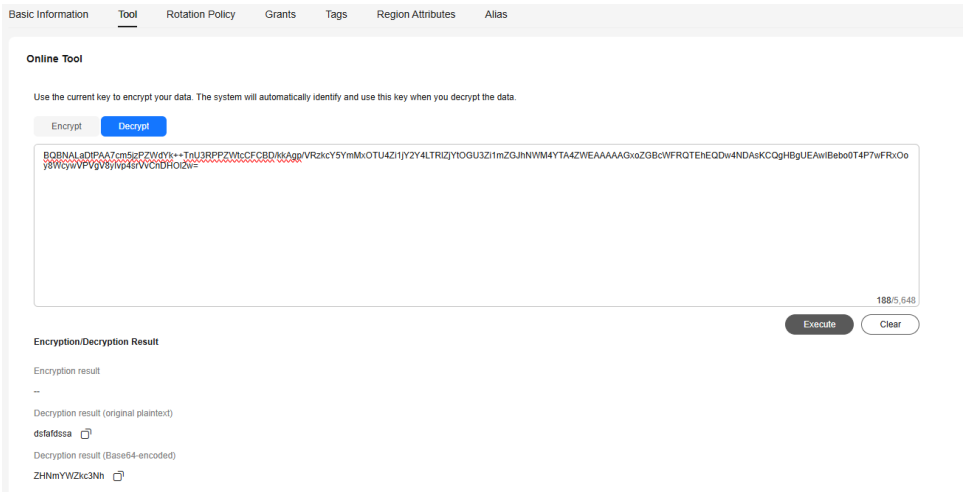
Step 1 Click a non-default key in the **Enabled** state and go to the **Tool** tab.

Step 2 Click **Decrypt** and enter the data to be decrypted in the text box, as shown in Figure 1-2.

NOTE

- The tool will identify the original encryption CMK and use it to decrypt the data.
- If the key has been deleted, the decryption will fail.

Figure 1-2 Decrypting data



Step 3 Click **Execute**. The decrypted data is displayed in the **Encryption/Decryption Result** area.

NOTE

- In the **Decryption result** area, click to copy the decrypted data and save it to a local file.
- The information to be encrypted using commands or APIs cannot contain special characters. Otherwise, the decryption result may fail to be displayed on the console.
- Enter the plaintext on the console, the text will be encoded to Base64 format before encryption.

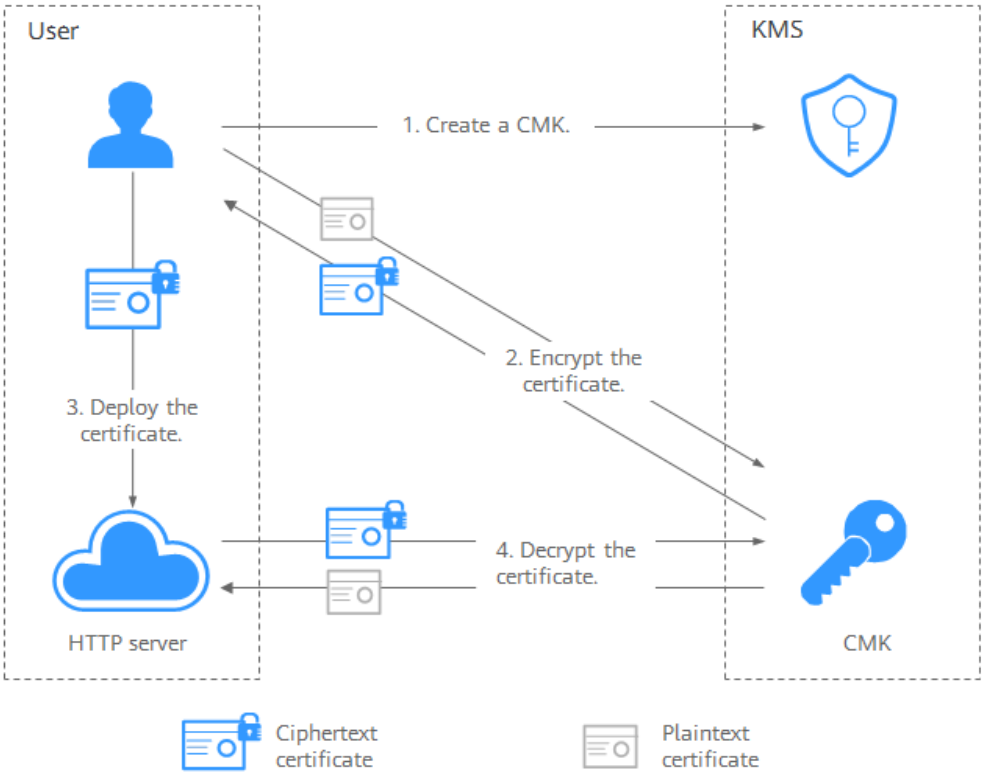
  The decryption result returned via API will be in Base64 format. Perform Base64 decoding to obtain the plaintext entered on the console.

----End

Calling APIs for Encryption and Decryption

Figure 1-3 shows an example about how to call KMS APIs to encrypt and decrypt an HTTPS certificate.

Figure 1-3 Encrypting and decrypting an HTTPS certificate



The procedure is as follows:

1. Create a CMK on KMS.
2. Call the KMS API for **encrypting a data key** and use the specified CMK to encrypt the plaintext certificate.
3. Deploy the certificate onto a server.
4. The server calls the KMS API for **decrypting a data key** and decrypts the ciphertext certificate.

   NOTE

   If you enter and encrypt text on the console, the text will be encoded to Base64 format before being transferred to the backend for encryption. The decryption result returned via API will be in Base64 format. Text encrypted via API cannot be decrypted on the console, or garbled characters will be returned.

1.1.2 Encrypting or Decrypting a Large Amount of Data

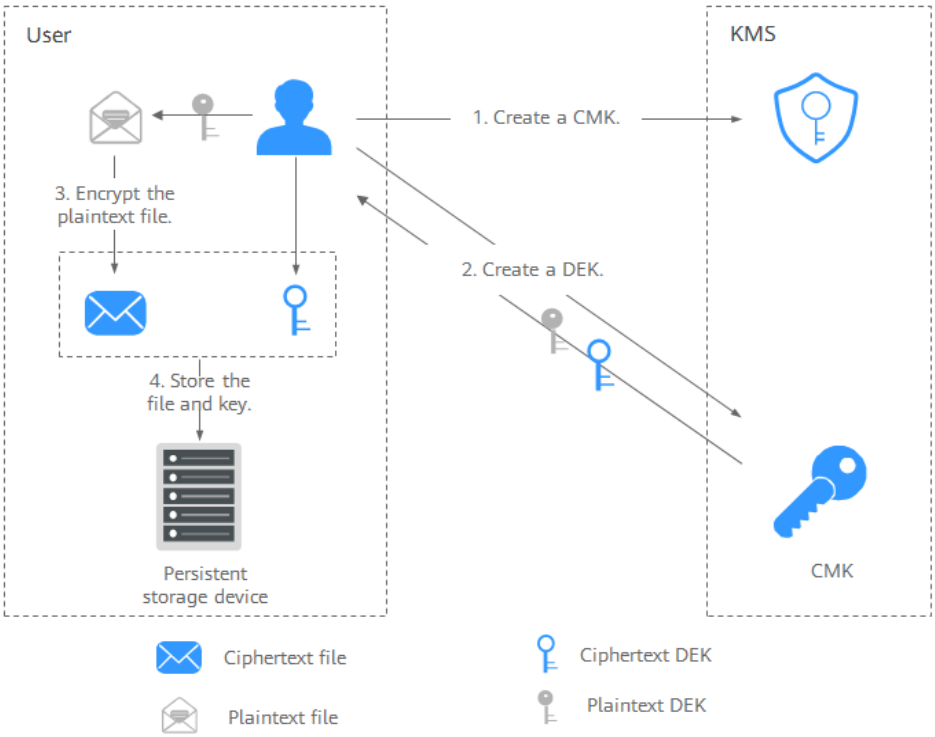
Scenario

If you want to encrypt or decrypt large volumes of data, such as pictures, videos, and database files, you can use envelope encryption, which allows you to encrypt and decrypt files without having to transfer a large amount of data over the network.

Encryption and Decryption Processes

- Large-size data encryption

  Figure 1-4 Encrypting a local file



  The process is as follows:

  a. Create a CMK on KMS.

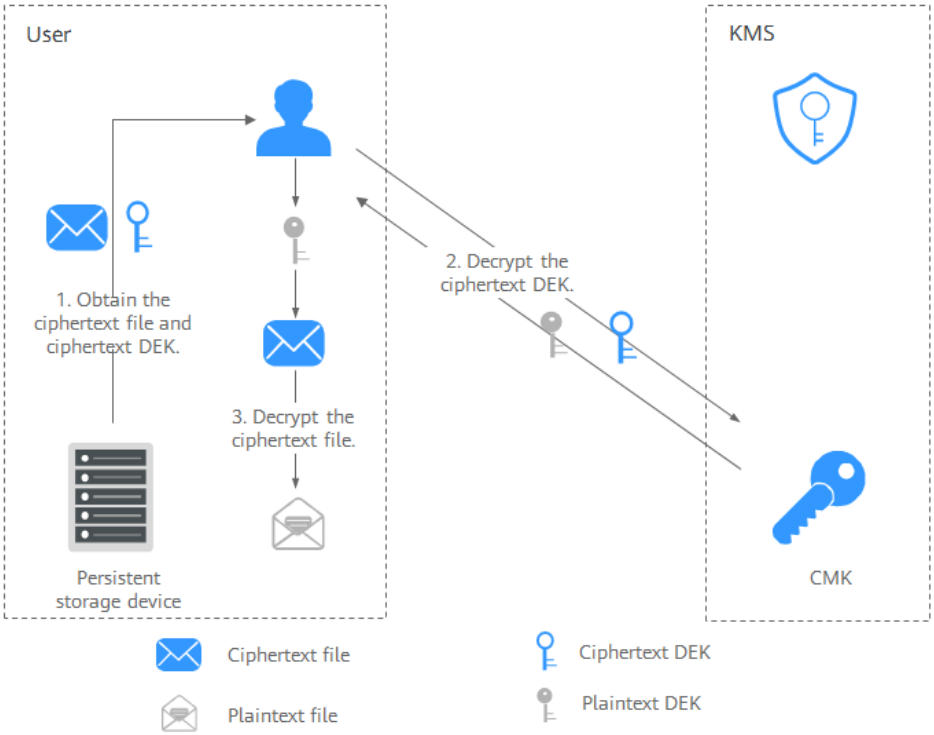
  b. Call the **create-datakey** API of KMS to create a DEK. A plaintext DEK and a ciphertext DEK will be generated. The ciphertext DEK is generated when you use a CMK to encrypt the plaintext DEK.

  c. Use the plaintext DEK to encrypt a plaintext file, generating a ciphertext file.

  d. Store the ciphertext DEK and the ciphertext file together in a permanent storage device or a storage service.

- Large-size data decryption

Figure 1-5 Decrypting a local file



The process is as follows:

a. Read the ciphertext DEK and the ciphertext file from the permanent storage device or 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 will fail. Properly keep your CMKs.

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

APIs Related to Envelope Encryption

You can use the following APIs 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.

Encrypting a Local File

1. Create a CMK on the management console. For details, see **Creating a Key**.
2. Prepare basic authentication information.
   - **ACCESS_KEY**: access key of the Huawei ID
   - **SECRET_ACCESS_KEY**: secret access key of the Huawei ID
   - **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.
3. Encrypt a local file.

   The example code is shown below.
   - CMK is the ID of the key created on the Huawei Cloud management console.
   - The plaintext data file is **FirstPlainFile.jpg**.
   - The data file generated after encryption is **SecondEncryptFile.jpg**.

```
import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.kms.v1.KmsClient;
import com.huaweicloud.sdk.kms.v1.model.CreateDatakeyRequest;
import com.huaweicloud.sdk.kms.v1.model.CreateDatakeyRequestBody;
import com.huaweicloud.sdk.kms.v1.model.CreateDatakeyResponse;
import com.huaweicloud.sdk.kms.v1.model.DecryptDatakeyRequest;
import com.huaweicloud.sdk.kms.v1.model.DecryptDatakeyRequestBody;

import javax.crypto.Cipher;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.security.SecureRandom;

/**
* Use a DEK to encrypt and decrypt files.
* To enable the assert syntax, add -ea to enable VM_OPTIONS.
 */
public class FileStreamEncryptionExample {

    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 KMS_ENDPOINT = "<KmsEndpoint>";


    /**
     * AES algorithm flags:
     * - AES_KEY_BIT_LENGTH: bit length of the AES256 key
```



```
     * - AES_KEY_BYTE_LENGTH: byte length of the AES256 key
     * - AES_ALG: AES256 algorithm. In this example, the Group mode is GCM and the padding mode is PKCS5Padding.
     * - AES_FLAG: AES algorithm flag
     * - GCM_TAG_LENGTH: GCM tag length
     * - GCM_IV_LENGTH: length of the GCM initial vector
     */
    private static final String AES_KEY_BIT_LENGTH = "256";
    private static final String AES_KEY_BYTE_LENGTH = "32";
    private static final String AES_ALG = "AES/GCM/PKCS5Padding";
    private static final String AES_FLAG = "AES";
    private static final int GCM_TAG_LENGTH = 16;
    private static final int GCM_IV_LENGTH = 12;

    public static void main(final String[] args) {
        // ID of the CMK you created on the Huawei Cloud management console
        final String keyId = args[0];

        encryptFile(keyId);
    }

    /**
     * Using a DEK to encrypt and decrypt a file
     *
     * @param keyId: user CMK ID
     */
    static void encryptFile(String keyId) {
        // 1. Prepare the authentication information for accessing Huawei Cloud.
        final BasicCredentials auth = new BasicCredentials().withAk(ACCESS_KEY).withSk(SECRET_ACCESS_KEY)
                .withProjectId(PROJECT_ID);

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

        // 3. Assemble the request message for creating a DEK.
        final CreateDatakeyRequest createDatakeyRequest = new CreateDatakeyRequest()
                .withBody(new CreateDatakeyRequestBody().withKeyId(keyId).withDatakeyLength(AES_KEY_BIT_LENGTH));

        // 4. Create a DEK.
        final CreateDatakeyResponse createDatakeyResponse = kmsClient.createDatakey(createDatakeyRequest);

        // 5. Receive the created DEK information.
        // It is recommended that the ciphertext key and key ID be stored locally so that the plaintext key can be easily obtained for data decryption.
        // The plaintext key should be used immediately after being created. Before using it, convert the hexadecimal plaintext key to a byte array.
        final String cipherText = createDatakeyResponse.getCipherText();
        final byte[] plainKey = hexToBytes(createDatakeyResponse.getPlainText());

        // 6. Prepare the file to be encrypted.
        // inFile: file to be encrypted
        // outEncryptFile: file generated after encryption

        final File inFile = new File("FirstPlainFile.jpg");
        final File outEncryptFile = new File("SecondEncryptFile.jpg");

        // 7. If the AES algorithm is used for encryption, you can create an initial vector.
        final byte[] iv = new byte[GCM_IV_LENGTH];
        final SecureRandom secureRandom = new SecureRandom();
        secureRandom.nextBytes(iv);

        // 8. Encrypt the file and store the encrypted file.
        doFileFinal(Cipher.ENCRYPT_MODE, inFile, outEncryptFile, plainKey, iv);
```



```
    }

    /**
     * Encrypting and decrypting a file
     *
     * @param cipherMode: Encryption mode. It can be Cipher.ENCRYPT_MODE or
Cipher.DECRYPT_MODE.
     * @param infile: file to be encrypted or decrypted
     * @param outFile: file generated after encryption and decryption
     * @param keyPlain: plaintext key
     * @param iv: initial vector
     */
    static void doFileFinal(int cipherMode, File infile, File outFile, byte[] keyPlain, byte[] iv) {

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(infile));
             BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream(outFile))) {
            final byte[] bytIn = new byte[(int) infile.length()];
            final int fileLength = bis.read(bytIn);

            assert fileLength > 0;

            final SecretKeySpec secretKeySpec = new SecretKeySpec(keyPlain, AES_FLAG);
            final Cipher cipher = Cipher.getInstance(AES_ALG);
            final GCMParameterSpec gcmParameterSpec = new
GCMParameterSpec(GCM_TAG_LENGTH * Byte.SIZE, iv);
            cipher.init(cipherMode, secretKeySpec, gcmParameterSpec);
            final byte[] bytOut = cipher.doFinal(bytIn);
            bos.write(bytOut);
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage());
        }
    }

}
```

Decrypting a Local File

1. Prepare basic authentication information.
   - **ACCESS_KEY**: access key of the Huawei ID
   - **SECRET_ACCESS_KEY**: secret access key of the Huawei ID
   - **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.
2. Decrypt a local file.

   The example code is shown below.
   - CMK is the ID of the key created on the Huawei Cloud management console.
   - The data file generated after encryption is **SecondEncryptFile.jpg**.
   - The data file generated after encryption and decryption is **ThirdDecryptFile.jpg**.


```
import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.kms.v1.KmsClient;
import com.huaweicloud.sdk.kms.v1.model.CreateDatakeyRequest;
import com.huaweicloud.sdk.kms.v1.model.CreateDatakeyRequestBody;
import com.huaweicloud.sdk.kms.v1.model.CreateDatakeyResponse;
import com.huaweicloud.sdk.kms.v1.model.DecryptDatakeyRequest;
import com.huaweicloud.sdk.kms.v1.model.DecryptDatakeyRequestBody;

import javax.crypto.Cipher;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.security.SecureRandom;

/**
* Use a DEK to encrypt and decrypt files.
* To enable the assert syntax, add -ea to enable VM_OPTIONS.
 */
public class FileStreamEncryptionExample {

    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 KMS_ENDPOINT = "<KmsEndpoint>";

    /**
     * AES algorithm flags:
     * - AES_KEY_BIT_LENGTH: bit length of the AES256 key
      * - AES_KEY_BYTE_LENGTH: byte length of the AES256 key
     * - AES_ALG: AES256 algorithm. In this example, the Group mode is GCM and the padding
mode is PKCS5Padding.
     * - AES_FLAG: AES algorithm flag
     * - GCM_TAG_LENGTH: GCM tag length
     * - GCM_IV_LENGTH: length of the GCM initial vector
     */
    private static final String AES_KEY_BIT_LENGTH = "256";
    private static final String AES_KEY_BYTE_LENGTH = "32";
    private static final String AES_ALG = "AES/GCM/PKCS5Padding";
    private static final String AES_FLAG = "AES";
    private static final int GCM_TAG_LENGTH = 16;
    private static final int GCM_IV_LENGTH = 12;

    public static void main(final String[] args) {
        // ID of the CMK you created on the Huawei Cloud management console
        final String keyId = args[0];
        // // Returned ciphertext DEK after DEK creation
        final String cipherText = args[1];

        decryptFile(keyId, cipherText);
    }

    /**
     * Using a DEK to encrypt and decrypt a file
     *
     * @param keyId: user CMK ID
     * @param cipherText: ciphertext data key
     */
    static void decryptFile(String keyId, String cipherText) {
        // 1. Prepare the authentication information for accessing Huawei Cloud.
        final BasicCredentials auth = new
BasicCredentials().withAk(ACCESS_KEY).withSk(SECRET_ACCESS_KEY)
                .withProjectId(PROJECT_ID);
```



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

        // 3. Prepare the file to be encrypted.
        // inFile: file to be encrypted
        // outEncryptFile: file generated after encryption
        // outDecryptFile: file generated after encryption and decryption
        final File inFile = new File("FirstPlainFile.jpg");
        final File outEncryptFile = new File("SecondEncryptFile.jpg");
        final File outDecryptFile = new File("ThirdDecryptFile.jpg");

        // 4. Use the same initial vector for AES encryption and decryption.
        final byte[] iv = new byte[GCM_IV_LENGTH];

        // 5. Assemble the request message for decrypting the DEK. cipherText is the ciphertext
DEK returned after DEK creation.
        final DecryptDatakeyRequest decryptDatakeyRequest = new DecryptDatakeyRequest()
                .withBody(new DecryptDatakeyRequestBody()
                        .withKeyId(keyId).withCipherText(cipherText).withDatakeyCipherLength(AES_KEY
_BYTE_LENGTH));

        // 6. Decrypt the DEK and convert the returned hexadecimal plaintext key into a byte array.
        final byte[] decryptDataKey =
hexToBytes(kmsClient.decryptDatakey(decryptDatakeyRequest).getDataKey());

        // 7. Decrypt the file and store the decrypted file.
// iv at the end of the statement is the initial vector created in the encryption example.
        doFileFinal(Cipher.DECRYPT_MODE, outEncryptFile, outDecryptFile, decryptDataKey, iv);

        // 8. Compare the original file with the decrypted file.
        assert getFileSha256Sum(inFile).equals(getFileSha256Sum(outDecryptFile));

    }

    /**
     * Encrypting and decrypting a file
     *
     * @param cipherMode: Encryption mode. It can be Cipher.ENCRYPT_MODE or
Cipher.DECRYPT_MODE.
     * @param infile: file to be encrypted or decrypted
     * @param outFile: file generated after encryption and decryption
     * @param keyPlain: plaintext key
     * @param iv: initial vector
     */
    static void doFileFinal(int cipherMode, File infile, File outFile, byte[] keyPlain, byte[] iv) {

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(infile));
             BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream(outFile))) {
            final byte[] bytIn = new byte[(int) infile.length()];
            final int fileLength = bis.read(bytIn);

            assert fileLength > 0;

            final SecretKeySpec secretKeySpec = new SecretKeySpec(keyPlain, AES_FLAG);
            final Cipher cipher = Cipher.getInstance(AES_ALG);
            final GCMParameterSpec gcmParameterSpec = new
GCMParameterSpec(GCM_TAG_LENGTH * Byte.SIZE, iv);
            cipher.init(cipherMode, secretKeySpec, gcmParameterSpec);
            final byte[] bytOut = cipher.doFinal(bytIn);
            bos.write(bytOut);
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage());
        }
    }

    /**
```



```
* Converting a hexadecimal string to a byte array
     *
     * @param hexString: a hexadecimal string
     * @return: byte array
     */
    static byte[] hexToBytes(String hexString) {
        final int stringLength = hexString.length();
        assert stringLength > 0;
        final byte[] result = new byte[stringLength / 2];
        int j = 0;
        for (int i = 0; i < stringLength; i += 2) {
            result[j++] = (byte) Integer.parseInt(hexString.substring(i, i + 2), 16);
        }
        return result;
    }

    /**
     * Calculate the SHA256 digest of the file.
     *
     * @param file
     * @return SHA256 digest
     */
    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 new BigInteger(1, sha256.digest()).toString(16);
        } catch (IOException e) {
            throw new RuntimeException(e.getMessage());
        }
    }
}
```

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.

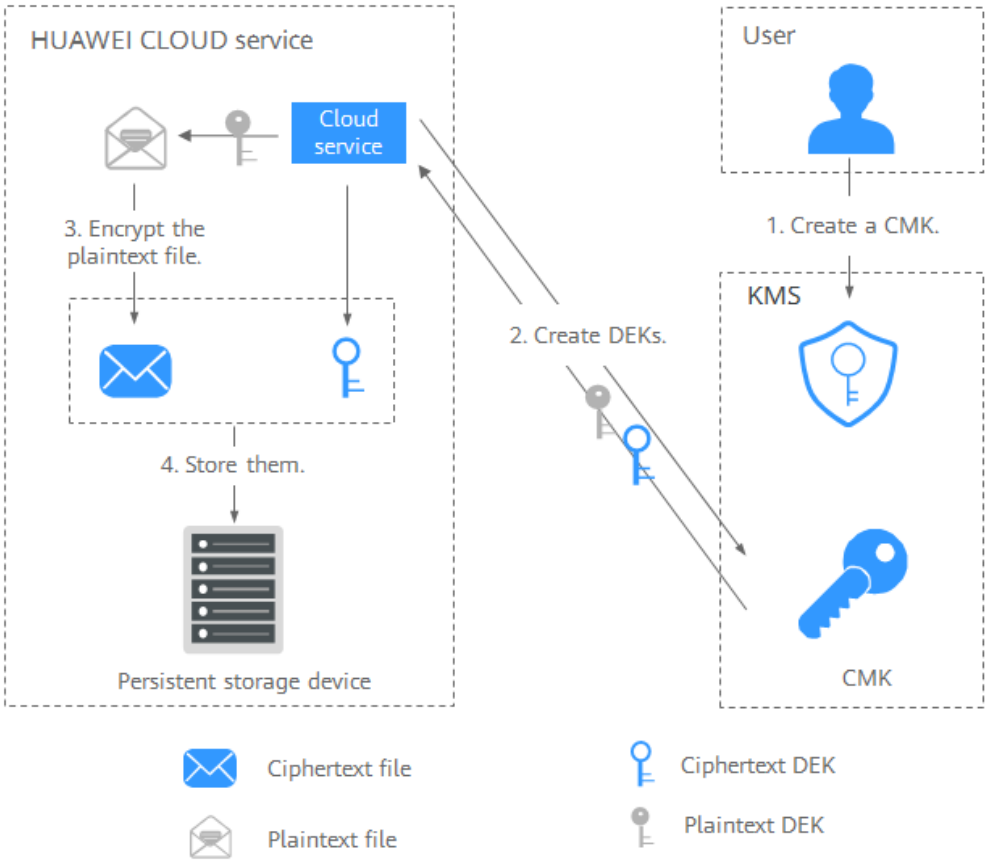
Table 1-1 Cloud services integrated with KMS

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**

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 1-6 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 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 1-2**.

Table 1-2 Impact on encrypted disks after a custom key becomes unavailable

Custom Key Status	Impact on Encrypted Disks	Restoration Method
Disabled	• 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.	Enable the CMK. For details, see **Enabling a Key**.
Scheduled deletion		Cancel the scheduled deletion for the CMK. For details, see **Canceling the Scheduled Deletion of One or More CMKs**.
Deleted		Data on the disks can never be restored.

Resource and Cost Planning

Table 1-3 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**.

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 1-7 More

2. Create an agency.

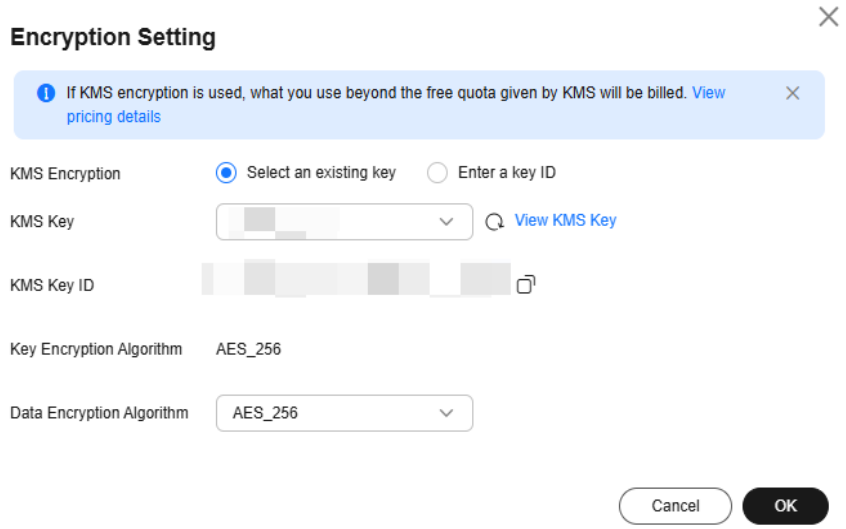
   Select **Encryption**. If EVS is not authorized to access KMS, the **Create Agency** dialog box is displayed. In this case, click **Yes** to authorize it. After the authorization, EVS can obtain KMS keys to encrypt and decrypt disks.

   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.

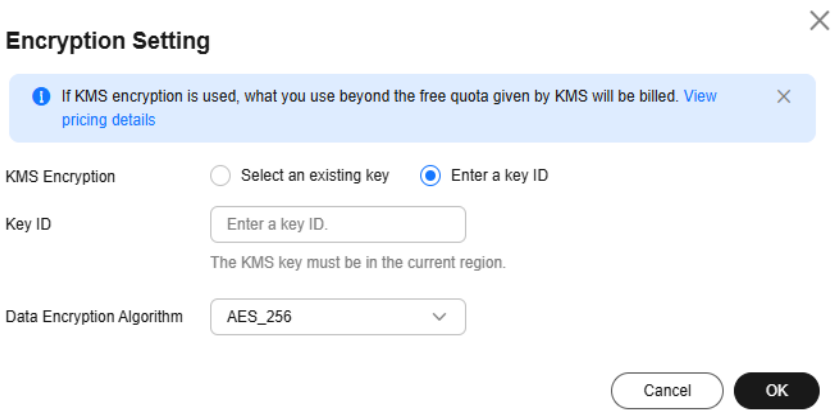
Figure 1-8 Encryption Settings dialog box



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 1-9 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:

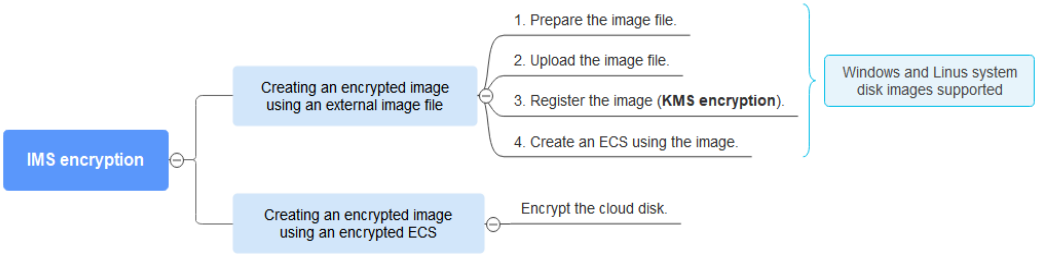
- Method 1: **Create an encrypted image using an external image file.**
  When you register an image file as a private image, select **KMS encryption** and select a key.
- Method 2: **Create an encrypted image using an encrypted ECS.**
  When you use an ECS to create a private image, if the system disk 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 1-10 describes how to use KMS to encrypt an IMS image file.

Figure 1-10 Encrypting IMS



Resource and Cost Planning

Table 1-4 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**.

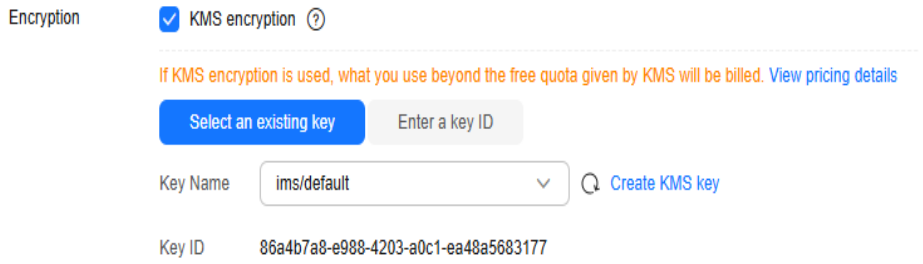
Step 2 Upload the external image file to the OBS bucket. For details, see **Creating a Windows System Disk Image from an External Image File**.

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**.

- **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 1-11 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.

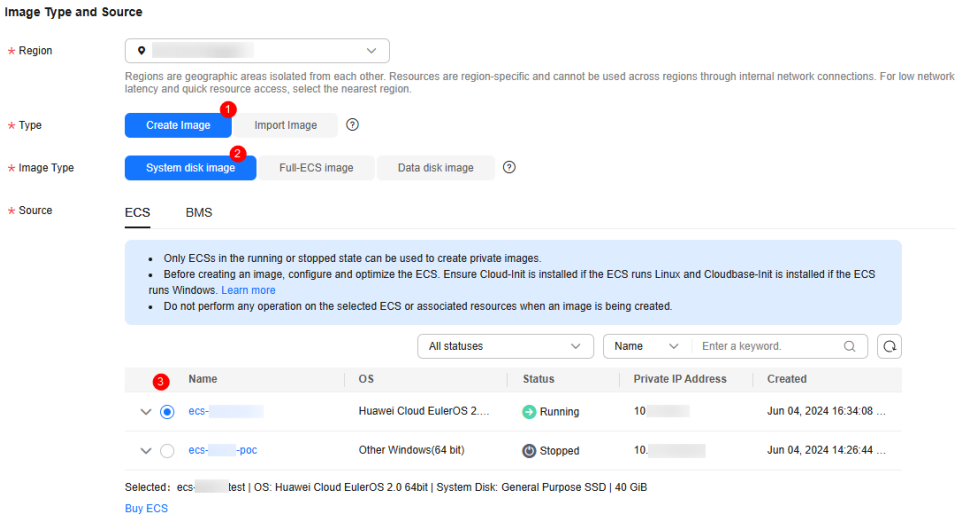
Step 1 Encrypt the EVS system disk. For details, see **Encrypting Data in EVS**.

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**.

Figure 1-12 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.

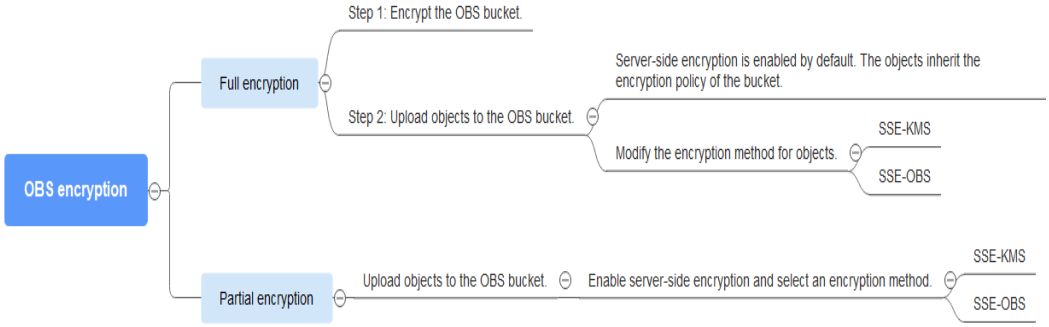
  In this case, you only need to encrypt the OBS bucket, as the objects in the bucket inherit the bucket encryption configurations by default. For details, see **Enabling Server-Side Encryption When Creating an OBS Bucket** or **Enabling Encryption for a Created OBS Bucket**.

  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**.

Figure 1-13 OBS encryption

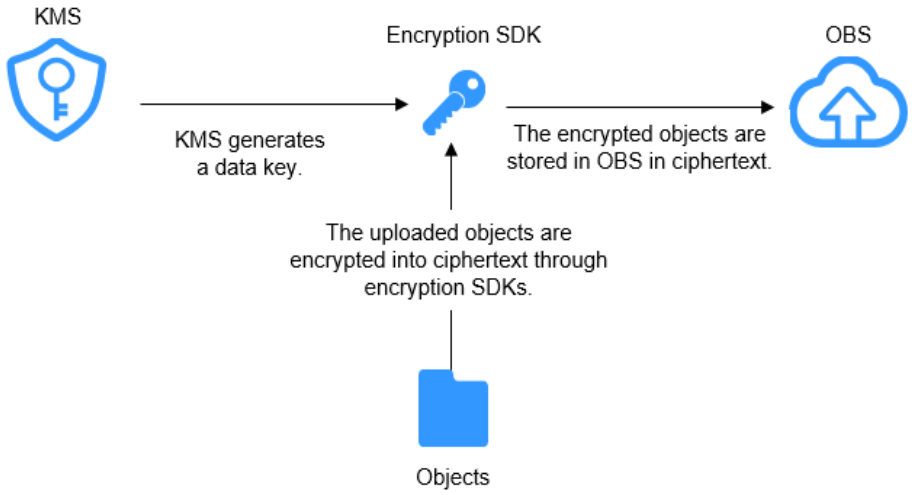


Solution Architecture

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

- Encryption principle

  Figure 1-14 Encryption principle



  a. Obtain the encryption key.

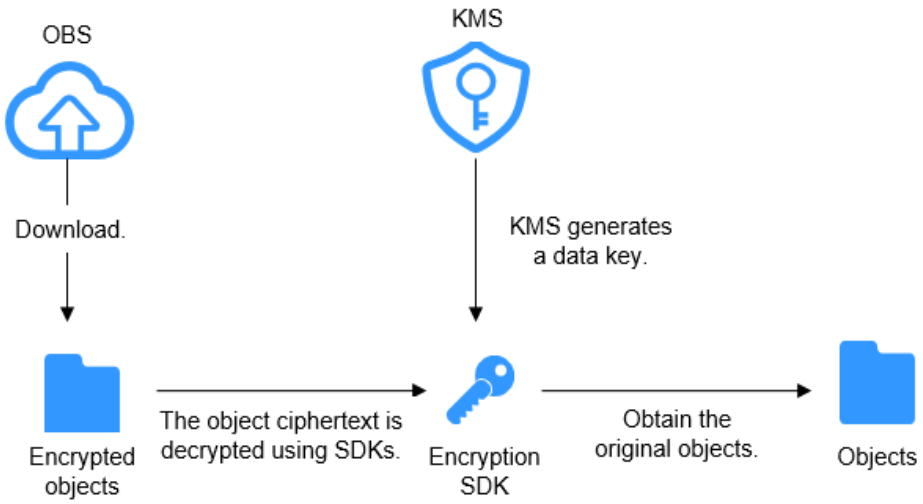
     Generate a data encryption key (DEK) on KMS to encrypt objects in an OBS bucket.

  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 1-15 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.

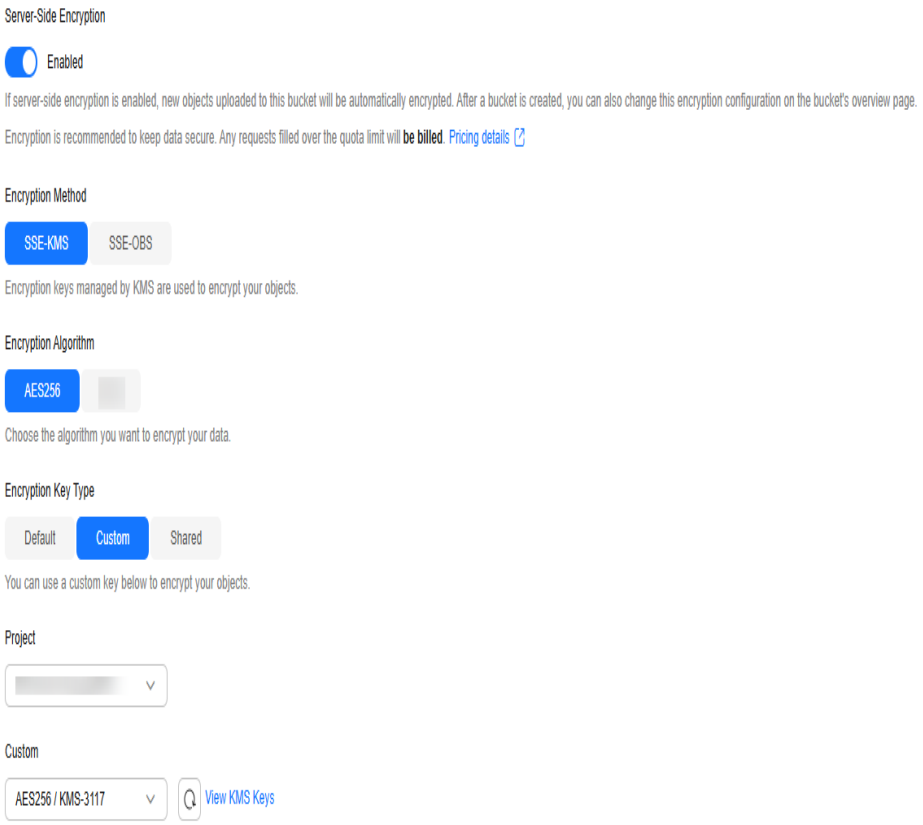
Enabling Server-Side Encryption When Creating an OBS Bucket

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 Metod**, and select an encryption key type.

Figure 1-16 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

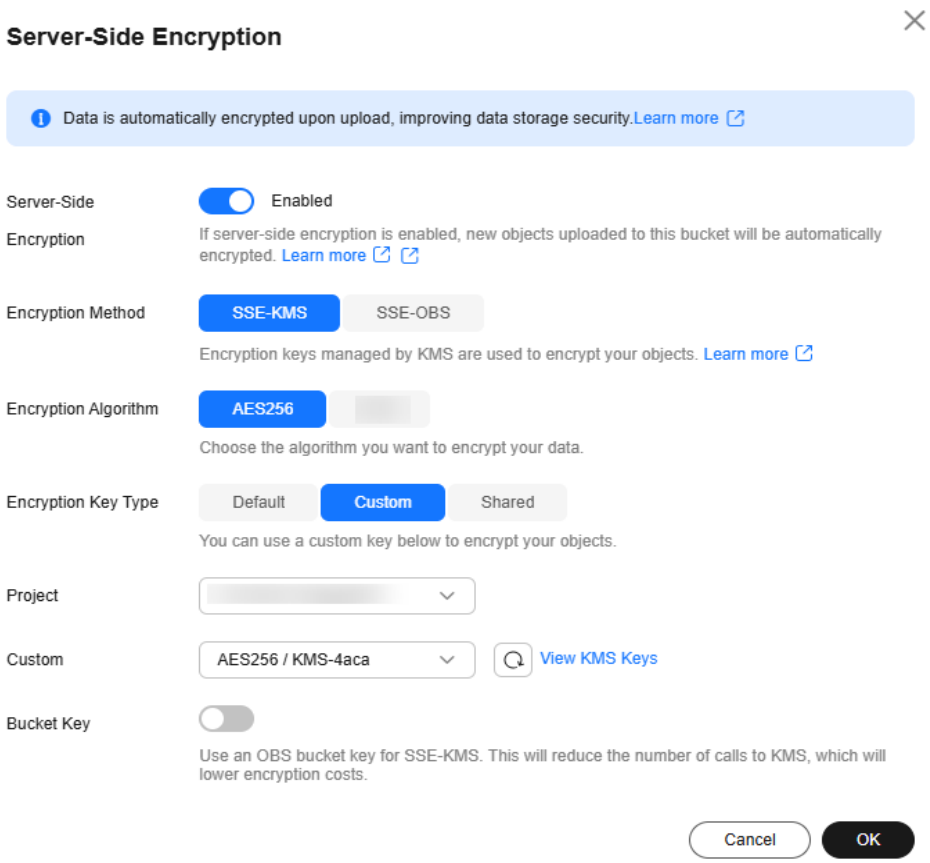
Step 1 In the navigation pane on the left, choose **Buckets**. Click the target bucket and access the **Objects** page.

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 1-17 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.

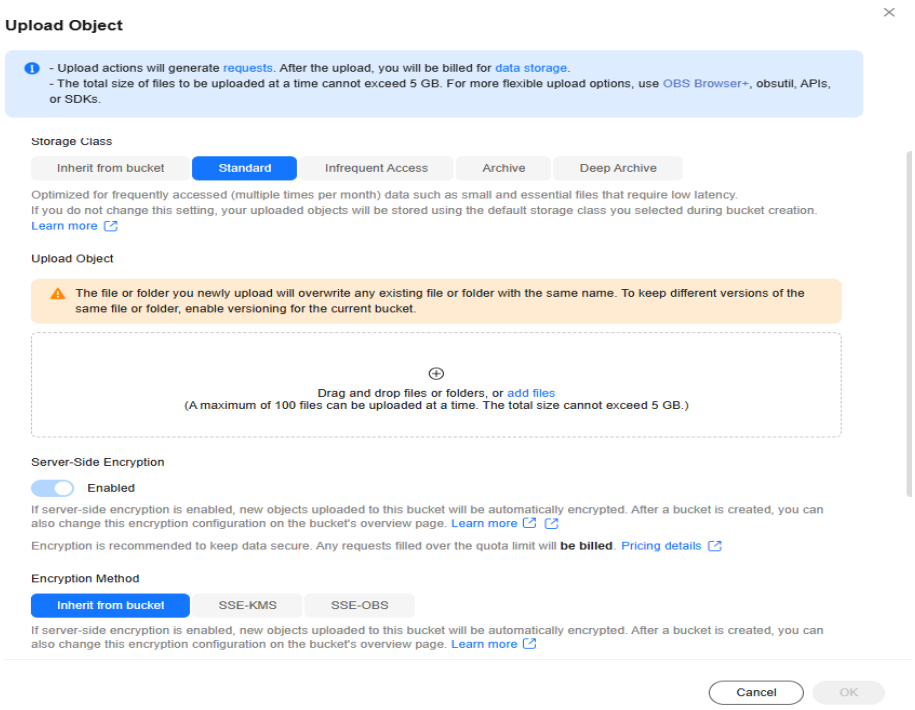
Step 2 In the navigation pane on the left, choose **Objects**.

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 1-18**.

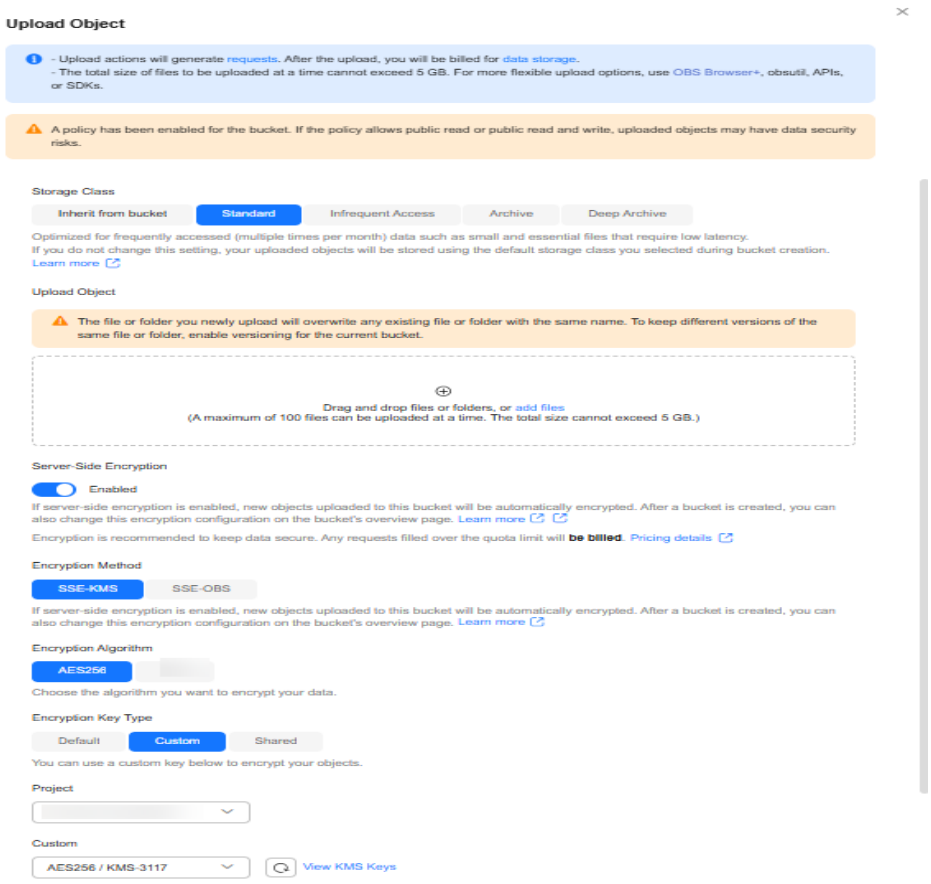
Figure 1-18 Uploading an object with server-side encryption enabled (OBS bucket encryption enabled)



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.

Figure 1-19 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**.

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.

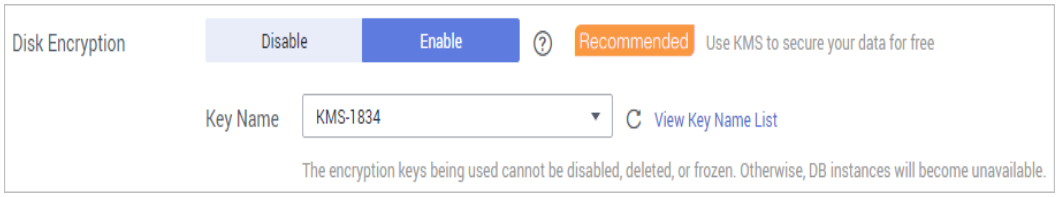
Restrictions

- 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 1-20 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

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

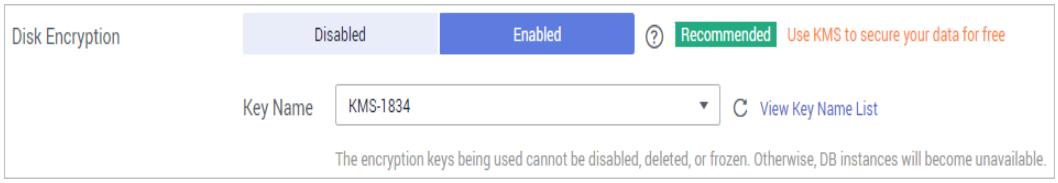
Restrictions

- 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 1-21 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.

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.

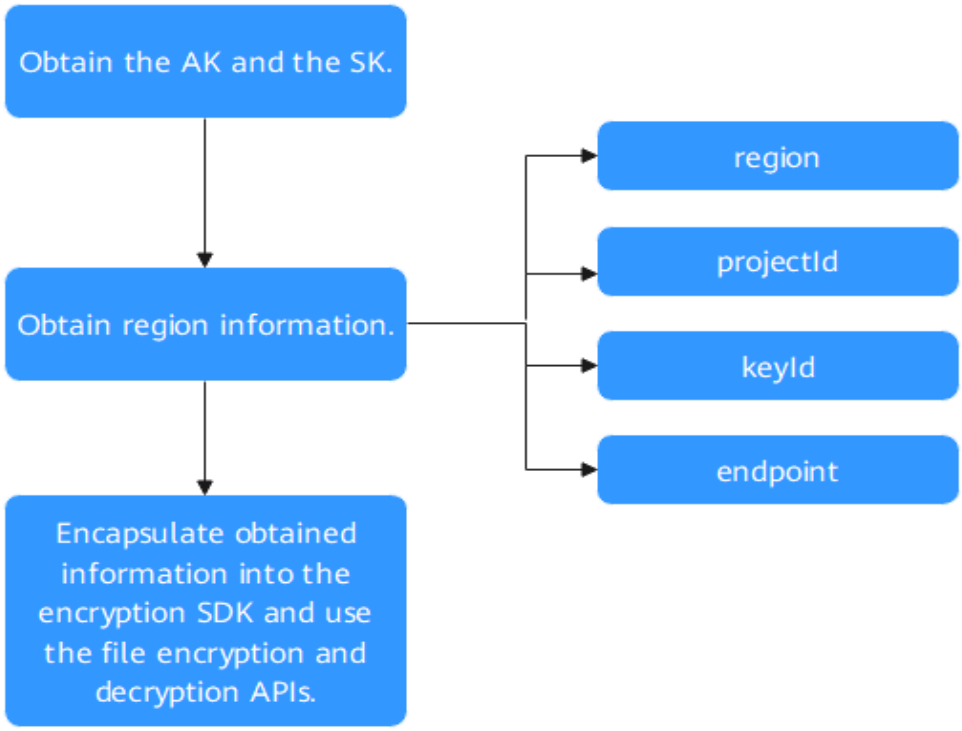
Solution

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.

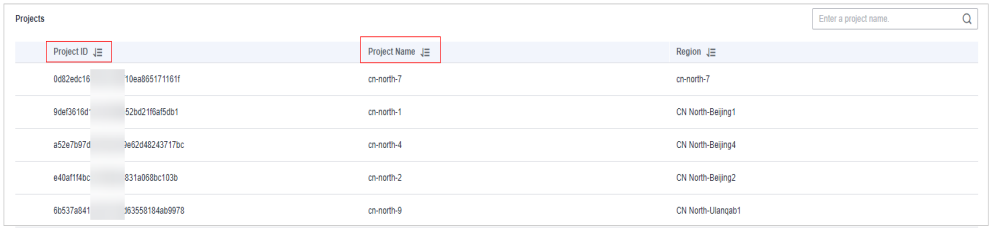
- **ACCESS_KEY**: Access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**
- **SECRET_ACCESS_KEY**: Secret access key of the Huawei account. For details, see **How Do I Obtain an Access Key (AK/SK)?**

- **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.

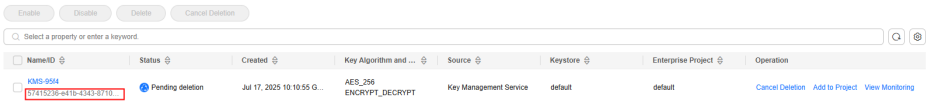
1. Log in to the **DEW console**.
2. Hover over the username in the upper right corner and choose **My Credentials** from the drop-down list.
3. Obtain the **Project ID** and **Project Name**.

   Figure 1-22 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



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
```



```
        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.

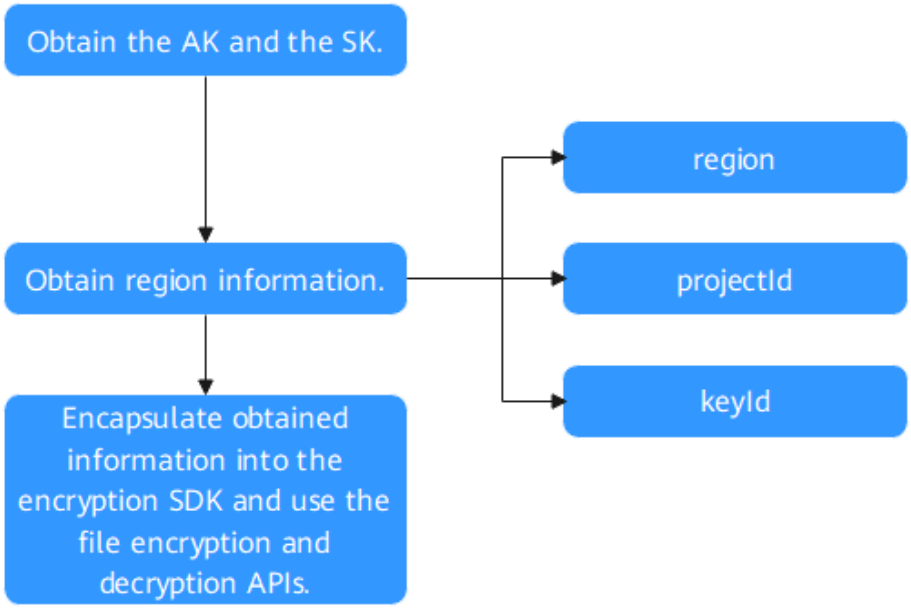
Solution

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

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

Process



Procedure

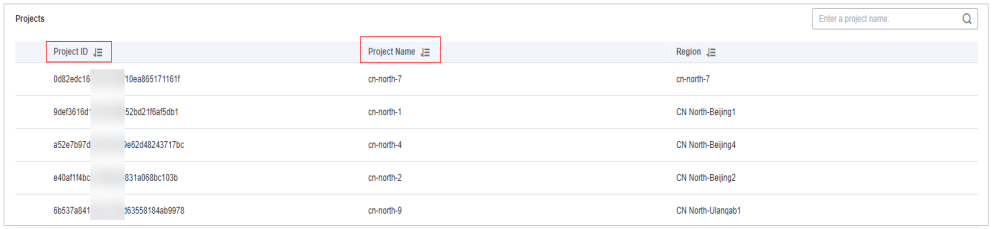
Step 1 Obtain the AK and the SK.

- **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.

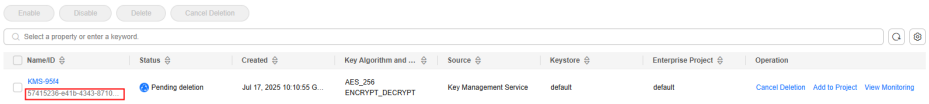
1. Log in to the **DEW console**.
2. Hover over the username in the upper right corner and choose **My Credentials** from the drop-down list.
3. Obtain the **Project ID** and **Project Name**.

Figure 1-24 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 1-25 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()));
```



```
    }
}
```

----End

1.5 Using KMS to Protect File Integrity

Scenario

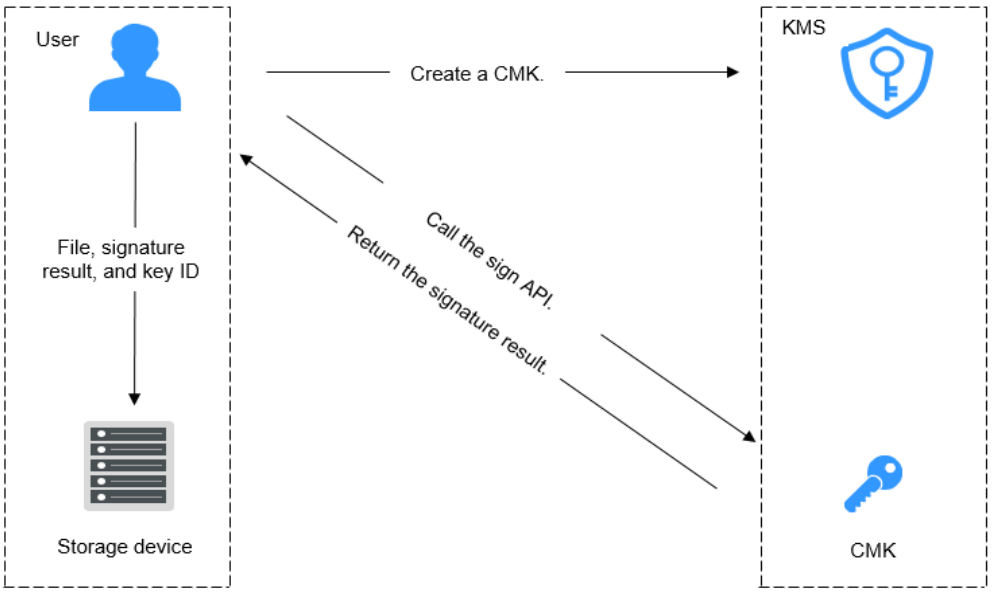
When a large amount of files (such as images, electronic insurance policies, and important files) need to be transmitted or stored securely, you can use KMS to sign the file digest. When the files are used again, you can recalculate the digest for signature verification. Ensure that files are not tampered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

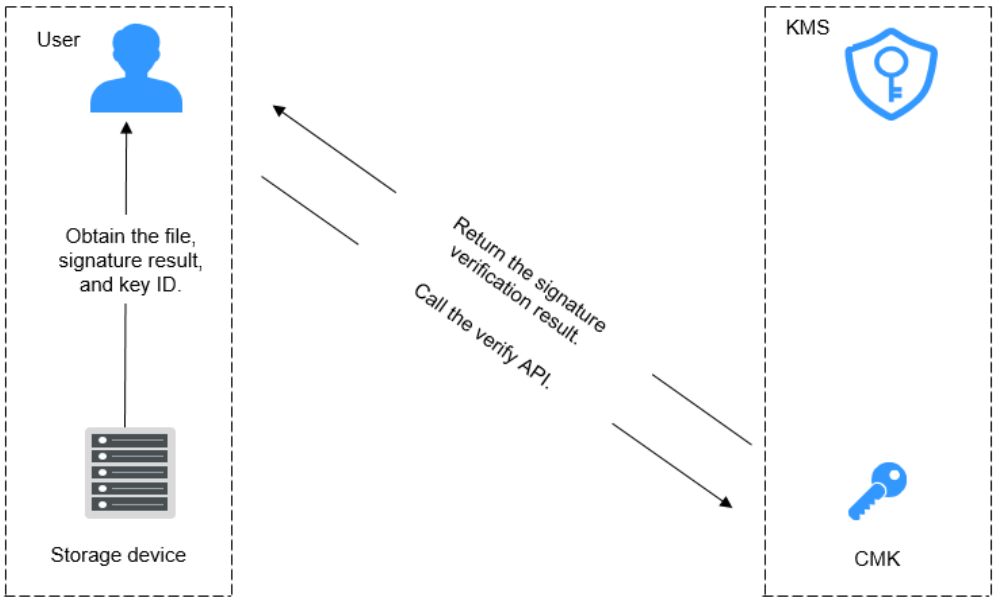
Figure 1-26 Signature process



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.

Figure 1-27 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>";
```



```
    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:

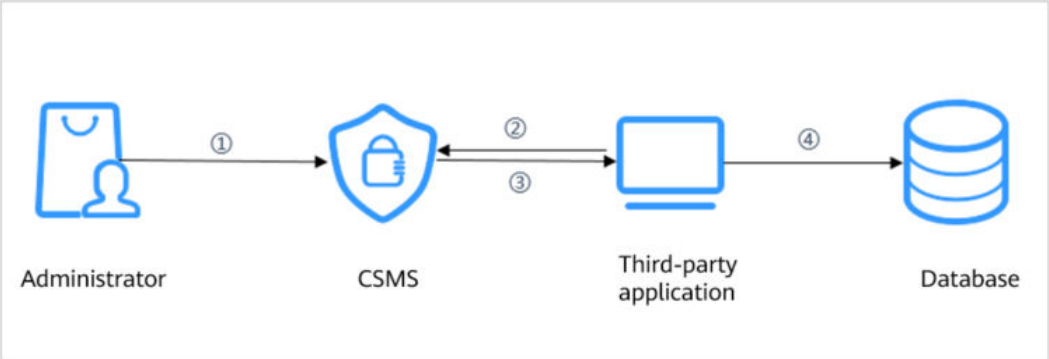
- You can host your secrets instead of using hardcoded secrets, improving the security of data and assets.
- Secure SDK access allows you to dynamically call your secrets.
- You can store many types of secrets. You can store service accounts, passwords, and database information, including but not limited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

Figure 2-1 Secret-based login process



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.

- 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.

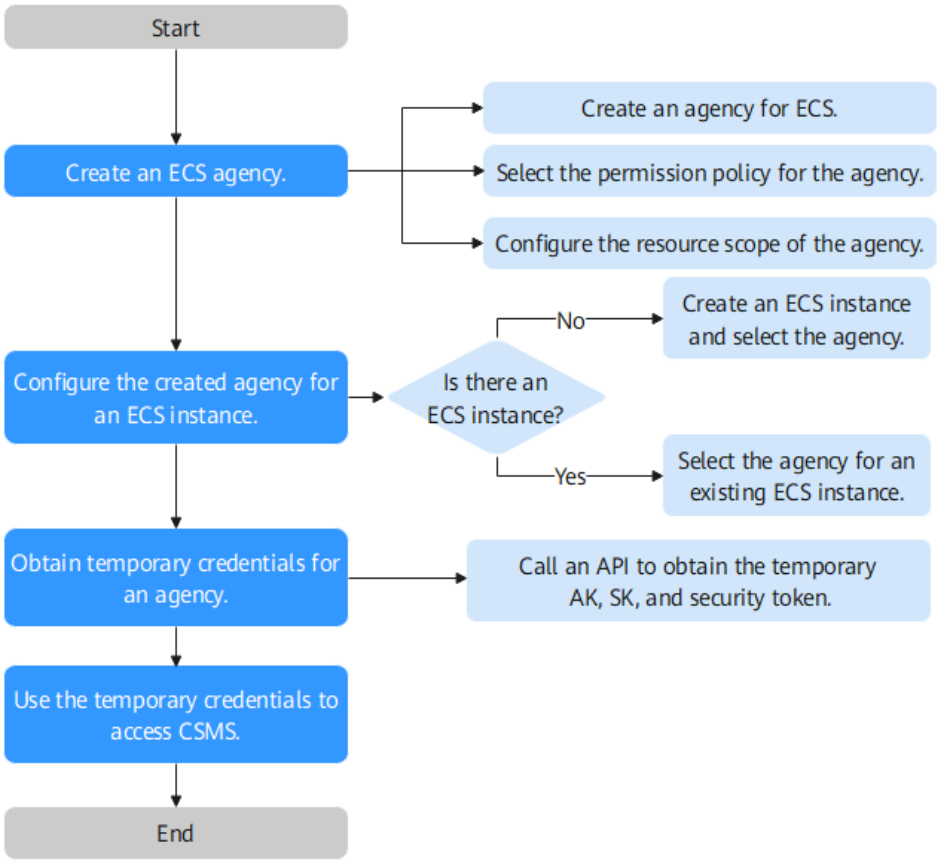
How It Works

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.

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**.

Figure 2-3 Creating an agency

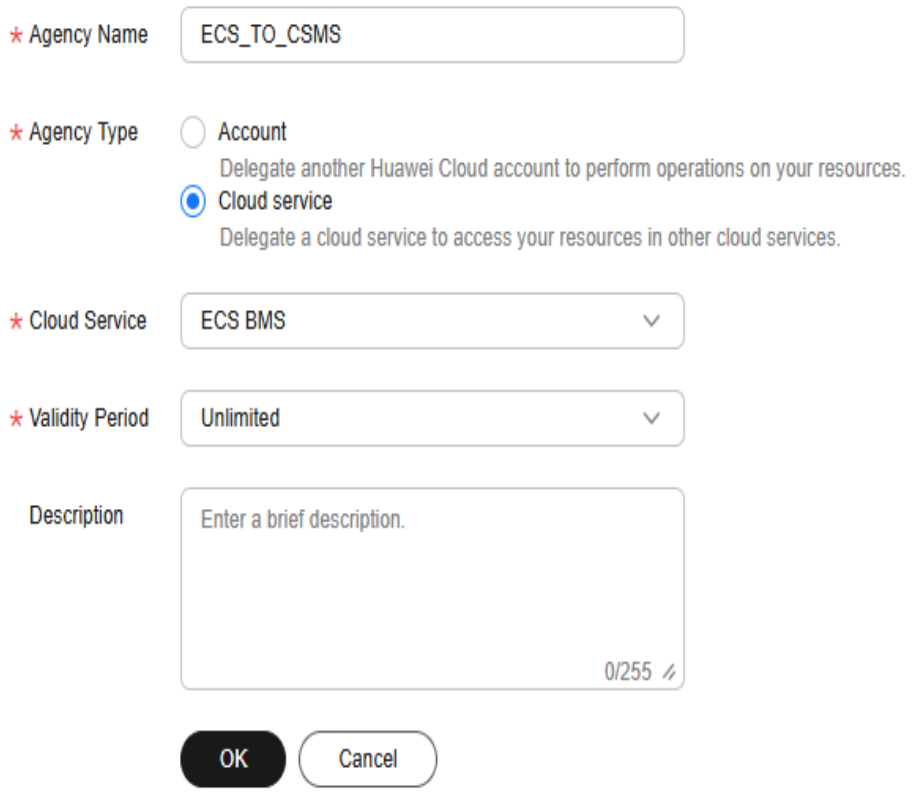


Table 2-1 Agency parameters

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**.

Figure 2-4 Creating a policy

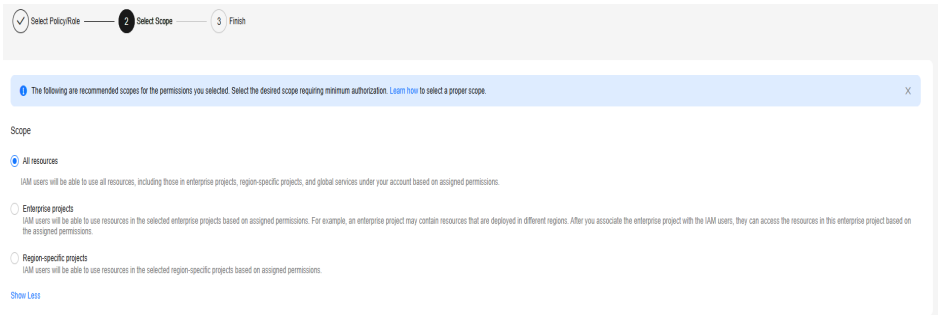
Table 2-2 Policy parameters

Parameter Name	Description
Policy Name	Enter a policy name.
Policy View	Select **Visual editor**.
Policy Content	■ **Allow**: Select **Allow**. ■ **Cloud Service**: Select **CSMS** and **KMS**. **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.

- **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.

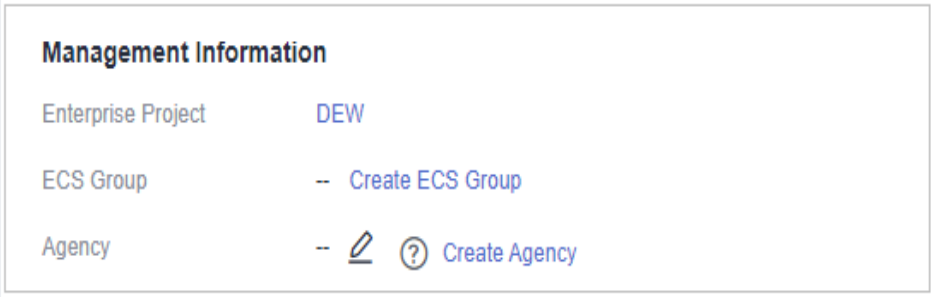
- If no ECS has been created, create an agency, for example, **ECS_TO_CSMS** for **Agency** under **Advanced Settings**. For details, see **Purchasing an ECS**.
- To use an existing ECS instance, perform the following steps:

  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://*xx.xx.xx.xx*/openstack/latest/securitykey

> **NOTE**
>
> Replace the IPv4 address *xx.xx.xx.xx* used in this example with the actual available address.

- Method

  GET request

- The following data is returned:

```
{
    "credential":{
        "access": "LDHZK30XXXXXXXXXXXV",
        "secret":"gyqcdzVXXXXXXXXXXXXXXXXXXXXXXXXMl6",
        "securitytoken": "El9FI2C65qXXXXXXXXXXXXXXXXXXXXnkcaoV",
        "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

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

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)?**

- **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) {
```



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

