AWS Encryption SDK

Developer Guide

AWS Encryption SDK: Developer Guide

Table of Contents

What Is the AWS Encryption SDK?

The AWS Encryption SDK provides client-side encryption libraries you can use to protect your data and the encryption keys used to encrypt that data. The SDK does the following things for you:

- Provides an API to define and use a *master key provider*, an interface for the top-level key or keys under which your data is encrypted.
- Tracks and protects the data encryption keys (DEKs) used to encrypt your data.
- Performs the low-level cryptographic operations.

You determine the top-level master keys that protect your data, and the SDK does the rest. The SDK helps you connect the low-level cryptography to the top-level master keys. For more information about master keys, master key providers, data encryption keys, and other cryptography concepts related to this SDK, see Encryption Concepts (p. 1) and Architecture (p. 3).

The SDK is similar to the Amazon DynamoDB Encryption Client for Java and the Amazon S3 Encryption Client, but unlike those clients the data encrypted by this SDK can be stored anywhere.

The SDK is provided for free under the Apache license and is available for the Java programming language at https://github.com/awslabs/aws-encryption-sdk-java.

Topics

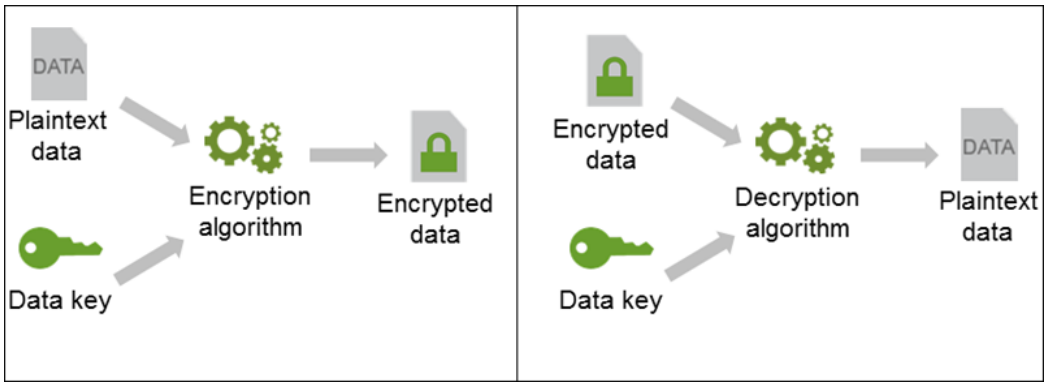
Encryption Concepts

You can use the AWS Encryption SDK to protect your data and the encryption keys used to encrypt that data.

Topics

Encryption Basics

To encrypt data, you provide the raw data (*plaintext*) and a *data key* to an encryption algorithm. The algorithm uses those inputs to produce encrypted data (*ciphertext*). To decrypt data, you provide the encrypted data and the data key to a decryption algorithm that uses those inputs to return the original data.



Some algorithms use the same data key to encrypt and decrypt data. This is called *symmetric key encryption*. Other algorithms use a *public key* to encrypt data, and only a related *private key* can decrypt that data. This is called *public key encryption*.

For both types of encryption, the security of your encrypted data depends on protecting the data key that can decrypt it. One accepted best practice for protecting the data key is to encrypt it. To encrypt the data key you need another encryption key called a *key encryption key* (KEK). This practice of using KEKs to encrypt data keys is called *envelope encryption*.

Envelope Encryption

Envelope encryption is the practice of encrypting plaintext data with a unique data key, and then encrypting the data key with a KEK. You might choose to encrypt the KEK with another KEK, and so on, but eventually you must have a *master key*. The master key is an unencrypted (plaintext) key with which you can decrypt one or more other keys.

Some of the benefits of envelope encryption include:

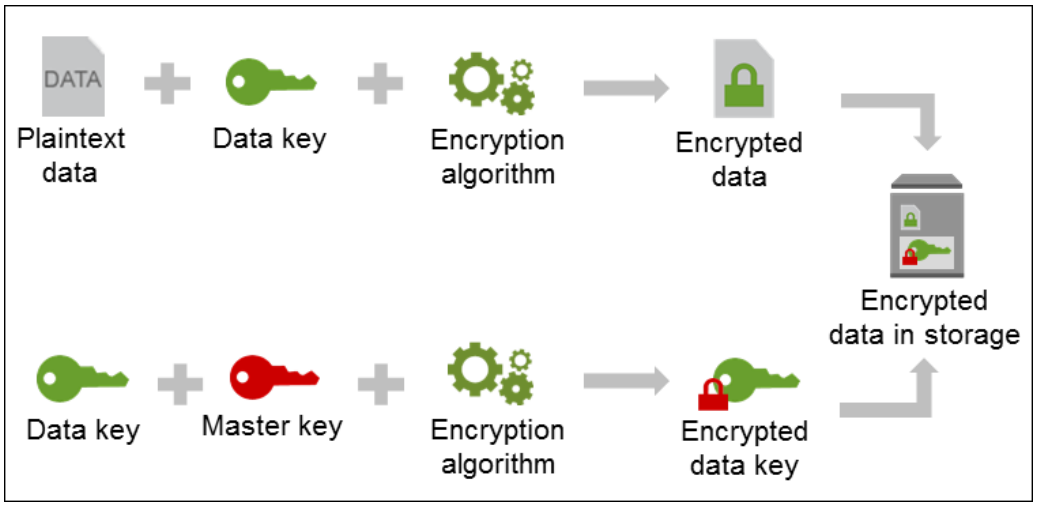
- **Protecting data keys**

  When you encrypt a data key, you do not have to worry about where to store the encrypted data key, because the security of that data key is inherently protected by encryption. You can safely store the encrypted data key alongside the encrypted data. The AWS Encryption SDK takes care of this for you by combining the encrypted data key and the encrypted data into a single encrypted message.
- **Encrypting the same data under multiple master keys**

  Encryption operations can be time-consuming, particularly when the data being encrypted are large objects. Instead of re-encrypting raw data multiple times with different keys, you can re-encrypt only the data keys that protect the raw data.
- **Combining the strengths of multiple algorithms**

  In general, symmetric key algorithms are faster and produce smaller ciphertexts than public key algorithms, but public key algorithms provide inherent separation of roles and easier key management. You might want to combine the strengths of each. For example, you might encrypt raw data with symmetric key encryption, and then encrypt the data key with public key encryption.

The following image provides an overview of envelope encryption. In this scenario, the data key is encrypted with a single KEK, which is the master key.



When you use envelope encryption, you must protect the master keys from unauthorized access. To protect your master keys, you can use a hardware security module (HSM) (for example, those offered by AWS CloudHSM), you can use the AWS Key Management Service (AWS KMS), or you can use your existing key management tools.

The AWS Encryption SDK supports the use of AWS KMS to protect your master keys, or you can use another master key provider, including a custom one. Even if you don't use AWS, you can still use this SDK.

Architecture

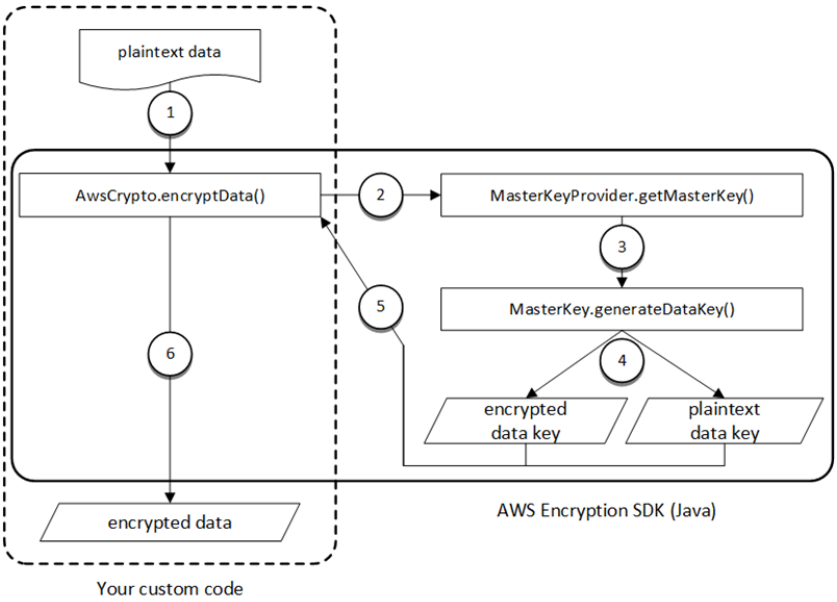
The AWS Encryption SDK provides methods that operate on byte arrays, byte streams, and strings. The following topics provide a high-level overview of how this SDK works.

For code samples in Java, see Example Code (Java) (p. 15).

Topics

Encryption

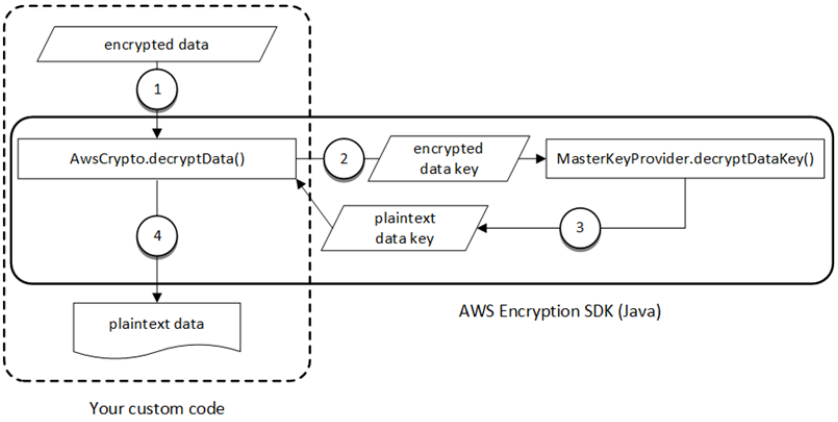
The following diagram shows how you can use the AWS Encryption SDK to encrypt data.



1. Your application passes data to one of the encryption methods.
2. The encryption method uses a master key provider to determine which master key to use.
3. The master key generates a data key.
4. The master key creates two copies of the data key, one in plaintext and one encrypted by the master key.
5. The encryption method uses the plaintext data key to encrypt the data, and then deletes the plaintext data key.
6. The encryption method returns, in a single message, encrypted data that consists of the plaintext data and the encrypted data key.

Decryption

The following diagram shows a high-level overview of how you can use the AWS Encryption SDK to decrypt data.



1. Your application passes encrypted data to one of the decryption methods.
2. The decryption method extracts the encrypted data key from the encrypted data, and then sends the encrypted data key to a master key provider for decryption.

3. The master key provider decrypts the encrypted data key, and then returns the plaintext data key to the decryption method.
4. The decryption method uses the plaintext data key to return the plaintext data, and then deletes the plaintext data key.

Getting Started

To get started with the AWS Encryption SDK, follow the steps in the following topics.

Topics

(Optional) Create an AWS Account

To use some of the example Java code (p. 15) in this guide, you need to create an AWS account and then create a customer master key (CMK) in AWS Key Management Service (AWS KMS). Some of the sample code demonstrates how to use a CMK in AWS KMS to protect the data keys that encrypt your data.

To create an AWS account

1. Go to the Sign In or Create an AWS Account page.
2. Type your email address or mobile phone number, and then choose **I am a new user.** Choose **Sign in using our secure server**.
3. Follow the instructions on the website.

During the sign-up process, you receive a phone call and enter a PIN with the phone keypad. You must also enter a valid credit card number during the sign-up process.

To create a customer master key (CMK) in AWS KMS

1. Open the Creating Keys page in the *AWS Key Management Service Developer Guide*.
2. Follow the instructions on that page.

Download the AWS Encryption SDK

The AWS Encryption SDK is currently available for the Java programming language. Before you download the SDK, you must have the following:

- **A Java 8 development environment**

  If you do not have one, go to Java SE Downloads and then download and install the Java SE Development Kit (JDK). Java 8 or higher is recommended.
- **Bouncy Castle**

  Bouncy Castle provides a cryptography API for Java. If you do not have Bouncy Castle, go to https://bouncycastle.org/latest_releases.html, and then download the provider file that corresponds to your JDK.
- **(Optional) AWS SDK for Java**

Although you do not need the AWS SDK for Java to use the AWS Encryption SDK, you do need it to use some of the example Java code (p. 15) in this guide. To download the AWS SDK for Java, go to http://aws.amazon.com/sdk-for-java/. For more information about installing and configuring the AWS SDK for Java, see Installing the AWS SDK for Java in the *AWS SDK for Java Developer Guide*.

If you already have these prerequisites, or after you have downloaded and installed them, you can download the AWS Encryption SDK at https://github.com/awslabs/aws-encryption-sdk-java.

If you use Apache Maven, you can specify the AWS Encryption SDK as a dependency in your project. Add the following dependency to your application's pom.xml file:

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

After you download the AWS Encryption SDK, see Example Code (Java) (p. 15) for examples that demonstrate how to use it.

Message Format

The encryption operations in the AWS Encryption SDK return a single data structure or *message* that contains the encrypted data and the encrypted data key. To understand this data structure, or to build libraries that read and write it, you need to understand the message format.

The message format consists of at least two parts: a *header* and a *body*. In some cases, the message format consists of a third part called a *footer*. The message format defines an ordered sequence of bytes in network byte order, also called big-endian format. The message format begins with the header, followed by the body, followed by the footer (when applicable).

> **Note**
> The following information is provided for reference only. The message format should not be modified.

Topics

Header Structure

The message header contains the encrypted data key and information about how the message body is formed. The following table describes the fields that form the header. The bytes are appended in the order shown.

Header Structure

Field	Length, in bytes
Version	1
Type	1
Algorithm ID	2
Message ID	16
AAD Length	2

Field	Length, in bytes
AAD	Variable. Equal to the value specified in the previous 2 bytes (AAD Length).
Encrypted Data Key Count	2
Encrypted Data Key(s)	Variable. Determined by the number of encrypted data keys and the length of each.
Content Type	1
Reserved	4
IV Length	1
Frame Length	4
IV	Variable. Equal to the value specified in the IV Length byte.
Authentication Tag	Variable. Determined by the algorithm used, as specified in Algorithm ID.

Version
The version of this message format. The current version is 1.0, encoded as the byte `01` in hexadecimal notation.

Type
The type of this message format. The type indicates the kind of structure with regard to the AWS Key Management Service (AWS KMS). This message format is described as *customer authenticated encrypted data*. Its type value is 128, encoded as byte `80` in hexadecimal notation.

Algorithm ID
An identifier for the algorithm used. It is a 2-byte value interpreted as a 16-bit unsigned integer. The following table shows the supported algorithm IDs, in hexadecimal notation, and the algorithm that corresponds to each ID.

For more information about these algorithms, see Which cryptographic algorithms are supported by the AWS Encryption SDK, and which one is the default? (p. 24) on the Frequently Asked Questions (p. 23) page.

Algorithms IDs

Algorithm ID, in 2-byte hex	Corresponding algorithm
`00 14`	ALG_AES_128_GCM_IV12_TAG16_NO_KDF
`00 46`	ALG_AES_192_GCM_IV12_TAG16_NO_KDF
`00 78`	ALG_AES_256_GCM_IV12_TAG16_NO_KDF
`01 14`	ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256
`01 46`	ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA256
`01 78`	ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256

Algorithm ID, in 2-byte hex	Corresponding algorithm
`02 14`	ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256_ECDSA_P256
`03 46`	ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384
`03 78`	ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384

Message ID

A randomly-generated 128-bit value that identifies the message. The Message ID:

- Uniquely identifies the encrypted message.
- Weakly binds the message header to the message body.
- Provides a mechanism to securely reuse an AWS KMS-encrypted data key with multiple encrypted objects.
- Protects against accidental reuse of a data key or the wearing out of keys in the AWS Encryption SDK.

AAD Length

The length of the additional authenticated data (AAD). It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of bytes that contain the AAD.

AAD

The additional authenticated data. The AAD is an encoding of the encryption context, an array of key-value pairs where each key and value is a string of UTF-8 encoded characters. The encryption context is converted to a sequence of bytes and used for the AAD value.

When the signed algorithms (p. 24) are used, the encryption context must contain the key-value pair `{'aws-crypto-public-key', Qtxt}` where Qtxt is the base64-encoded text of the compressed elliptic curve point Q. The encryption context can contain additional values.

The following table describes the fields that form the AAD. Key-value pairs are sorted, by key, in ascending order according to UTF-8 character code.

AAD Structure

Field	Length, in bytes
Key-Value Pair Count	2
Key Length	2
Key	Variable. Equal to the value specified in the previous 2 bytes (Key Length).
Value Length	2
Value	Variable. Equal to the value specified in the previous 2 bytes (Value Length).

Key-Value Pair Count

The number of key-value pairs in the AAD. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of key-value pairs in the AAD.

Key Length

The length of the key for the key-value pair. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of bytes that contain the key.

Key
The key for the key-value pair. It is a sequence of UTF-8 encoded bytes.

Value Length
The length of the value for the key-value pair. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of bytes that contain the value.

Value
The value for the key-value pair. It is a sequence of UTF-8 encoded bytes.

Encrypted Data Key Count
The number of encrypted data keys. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of encrypted data keys.

Encrypted Data Key(s)
A sequence of encrypted data keys. The length of the sequence is determined by the number of encrypted data keys and the length of each. The sequence contains at least one encrypted data key.

The following table describes the fields that form each encrypted data key. The bytes are appended in the order shown.

Encrypted Data Key Structure

Field	Length, in bytes
Key Provider ID Length	2
Key Provider ID	Variable. Equal to the value specified in the previous 2 bytes (Key Provider ID Length).
Key Provider Information Length	2
Key Provider Information	Variable. Equal to the value specified in the previous 2 bytes (Key Provider Information Length).
Encrypted Data Key Length	2
Encrypted Data Key	Variable. Equal to the value specified in the previous 2 bytes (Encrypted Data Key Length).

Key Provider ID Length
The length of the key provider identifier. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of bytes that contain the key provider ID.

Key Provider ID
The key provider identifier. It is used to indicate the provider of the encrypted data key and intended to be extensible.

Key Provider Information Length
The length of the key provider information. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of bytes that contain the key provider information.

Key Provider Information
The key provider information. It is determined by the key provider. When AWS KMS is the key provider, this value contains the Amazon Resource Name (ARN) of the AWS KMS customer master key (CMK).

Encrypted Data Key Length
The length of the encrypted data key. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of bytes that contain the encrypted data key.

Encrypted Data Key
The encrypted data key. It is determined by the key provider, but should be the underlying data encryption key encrypted by the key provider. For the AWS KMS provider, it is a binary ciphertext blob as returned by the AWS KMS `GenerateDataKey` API operation.

Content Type
The type of encrypted content, either non-framed or framed. Non-framed content is not broken into parts; it is a single encrypted blob. Framed content is broken into equal-length parts; each part is encrypted separately.

Non-framed content is type 1, encoded as the byte `01` in hexadecimal notation. Framed content is type 2, encoded as the byte `02` in hexadecimal notation.

Reserved
A reserved sequence of 4 bytes. This value must be 0. It is encoded as the bytes `00 00 00 00` in hexadecimal notation (that is, a 4-byte sequence of a 32-bit integer value equal to 0).

IV Length
The length of the initialization vector (IV). It is a 1-byte value interpreted as an 8-bit unsigned integer that specifies the number of bytes that contain the IV. All of the algorithms supported by the AWS Encryption SDK use a 12-byte (96-bit) IV, so the IV length is encoded as the byte `0C` in hexadecimal notation.

Frame Length
The length of each frame of framed content. It is a 4-byte value interpreted as a 32-bit unsigned integer that specifies the number of bytes that form each frame. When the content is non-framed—that is, when the value of the Content Type field is 1—this value must be 0.

IV
The initialization vector for the header authentication tag. It is used to generate the header authentication tag over the header fields up to, but not including, the IV.

Authentication Tag
The authentication value for the header. It is used to authenticate the header fields up to, but not including, the IV.

Body Structure

The message body contains the encrypted data. The structure of the body depends on the content type (non-framed or framed). The following sections describe the format of the message body for each content type.

Topics

Non-Framed Data

Non-framed data is encrypted in a single blob. The following table describes the fields that form non-framed data. The bytes are appended in the order shown.

Non-Framed Body Structure

Field	Length in bytes
IV	Variable. Equal to the value specified in the IV Length byte of the header.
Encrypted Content Length	8
Encrypted Content	Variable. Equal to the value specified in the previous 8 bytes (Encrypted Content Length).

Field	Length in bytes
Authentication Tag	Variable. Determined by the algorithm used, as specified in the Algorithm ID field of the header.

IV
: The initialization vector. The IV for the implemented encryption mode, the Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM) known as AES-GCM, is a randomly-generated 12-byte value.

Encrypted Content Length
: The length of the encrypted content. It is an 8-byte value interpreted as a 64-bit unsigned integer that specifies the number of bytes that contain the encrypted content.

  The maximal allowed value is 2^63 - 1 or 8 exbibytes (8 EiB). However, for the implemented mode of AES-GCM, the maximum value is 2^36 - 32 or 64 gibibytes (64 GiB), due to restrictions on the use of AES-GCM. The Java implementation of this SDK further restricts this value to 2^31 - 1 or 2 gibibytes (2 GiB), due to restrictions in the implementation.

Encrypted Content
: The encrypted content.

Authentication Tag
: The authentication value for the body. It is used to authenticate the body fields up to, but not including, the authentication tag.

Framed Data

Framed data is divided into equal-length parts, except for the last part. Each frame is encrypted separately with a unique IV and AAD.

There are two kinds of frames: a regular frame and a final frame. A final frame is always used, even when the content fits into a single regular frame. In this case, the final frame contains no data—that is, a content length of 0.

The following tables describe the fields that form the frames. The bytes are appended in the order shown.

Framed Body Structure, Regular Frame

Field	Length (bytes)
Sequence Number	4
IV	Variable. Equal to the value specified in the IV Length byte of the header.
Encrypted Content	Variable. Equal to the value specified in the Frame Length of the header.
Authentication Tag	Variable. Determined by the algorithm used, as specified in the Algorithm ID of the header.

Sequence Number
: The frame sequence number. It is an incremental counter number for the frame. Framed data must start at sequence number 1, encoded as the 4 bytes `00 00 00 01`, in hexadecimal notation. The frames must be in order and must contain an increment of 1 of the previous frame. Otherwise, the decryption process stops and reports an error.

IV
The initialization vector for the frame. The IV for the implemented encryption mode of AES-GCM is a randomly-generated 12-byte value.

Encrypted Content
The encrypted content for the frame.

Authentication Tag
The authentication value for the frame. It is used to authenticate the frame fields up to, but not including, the authentication tag.

Framed Body Structure, Final Frame

Field	Length (bytes)
Sequence Number End	4
Sequence Number	4
IV	Variable. Equal to the value specified in the IV Length byte of the header.
Encrypted Content Length	4
Encrypted Content	Variable. Equal to the value specified in the previous 4 bytes (Encrypted Content Length).
Authentication Tag	Variable. Determined by the algorithm used, as specified in the Algorithm ID of the header.

Sequence Number End
An indicator for the final frame. The value is encoded as the 4 bytes `FF FF FF FF`, in hexadecimal notation.

Sequence Number
The frame sequence number. It is an incremental counter number for the frame. Framed data must start at sequence number 1, encoded as the 4 bytes `00 00 00 01`, in hexadecimal notation. The frames must be in order and must contain an increment of 1 of the previous frame. Otherwise, the decryption process stops and reports an error.

IV
The initialization vector for the frame. The IV for the implemented encryption mode of AES-GCM is a randomly-generated 12-byte value.

Encrypted Content Length
The length of the encrypted content for the frame. It is a 4-byte value interpreted as a 32-bit unsigned integer that specifies the number of bytes that contain the encrypted content for the frame.

Encrypted Content
The encrypted content for the frame.

Authentication Tag
The authentication value for the frame. It is used to authenticate the frame fields up to, but not including, the authentication tag.

Footer Structure

When the signed algorithms (p. 24) are used, the message format contains a footer. The message footer contains a signature that authenticates the message header and message body. The following table describes the fields that form the footer. The bytes are appended in the order shown.

Footer Structure

Field	Length in bytes
Signature Length	2
Signature	Variable. Equal to the value specified in the previous 2 bytes (Signature Length).

Signature Length
The length of the signature. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of bytes that contain the signature.

Signature
The signature. It is used to authenticate the header and body of the message.

Example Code (Java)

The following examples demonstrate how you can use the Java implementation of the AWS Encryption SDK to encrypt and decrypt data.

Topics

Encrypting and Decrypting Strings

The following example demonstrates how you can use the AWS Encryption SDK to encrypt and decrypt strings. This example uses a customer master key (CMK) in AWS Key Management Service (AWS KMS) as the master key.

```
/*
 * Copyright 2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not
 use this file except
 * in compliance with the License. A copy of the License is located at
 *
 * https://aws.amazon.com/apache-2-0/
 *
 * or in the "license" file accompanying this file. This file is distributed
on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See
 the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.crypto.examples;

import java.util.Collections;
import java.util.Map;

import com.amazonaws.encryptionsdk.AwsCrypto;
```

```
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.kms.KmsMasterKey;
import com.amazonaws.encryptionsdk.kms.KmsMasterKeyProvider;

/**
 * <p>
 * Encrypts and then decrypts a string under a KMS customer master key (CMK)
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Amazon Resource Name (ARN) of the KMS CMK
 * <li>String to encrypt
 * </ol>
 */
public class StringExample {
    private static String keyArn;
    private static String data;

    public static void main(final String[] args) {
        keyArn = args[0];
        data = args[1];

        // Instantiate the SDK
        final AwsCrypto crypto = new AwsCrypto();

        // Set up the KmsMasterKeyProvider backed by the default credentials
        final KmsMasterKeyProvider prov = new KmsMasterKeyProvider(keyArn);

        // Encrypt the data
        //
        // Most encrypted data should have associated encryption context to
        // protect integrity. For this example, just use a placeholder value.
        //
        // For more information about encryption context,
        // see https://amzn.to/1nSbe9X (blogs.aws.amazon.com)
       final Map<String, String> context = Collections.singletonMap("Example",
 "String");

        final String ciphertext = crypto.encryptString(prov, data, con
text).getResult();
        System.out.println("Ciphertext: " + ciphertext);

        // Decrypt the data
        final CryptoResult<String, KmsMasterKey> decryptResult = crypto.de
cryptString(prov, ciphertext);
        // Check the encryption context (and ideally the master key) to ensure

        // this was the expected ciphertext
        if (!decryptResult.getMasterKeyIds().get(0).equals(keyArn)) {
            throw new IllegalStateException("Wrong key id!");
        }

        // The SDK may add information to the encryption context, so check to
ensure
        // that all of the values are present
        for (final Map.Entry<String, String> e : context.entrySet()) {
            if (!e.getValue().equals(decryptResult.getEncryptionCon
```

```
text().get(e.getKey())) {
                throw new IllegalStateException("Wrong Encryption Context!");
            }
        }

        // The data is correct, so output it.
        System.out.println("Decrypted: " + decryptResult.getResult());
    }
}
```

Encrypting and Decrypting Byte Streams

The following example demonstrates how you can use the AWS Encryption SDK to encrypt and decrypt byte streams. This example does not use AWS. It uses the Java Cryptography Extension (JCE) to protect the master key.

```
/*
 * Copyright 2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not
 use this file except
 * in compliance with the License. A copy of the License is located at
 *
 * https://aws.amazon.com/apache-2-0/
 *
 * or in the "license" file accompanying this file. This file is distributed
on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See
 the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.crypto.examples;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.SecureRandom;
import java.util.Collections;
import java.util.Map;

import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoInputStream;
import com.amazonaws.encryptionsdk.MasterKey;
import com.amazonaws.encryptionsdk.jce.JceMasterKey;
import com.amazonaws.util.IOUtils;

/**
 * <p>
 * Encrypts and then decrypts a file under a random key.
 *
```

```
 * <p>
 * Arguments:
 * <ol>
 * <li>Name of the file to encrypt
 * </ol>
 *
 * <p>
 * This program demonstrates how to use a normal Java {@link SecretKey} object
 as a {@link MasterKey}
 * to encrypt and decrypt streaming data.
 */
public class FileStreamingExample {
    private static String srcFile;

    public static void main(String[] args) throws IOException {
        srcFile = args[0];

       // In a production implementation, load this master key from an existing
 store.
        // For this example, just generate a random one.
        SecretKey cryptoKey = retrieveEncryptionKey();

        // Convert the master key into a provider. This example uses AES-GCM
because it is
        // a secure algorithm.
       JceMasterKey masterKey = JceMasterKey.getInstance(cryptoKey, "Example",
 "RandomKey", "AES/GCM/NoPadding");

        // Instantiate the SDK
        AwsCrypto crypto = new AwsCrypto();

        // Create the encryption context to identify this ciphertext
        // For more information about encryption context,
        // see https://amzn.to/1nSbe9X (blogs.aws.amazon.com)
        Map<String, String> context = Collections.singletonMap("Example",
"FileStreaming");

        // The file might be really big, so don't load
        // it all into memory. Streaming is necessary.
        FileInputStream in = new FileInputStream(srcFile);
       CryptoInputStream<JceMasterKey> encryptingStream = crypto.createEncrypt
ingStream(masterKey, in, context);

        FileOutputStream out = new FileOutputStream(srcFile + ".encrypted");
        IOUtils.copy(encryptingStream, out);
        encryptingStream.close();
        out.close();

        // Decrypt the file now, remembering to check the encryption context
        in = new FileInputStream(srcFile + ".encrypted");
       CryptoInputStream<JceMasterKey> decryptingStream = crypto.createDecrypt
ingStream(masterKey, in);
        // Does it have the right encryption context?
        if (!"FileStreaming".equals(decryptingStream.getCryptoResult().getEn
cryptionContext().get("Example"))) {
            throw new IllegalStateException("Bad encryption context");
        }
```

```
        // Finally, write out the data
        out = new FileOutputStream(srcFile + ".decrypted");
        IOUtils.copy(decryptingStream, out);
        decryptingStream.close();
        out.close();
    }

    /**
     * In a production implementation, this key needs to be persisted somewhere.
 For this demo,
     * just generate a new random one each time.
     */
    private static SecretKey retrieveEncryptionKey() {
        SecureRandom rnd = new SecureRandom();
        byte[] rawKey = new byte[16]; // 128 bits
        rnd.nextBytes(rawKey);
        return new SecretKeySpec(rawKey, "AES");
    }
}
```

Encrypting and Decrypting Byte Streams with Multiple Master Key Providers

The following example demonstrates how you can use the AWS Encryption SDK with more than one master key provider. Using more than one master key provider creates redundancy in case one master key provider is unavailable for decryption. This example uses a CMK in AWS KMS and an RSA key pair as the master keys.

```
/*
 * Copyright 2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not
 use this file except
 * in compliance with the License. A copy of the License is located at
 *
 * https://aws.amazon.com/apache-2-0/
 *
 * or in the "license" file accompanying this file. This file is distributed
on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See
 the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.crypto.examples;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.GeneralSecurityException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
```

```
import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoOutputStream;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.jce.JceMasterKey;
import com.amazonaws.encryptionsdk.kms.KmsMasterKeyProvider;
import com.amazonaws.encryptionsdk.multi.MultipleProviderFactory;
import com.amazonaws.util.IOUtils;

/**
 * <p>
 * Encrypts a file with two master keys: a KMS CMK and an asymmetric key pair.

 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Amazon Resource Name (ARN) of the KMS CMK
 * <li>Name of the file to encrypt
 * </ol>
 *
 * Some organizations want the ability to decrypt their data even if KMS is
unavailable. This
 * program demonstrates one possible way of accomplishing this by generating
an "escrow" RSA
 * key pair and using that, in addition to the KMS CMK, as the master key for
encryption.
 * The organization should keep the RSA private key in a secure place (such as
 an offline HSM) and
 * distribute the public key to their developers. This way, normal use would
use the KMS CMK
 * for decryption, but the organization maintains the ability to decrypt all
ciphertexts in a
 * completely offline manner.
 */
public class EscrowedEncryptExample {
    private static PublicKey publicEscrowKey;
    private static PrivateKey privateEscrowKey;

    public static void main(final String[] args) throws Exception {
        // In a production implementation, the public key is distributed by the
 organization.
        // For this example, just generate a new random one each time.
        generateEscrowKeyPair();

        final String kmsArn = args[0];
        final String fileName = args[1];

        standardEncrypt(kmsArn, fileName);
        standardDecrypt(kmsArn, fileName);

        escrowDecrypt(fileName);
    }

   private static void standardEncrypt(final String kmsArn, final String file
Name) throws Exception {
        // Standard user encrypting to both KMS and the escrow public key
        // 1. Instantiate the SDK
```

```
        final AwsCrypto crypto = new AwsCrypto();

        // 2. Instantiate the providers
        final KmsMasterKeyProvider kms = new KmsMasterKeyProvider(kmsArn);
       // Note that the standard user does not have access to the private escrow

        // key and so simply passes in "null"
       final JceMasterKey escrowPub = JceMasterKey.getInstance(publicEscrowKey,
 null, "Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

        // 3. Combine the providers into a single one
       final MasterKeyProvider<?> provider = MultipleProviderFactory.buildMul
tiProvider(kms, escrowPub);

        // 4. Encrypt the file
        // To simplify the code, this example omits encryption context this
time. Production code
        // should always use encryption context. See https://amzn.to/1nSbe9X
(blogs.aws.amazon.com)
        // for more information.
        final FileInputStream in = new FileInputStream(fileName);
        final FileOutputStream out = new FileOutputStream(fileName + ".encryp
ted");
       final CryptoOutputStream<?> encryptingStream = crypto.createEncrypting
Stream(provider, out);

        IOUtils.copy(in, encryptingStream);
        in.close();
        encryptingStream.close();
    }

   private static void standardDecrypt(final String kmsArn, final String file
Name) throws Exception {
       // A standard user decrypts the file. The user can use the same provider
 from before,
       // or can use a provider that refers to the KMS CMK. It doesn't matter.


        // 1. Instantiate the SDK
        final AwsCrypto crypto = new AwsCrypto();

        // 2. Instantiate the providers
        final KmsMasterKeyProvider kms = new KmsMasterKeyProvider(kmsArn);
       // Note that the standard user does not have access to the private escrow

        // key and so simply passes in "null"
       final JceMasterKey escrowPub = JceMasterKey.getInstance(publicEscrowKey,
 null, "Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

        // 3. Combine the providers into a single one
       final MasterKeyProvider<?> provider = MultipleProviderFactory.buildMul
tiProvider(kms, escrowPub);

        // 4. Decrypt the file
        // To simplify the code, this example omits encryption context this
time. Production code
```

```
        // should always use encryption context. See https://amzn.to/1nSbe9X
(blogs.aws.amazon.com)
        // for more information.
       final FileInputStream in = new FileInputStream(fileName + ".encrypted");

        final FileOutputStream out = new FileOutputStream(fileName + ".decryp
ted");
       final CryptoOutputStream<?> decryptingStream = crypto.createDecrypting
Stream(provider, out);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();
    }

    private static void escrowDecrypt(final String fileName) throws Exception
{
        // The organization can decrypt using the private escrow key with no
calls to AWS KMS

        // 1. Instantiate the SDK
        final AwsCrypto crypto = new AwsCrypto();

        // 2. Instantiate the provider
        // Note that the organization does have access to the private escrow
key and can use it.
       final JceMasterKey escrowPriv = JceMasterKey.getInstance(publicEscrowKey,
 privateEscrowKey, "Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

        // 3. Decrypt the file
        // To simplify the code, this example omits encryption context this
time. Production code
        // should always use encryption context. See https://amzn.to/1nSbe9X
(blogs.aws.amazon.com)
        // for more information.
       final FileInputStream in = new FileInputStream(fileName + ".encrypted");

        final FileOutputStream out = new FileOutputStream(fileName +
".deescrowed");
       final CryptoOutputStream<?> decryptingStream = crypto.createDecrypting
Stream(escrowPriv, out);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();

    }

   private static void generateEscrowKeyPair() throws GeneralSecurityException
 {
        final KeyPairGenerator kg = KeyPairGenerator.getInstance("RSA");
        kg.initialize(4096); // Escrow keys should be very strong
        final KeyPair keyPair = kg.generateKeyPair();
        publicEscrowKey = keyPair.getPublic();
        privateEscrowKey = keyPair.getPrivate();

    }
}
```

Frequently Asked Questions

Which data types are supported by the AWS Encryption SDK?

The AWS Encryption SDK can encrypt raw bytes (byte arrays), I/O streams (byte streams), and strings. For examples, see Example Code (Java) (p. 15).

How is the AWS Encryption SDK different from the AWS SDKs?

The AWS SDKs provide language-specific APIs for all of the Amazon Web Services (AWS). The AWS Encryption SDK provides an API for client-side encryption and decryption that optionally integrates with AWS. The AWS Encryption SDK supports the AWS Key Management Service (AWS KMS) as a master key provider, which means the AWS Encryption SDK overlaps with the AWS SDKs regarding some of the AWS KMS API. However, the implementations of the AWS KMS API in the AWS SDKs do not manage data encryption keys for you. The AWS Encryption SDK manages data keys for you by inserting them (in encrypted form) into the encrypted data (ciphertexts) that are returned by the encryption methods. You can also use the AWS Encryption SDK without using AWS.

How is the AWS Encryption SDK different from the Amazon S3 encryption client in the AWS SDK for Java, the AWS SDK for Ruby, and the AWS SDK for .NET?

The Amazon S3 encryption client in the AWS SDK for Java, AWS SDK for Ruby, and AWS SDK for .NET provides client-side encryption and decryption for data stored in Amazon Simple Storage Service (Amazon S3). It is tightly coupled to Amazon S3 and is intended for use only with data stored in Amazon S3. The AWS Encryption SDK provides client-side encryption and decryption for data you can store anywhere. The encrypted data formats produced by the Amazon S3 encryption client and the AWS Encryption SDK are not interoperable.

How does the AWS Encryption SDK encode each type of encrypted output?

The AWS Encryption SDK does not add encoding to its output. The methods in the SDK that operate on strings return strings; the methods that operate on bytes return bytes.

How does the AWS Encryption SDK encrypt and decrypt input/output streams?

The AWS Encryption SDK creates an encrypting or decrypting stream that wraps an underlying I/O stream. The encrypting or decrypting stream performs a cryptographic operation on a read or write call. For example, it can read plaintext data on the underlying stream and encrypt it before returning the result, or read ciphertext from an underlying stream and decrypt it before returning the result. For example code that uses encrypting and decrypting streams, see Encrypting and Decrypting Byte Streams (p. 17).

How do I keep track of the data keys used to encrypt my data?

The AWS Encryption SDK does this for you. When you encrypt data, the AWS Encryption SDK creates a unique symmetric data encryption key for each data object, and the object's data key is encrypted and returned as part of the encrypted data. When you decrypt data, the AWS Encryption SDK extracts the encrypted data key, decrypts it, and then uses it to decrypt the data.

Can I add additional key encryption keys (KEKs) to the envelope encryption (p. 2) scheme?

The AWS Encryption SDK encrypts the data you pass to the encryption methods with a unique data encryption key (DEK), and then encrypts that DEK with a key encryption key (KEK) called a *master key*. You can encrypt the DEK with additional master keys to add redundancy, in case one of the master keys is unavailable. For a code sample (Java) that demonstrates how you can do this, see Encrypting and Decrypting Byte Streams with Multiple Master Key Providers (p. 19).

What is the message format used by the AWS Encryption SDK?

The encryption operations in the AWS Encryption SDK return a single data structure, or *message*, that contains the encrypted data and the encrypted data key. The message format consists of at least two parts, a *header* and a *body*. In some cases the message format consists of a third part called a *footer*. The message header contains the encrypted data key and information about how the message body is formed. The message body contains the encrypted data. The message footer contains a signature that authenticates the message header and message body. For more information, see Message Format (p. 7).

Which cryptographic algorithms are supported by the AWS Encryption SDK, and which one is the default?

The SDK uses the Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM), known as AES-GCM. The SDK supports encryption key lengths of 256 bits, 192 bits, and 128 bits. In all cases, the length of the initialization vector (IV) is 12 bytes; the length of the authentication tag is 16 bytes.

The AWS Encryption SDK supports the following encryption algorithms. By default, the SDK uses the first algorithm in the list.

1. ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384
2. ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384
3. ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256_ECDSA_P256
4. ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256
5. ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA256
6. ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256
7. ALG_AES_256_GCM_IV12_TAG16_NO_KDF
8. ALG_AES_192_GCM_IV12_TAG16_NO_KDF
9. ALG_AES_128_GCM_IV12_TAG16_NO_KDF

The algorithms belong to one of three categories, which are described in more detail in the following list.

Signed algorithms

The algorithms numbered 1 through 3 in the preceding list are the *signed algorithms*. The signed algorithms use the data encryption key as an input to the HMAC key derivation function (HKDF) to derive the AES-GCM encryption key. These algorithms also add an Elliptic Curve Digital Signature Algorithm (ECDSA) signature. When the key length is 256 bits or 192 bits, the HKDF uses SHA-384 and the ECDSA signature uses the secp384r1 curve. When the key length is 128 bits, the HKDF uses SHA-256 and the ECDSA signature uses the secp256r1 curve.

These algorithms help protect against accidental reuse of a data encryption key, and the ECDSA signature helps provide stronger authenticity and non-repudiation of the original data. Use these algorithms when authorized users of a master key—that is, the users who encrypt data and those who decrypt data—are not equally trusted. These algorithms help protect against some users of the master key attempting to impersonate other users of the master key.

Standard algorithms

The algorithms numbered 4 through 6 in the preceding list are the *standard algorithms*. The standard algorithms are like the signed algorithms but without the ECDSA signature.

These algorithms help protect against accidental reuse of a data encryption key. These algorithms are appropriate for standard use cases in which all authorized users of a master key—that is, the users who encrypt data and those who decrypt data—are equally trusted.

Compatibility algorithms

The algorithms numbered 7 through 9 in the preceding list are the *compatibility algorithms*. The compatibility algorithms do not use a key derivation function (KDF) to derive the encryption key; they use the data encryption key as the AES-GCM encryption key.

The use of these algorithms is not recommended; the SDK provides them for compatibility reasons only.