AWS Encryption SDK

Developer Guide

AWS Encryption SDK: Developer Guide

Table of Contents

What Is the AWS Encryption SDK? 1
Where to find more information 2
How the SDK Works 2
Symmetric Key Encryption 2
Envelope Encryption 3
AWS Encryption SDK Encryption Workflows 4
Concepts 5
Data Keys 6
Master key 6
Master key operations: Generate, Encrypt, Decrypt 6
Master key provider 7
Cryptographic Materials Manager 7
Algorithm Suite 7
Encryption Context 8
Encrypted Message 8
Getting Started 9
Supported Algorithm Suites 10
Recommended: AES-GCM with Key Derivation and Signing 10
Other Supported Algorithm Suites 11
Programming Languages 12
Java 12
Prerequisites 12
Installation 13
Example Code 13
Python 20
Prerequisites 20
Installation 20
Example Code 21
Command Line Interface 26
Installing the CLI 27
How to Use the CLI 29
Examples 36
Syntax and Parameter Reference 49
Data Key Caching 55
How to Implement Data Key Caching 55
Implement Data Key Caching: Step-by-Step 56
Data Key Caching Example: Encrypt a String 58
Setting Cache Security Thresholds 60
Data Key Caching Details 61
How Data Key Caching Works 62
Creating a Cryptographic Materials Cache 64
Creating a Caching Cryptographic Materials Manager 65
What Is in a Data Key Cache Entry? 65
Encryption Context: How to Select Cache Entries 66
Data Key Caching Example 66
LocalCryptoMaterialsCache Results 67
Java Example 68
Python Example 72
AWS CloudFormation Template 75
Frequently Asked Questions 79
Reference 82
Message Format Reference 82
Header Structure 83
Body Structure 87

Footer Structure 89
Body AAD Reference 90
Message Format Examples 91
Non-Framed Data 91
Framed Data 93
Algorithms Reference 96
Initialization Vector Reference 98
Document History 99
Recent Updates 99
Earlier Updates 99

What Is the AWS Encryption SDK?

The AWS Encryption SDK is an encryption library that helps make it easier for you to implement encryption best practices in your application. It enables you to focus on the core functionality of your application, rather than on how to best encrypt and decrypt your data.

The AWS Encryption SDK answers questions like the following for you:

- Which encryption algorithm should I use?
- How, or in which mode, should I use that algorithm?
- How do I generate the encryption key?
- How do I protect the encryption key, and where should I store it?
- How can I make my encrypted data portable?
- How do I ensure that the intended recipient can read my encrypted data?
- How can I ensure my encrypted data is not modified between the time it is written and when it is read?

Without the AWS Encryption SDK, you might spend more effort on building an encryption solution than on the core functionality of your application. The AWS Encryption SDK answers these questions by providing the following things.

A Default Implementation that Adheres to Cryptography Best Practices

By default, the AWS Encryption SDK generates a unique data key for each data object that it encrypts. This follows the cryptography best practice of using unique data keys for each encryption operation.

The AWS Encryption SDK encrypts your data using a secure, authenticated, symmetric key algorithm. For more information, see *Supported Algorithm Suites* (p. 10).

A Framework for Protecting Data Keys with Master Keys

The AWS Encryption SDK protects the data keys that encrypt your data by encrypting them under one or more master keys. By providing a framework to encrypt data keys with more than one master key, the AWS Encryption SDK helps make your encrypted data portable.

For example, you can encrypt data under multiple AWS Key Management Service (AWS KMS) customer master keys (CMKs), each in a different AWS Region. Then you can copy the encrypted data to any of the regions and use the CMK in that region to decrypt it. You can also encrypt data under a CMK in AWS KMS and a master key in an on-premises HSM, enabling you to later decrypt the data even if one of the options is unavailable.

A Formatted Message that Stores Encrypted Data Keys with the Encrypted Data

The AWS Encryption SDK stores the encrypted data and encrypted data key together in an encrypted message (p. 8) that uses a defined data format. This means you don't need to keep track of or protect the data keys that encrypt your data because the AWS Encryption SDK does it for you.

With the AWS Encryption SDK, you define a master key provider (p. 7) that returns one or more master keys (p. 6). Then you encrypt and decrypt your data using straightforward methods provided by the AWS Encryption SDK. The AWS Encryption SDK does the rest.

Where to find more information

If you're looking for more information about the AWS Encryption SDK and client-side encryption, try these sources.

- To get started quickly, see Getting Started (p. 9).
- For information about how this SDK works, see How the SDK Works (p. 2).
- For help with the terms and concepts used in this SDK, see Concepts in the AWS Encryption SDK (p. 5).
- For detailed technical information, see the *Reference* (p. 82).
- For answers to your questions about using the AWS Encryption SDK, read and post on the AWS Crypto Tools Discussion Forum.

For information about implementations of the AWS Encryption SDK in different programming languages.

- **Java**: See AWS Encryption SDK for Java (p. 12), the AWS Encryption SDK Javadoc, and the aws-encryption-sdk-java repository on GitHub.
- **Python**: See AWS Encryption SDK for Python (p. 20), the AWS Encryption SDK Python documentation, and the aws-encryption-sdk-python repository on GitHub.
- **Command Line Interface**: See AWS Encryption SDK Command Line Interface (p. 26), Read the Docs for the AWS Encryption CLI, and the aws-encryption-sdk-cli repository on GitHub.

If you have questions or comments about this guide, let us know! Choose the feedback link in the lower-right corner of the page or the GitHub link in the upper-right corner of the page. You can also file an issue in the aws-encryption-sdk-docs GitHub repository for this guide.

The AWS Encryption SDK is provided free of charge under the Apache license.

How the AWS Encryption SDK Works

The AWS Encryption SDK uses *envelope encryption* to protect your data and the corresponding data keys. For more information, see the following topics.

Topics

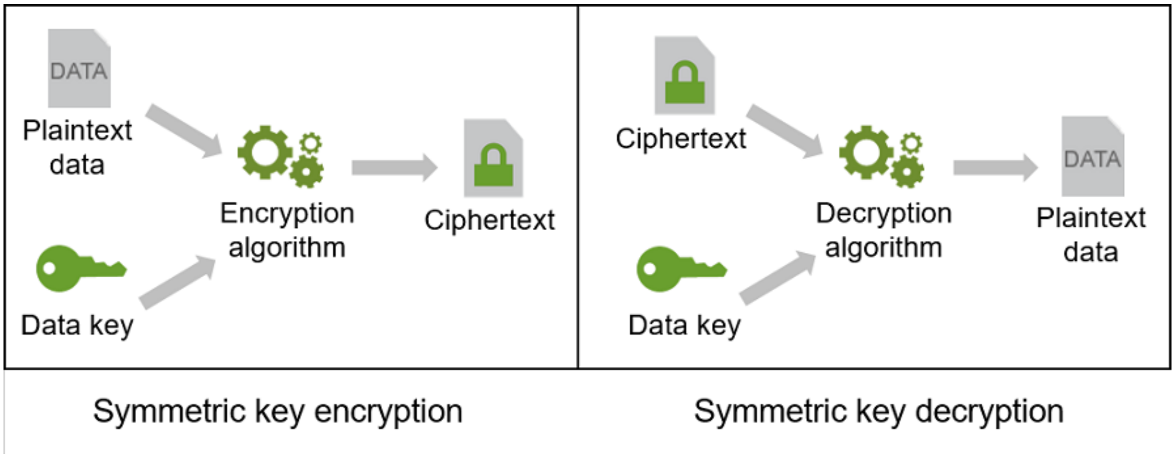
- Symmetric Key Encryption (p. 2)
- Envelope Encryption (p. 3)
- AWS Encryption SDK Encryption Workflows (p. 4)

Symmetric Key Encryption

To encrypt data, the AWS Encryption SDK provides raw data, known as *plaintext data*, and a data key to an encryption algorithm. The encryption algorithm uses those inputs to encrypt the data. Then, the AWS Encryption SDK returns an encrypted message (p. 8) that includes the encrypted data and an encrypted copy of the data key.

To decrypt the encrypted message, the AWS Encryption SDK provides the encrypted message to a decryption algorithm that uses those inputs to return the plaintext data.

Because the same data key is used to encrypt and decrypt the data, the operations are known as *symmetric key* encryption and decryption. The following figure shows symmetric key encryption and decryption in the AWS Encryption SDK.



Envelope 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 do this, you need another encryption key, known as a master key (p. 6). This practice of using a master key to encrypt data keys is known as *envelope encryption*. Some of the benefits of envelope encryption include the following.

Protecting Data Keys

When you encrypt a data key, you don't have to worry about where to store it because the data key is inherently protected by encryption. You can safely store the encrypted data key with the encrypted data. The AWS Encryption SDK does this for you. It saves the encrypted data and the encrypted data key together in an encrypted message (p. 8).

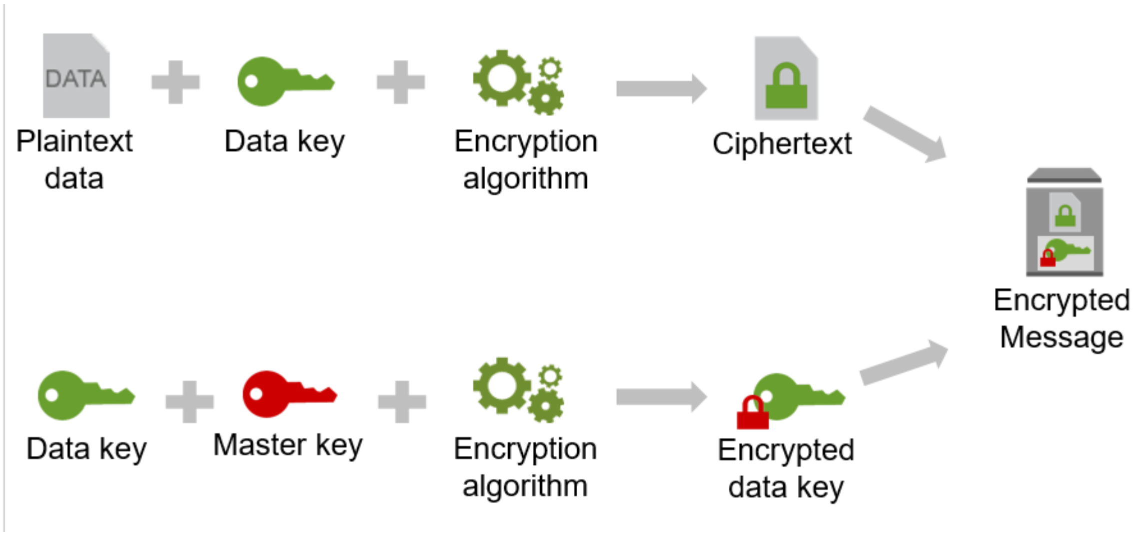
Encrypting the Same Data Under Multiple Master Keys

Encryption operations can be time-consuming, particularly when the data being encrypted are large objects. Instead of reencrypting raw data multiple times with different keys, you can reencrypt only the data keys that protect the raw data.

Combining the Strengths of Multiple Algorithms

In general, symmetric key encryption algorithms are faster and produce smaller ciphertexts than asymmetric or *public key encryption*. 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 AWS Encryption SDK uses envelope encryption. It encrypts your data with a data key. Then, it encrypts the data key with a master key. The AWS Encryption SDK returns the encrypted data and the encrypted data keys in a single encrypted message, as shown in the following diagram.



If you have multiple master keys, each of them can encrypt the plaintext data key. Then, the AWS Encryption SDK returns an encrypted message that contains the encrypted data and the collection of encrypted data keys. Any one of the master keys can decrypt one of the encrypted data keys, which can then decrypt the data.

When you use envelope encryption, you must protect your master keys from unauthorized access. You can do this in one of the following ways:

- Use a web service designed for this purpose, such as AWS Key Management Service (AWS KMS).
- Use a hardware security module (HSM) such as those offered by AWS CloudHSM.
- Use your existing key management tools.

If you don't have a key management system, we recommend AWS KMS. The AWS Encryption SDK integrates with AWS KMS to help you protect and use your master keys. You can also use the AWS Encryption SDK with other master key providers, including custom ones that you define. Even if you don't use AWS, you can still use this AWS Encryption SDK.

AWS Encryption SDK Encryption Workflows

The workflows in this section explain how the SDK encrypts data and decrypts encrypted messages (p. 8). They show how the SDK uses the components that you create, including the cryptographic materials manager (p. 7) (CMM), master key provider (p. 7), and master key (p. 6), to respond to encryption and decryption requests from your application.

How the SDK Encrypts Data

The SDK provides methods that encrypt strings, byte arrays, and byte streams. For code examples showing calls to encrypt and decrypt strings and byte streams in each supported programming languages, see the examples in the Programming Languages (p. 12) section.

1. Your application passes plaintext data to one of the encryption methods.

   To indicate the source of the data keys (p. 6) that you want to use to encrypt your data, your request specifies a cryptographic materials manager (CMM) or a master key provider. (If you specify a master key provider, the AWS Encryption SDK creates a default CMM that interacts with your chosen master key provider.)
2. The encryption method asks the CMM for data keys (and related cryptographic material).

3. The CMM gets a master key (p. 6) from its master key provider.

   Note
   If you are using AWS Key Management Service (AWS KMS), the KMS master key object that is returned identifies the CMK, but the actual CMK never leaves the AWS KMS service.
4. The CMM asks the master key to generate a data key. The master key returns two copies of the data key, one in plaintext and one encrypted under the master key.
5. The CMM returns the plaintext and encrypted data keys to the encryption method.
6. The encryption method uses the plaintext data key to encrypt the data, and then discards the plaintext data key.
7. The encryption method returns an encrypted message (p. 8) that contains the encrypted data and the encrypted data key.

How the SDK Decrypts an Encrypted Message

The SDK provides methods that decrypt an encrypted message and return plaintext strings, byte arrays, or byte streams. For code examples in each supported programming languages, see the examples in the Programming Languages (p. 12) section.

1. Your application passes an encrypted message to a decryption method.

   To indicate the source of the data keys (p. 6) that were used to encrypt your data, your request specifies a cryptographic materials manager (CMM) or a master key provider. (If you specify a master key provider, the AWS Encryption SDK creates a default CMM that interacts with the specified master key provider.)
2. The decryption method asks the CMM for cryptographic materials to decrypt the encrypted message. It passes in information from the message header, including the encrypted data keys.
3. To get decryption materials, the Default CMM asks its master key provider for a master key that can decrypt the encrypted data key. It uses the master key to decrypt the encrypted data key. Then, it returns the decryption materials (including the plaintext data key) to the decryption method. Other CMMs might use different techniques to get the decryption materials.
4. The decryption method uses the plaintext data key to decrypt the data, then discards the plaintext data key.
5. The decryption method returns the plaintext data.

Concepts in the AWS Encryption SDK

This section introduces the concepts used in the AWS Encryption SDK. The AWS Encryption SDK is designed so that you can use the default implementations of the components without detailed knowledge about their functionality. This section is provided as a glossary and reference.

Topics

- Data Keys (p. 6)
- Master key (p. 6)
- Master key operations: Generate, Encrypt, Decrypt (p. 6)
- Master key provider (p. 7)
- Cryptographic Materials Manager (p. 7)
- Algorithm Suite (p. 7)
- Encryption Context (p. 8)
- Encrypted Message (p. 8)

Data Keys

A *data key* consists of cryptographic material. It is the secret key that protects the data that you encrypt.

Data keys are generated by master keys (p. 6). You do not need to implement or extend data keys to use the AWS Encryption SDK. When a master key generates a data key, it returns two copies of the data key; one in plaintext and one that is encrypted by the master key that generated it. The plaintext data key can be encrypted by multiple master keys, each of which returns an encrypted copy of the data key. Every encrypted data key is associated with the master key that encrypted it and the master key provider (p. 7) that supplied the master key.

When you encrypt data in the AWS Encryption SDK, the encrypted data keys are stored in an encrypted message (p. 8) along with the encrypted data.

In the AWS Encryption SDK, we distinguish *data keys* from *data encryption keys*. Several of the supported algorithm suites (p. 7), including the default suite, use a key derivation function that prevents the data key from hitting its cryptographic limits. The key derivation function takes the data key as input and returns a data encryption key that is actually used to encrypt the data. For this reason, we often say that data is encrypted "under" a data key rather than "by" the data key.

Master key

A *master key* encrypts, decrypts, and generates data keys (p. 2).

The AWS Encryption SDK represents master keys as abstract classes or interfaces so you can implement the master key operations in the way that best meets the security requirements of your organization. For example, although they are called "keys," master keys might not have their own cryptographic material. Also, unlike data keys, whose use and algorithm suite (p. 7) are strictly defined by AWS Encryption SDK, master keys can use any algorithm suite or implementation.

Master keys are instrumental to envelope encryption (p. 3). In envelope encryption, one master key generates and encrypts a data key that is used to encrypt data. Other master keys then re-encrypt the plaintext data key. As a result, any master key is sufficient to decrypt the data.

Each master key is associated with one master key provider (p. 7) that returns one or more master keys to the caller.

The AWS Encryption SDK provides several commonly used master keys, such as AWS Key Management Service (AWS KMS) customer master keys (CMKs), raw AES-GCM (Advanced Encryption Standard / Galois Counter Mode) keys, and RSA keys. You can implement your own master keys for other cryptographic algorithms and services. For example, you could implement master keys backed by implementations of Elliptical Curve Integrated Encryption Scheme (ECIES), Key Management Interoperability Program (KMIP), tokenization services, or other proprietary systems.

Master key operations: Generate, Encrypt, Decrypt

Master keys in the AWS Encryption SDK generate, encrypt, and decrypt data keys (p. 6). You write methods to perform these operations when you create a master key, but your application does not call the methods directly. The SDK calls them when you ask it to encrypt or decrypt data.

You can implement the master key methods in the way that works best for your organization. For example, when asked to generate a data key, a master key can create or return a key in any way that fulfills the requirements of the algorithm suite that they use. Master keys can generate data keys locally or remotely. They can derive the keys algorithmically, call a service that generates the cryptographic material, or return previously-generated data keys. The SDK requires only that they return a valid data key object.

Also, although master keys must implement all three methods, you can create master keys that actually perform only one or two of the three operations. Calls to the remaining methods just fail or return errors. These limited master keys might be useful in a system with strict access controls that do not let the same users encrypt and decrypt data.

All master key operations take an encryption context (p. 8) as input. For optimal security, master key operations that encrypt data keys should cryptographically bind the encryption context to the encrypted data so that changing any key or value in the encryption context invalidates the encryption. Master key operations that decrypt should verify the encryption context and fail unless they include the same encryption context used to encrypt. The encryption context is most useful when there are users who have permission to decrypt, but not encrypt.

Master key provider

A *master key provider* returns objects that represent master keys. Each master key is associated with one master key provider, but a master key provider typically provides multiple master keys.

The simplest master key provider always returns the same master key (p. 6). In fact, master keys are implemented as master keys providers that only return themselves. More complex master key providers might use key rotation, the encryption context, application permissions, and other factors to select master keys from among the set they can provide.

Many master keys providers wrap or extend other master key providers to customize their behavior and functionality. For example, a custom master key provider might select a master key provider from a collection, delegate requests, and combine their results.

Cryptographic Materials Manager

The cryptographic materials manager (CMM) gets the cryptographic materials that are used to encrypt and decrypt data. The *cryptographic materials* include plaintext and encrypted data keys, and an optional message signing key. You can use the Default CMM that the AWS Encryption SDK provides (`DefaultCryptoMaterialsManager`) or write a custom CMM.

Each Default CMM is associated with a master key provider (p. 7). When it gets a materials request, the Default CMM gets master keys from its master key provider and uses them to generate the requested cryptographic material. This might involve a call to a cryptographic service, such as AWS Key Management Service (AWS KMS).

In each call to encrypt or decrypt data, you specify a CMM or a master key provider. This lets you choose a particular set of master keys for the operation. You can create a CMM explicitly and specify its master key provider, but that is not required. If you specify a master key provider in an encryption request, the SDK creates a Default CMM for the master key provider.

Because the CMM acts as a liaison between the SDK and a master key provider, it is an ideal point for customization and extension, such as support for policy enforcement and caching.

Algorithm Suite

The AWS Encryption SDK supports several (p. 12) algorithm suites (p. 10), all of which use Advanced Encryption Standard (AES) as the primary algorithm, and combine it with other algorithm and values.

The AWS Encryption SDK establishes a recommended algorithm suite as the default for all encryption operations. The default might change as standards and best practices improve. You can specify an alternate algorithm suite in requests to encrypt data or when creating a cryptographic materials manager (CMM) (p. 7), but unless an alternate is required for your situation, it is best to use the default. The current default is AES-GCM with an HMAC-based extract-and-expand key derivation function (HKDF), Elliptic Curve Digital Signature Algorithm (ECDSA) signing, and a 256-bit encryption key.

If you specify an algorithm suite, we recommend an algorithm suite that uses a key derivation function and a message signing algorithm. Algorithm suites that have neither feature are supported only for backward compatibility.

Encryption Context

To improve the security of your cryptographic operations, use an encryption context in all requests to encrypt data. The encryption context is optional, but recommended.

An *encryption context* is a set of key–value pairs that contain arbitrary nonsecret data. The encryption context can contain any data you choose, but it typically consists of data that is useful in logging and tracking, such as data about the file type, purpose, or ownership.

In requests to encrypt data, you can include an encryption context along with the plaintext data and a master key provider. The AWS Encryption SDK cryptographically binds the encryption context to the encrypted data so that the same encryption context is required to decrypt the data. The AWS Encryption SDK also includes the encryption context in the encrypted message (p. 8) that it returns, along with the encrypted data and data keys. The encryption context in the encrypted message always includes the encryption context that you specified in the encryption request, along with elements that the operation might add, such as a public signing key.

To decrypt the data, you pass in the encrypted message. Because the AWS Encryption SDK can extract the encryption context from the message, you do not need to pass it in separately. After decrypting the data, the AWS Encryption SDK returns a result that includes that encryption context along with the plaintext data. The functions in your application that decrypt data should always verify that the encryption context in the decrypt result includes the values that you expect before it returns the plaintext data.

When choosing an encryption context, remember that it is not a secret. The encryption context is displayed in plaintext in the header of the encrypted message (p. 8) that the SDK returns. If you are using AWS Key Management Service, the encryption context also might appear in plaintext in audit records and logs, such as AWS CloudTrail.

Encrypted Message

Encrypt operations in the AWS Encryption SDK return an encrypted message and decrypt operations take an encrypted message as input. An *encrypted message*, a formatted data structure (p. 82) that includes the encrypted data along with encrypted copies of the data keys, the algorithm ID, and, optionally, an encryption context and a message signature.

Combining the encrypted data and its encrypted data keys streamlines the decryption operation and frees you from having to store and manage encrypted data keys independently of the data that they encrypt.

For technical information about the encrypted message, see Encrypted Message Format (p. 82).

Getting Started with the AWS Encryption SDK

To use the AWS Encryption SDK, you need a master key provider (p. 7). If you don't have one, we recommend using AWS Key Management Service (AWS KMS). Many of the code samples in the AWS Encryption SDK require an AWS KMS customer master key (CMK).

To interact with AWS KMS, you need to use the AWS SDK for your preferred programming language, such as the AWS SDK for Java or the AWS SDK for Python (Boto). The AWS Encryption SDK client library works with the AWS SDKs to support master keys stored in AWS KMS.

To prepare to use the AWS Encryption SDK with AWS KMS

1. Create an AWS account. To learn how, see How do I create and activate a new Amazon Web Services account? in the AWS Knowledge Center.
2. Create a customer master key (CMK) in AWS KMS. To learn how, see Creating Keys in the *AWS Key Management Service Developer Guide*.

   Tip
   To use the CMK programmatically, you will need the ID or Amazon Resource Name (ARN) of the CMK. For help finding the ID or ARN of a CMK, see Viewing Keys in the *AWS Key Management Service Developer Guide*.
3. Create an IAM user with an access key. To learn how, see Creating IAM Users in the *IAM User Guide*. When you create the user, for **Access type**, choose **Programmatic access**. After you create the user, choose **Download.csv** to save the AWS access key that represents your user credentials. Store the file in a secure location.

   We recommend that you use AWS Identity and Access Management (IAM) access keys instead of AWS (root) account access keys. IAM lets you securely control access to AWS services and resources in your AWS account. For detailed best practice guidance, see Best Practices for Managing AWS Access Keys

   The `Download.csv` file contains an AWS access key ID and a secret access key that represents the AWS credentials of the user that you created. When you write code without using an AWS SDK, you use your access key to sign your requests to AWS. The signature assures AWS that the request came from you unchanged. However, when you use an AWS SDK, such as the AWS SDK for Java, the SDK signs all requests to AWS for you.
4. Set your AWS credentials using the instructions for Java or Python and the AWS access key in the `Download.csv` file that you downloaded in Step 3.

   This procedure allows AWS SDKs to sign requests to AWS for you. Code samples in the AWS Encryption SDK that interact with AWS KMS assume that you have completed this step.
5. Download and install the AWS Encryption SDK. To learn how, see the installation instructions for the programming language (p. 12) that you want to use.

Supported Algorithm Suites in the AWS Encryption SDK

An *algorithm suite* is a collection of cryptographic algorithms and related values. Cryptographic systems use the algorithm implemenation to generate the ciphertext message.

The AWS Encryption SDK algorithm suite uses the Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM), known as AES-GCM, to encrypt raw data. The SDK supports 256-bit, 192-bit, and 128-bit encryption keys. The length of the initialization vector (IV) is always 12 bytes; the length of the authentication tag is always 16 bytes.

The SDK implements AES-GCM in one of three ways. By default, the SDK uses AES-GCM with an HMAC-based extract-and-expand key derivation function (HKDF), signing, and a 256-bit encryption key.

Recommended: AES-GCM with Key Derivation and Signing

In the recommended algorithm suite, the SDK uses the data encryption key as an input to the HMAC-based extract-and-expand key derivation function (HKDF) to derive the AES-GCM encryption key. The SDK also adds an Elliptic Curve Digital Signature Algorithm (ECDSA) signature. By default, the SDK uses this algorithm suite with a 256-bit encryption key.

The HKDF helps you avoid accidental reuse of a data encryption key.

This algorithm suite uses ECDSA and a message signing algorithm (SHA-384 or SHA-256). ECDSA is used by default, even when it is not specified by the policy for the underlying master key. Message signing verifies the identity of the message sender and adds message authenticity to the envelope encrypted data. It is particularly useful when the authorization policy for a master key allows one set of users to encrypt data and a different set of users to decrypt data.

The following table lists the variations of the recommended algorithm suites.

AWS Encryption SDK Algorithm Suites

Algorithm Name	Data Encryption Key Length (in bits)	Algorithm Mode	Key Derivation Algorithm	Signature Algorithm
AES	256	GCM	HKDF with SHA-384	ECDSA with P-384 and SHA-384
AES	192	GCM	HKDF with SHA-384	ECDSA with P-384 and SHA-384
AES	128	GCM	HKDF with SHA-256	ECDSA with P-256 and SHA-256

Other Supported Algorithm Suites

The AWS Encryption SDK supports the alternate algorithm suites for backward compatibility, although we do not recommend them. If you cannot use an algorithm suite with HKDF and signing, we recommend an algorithm suite with HKDF over one that lacks both elements.

AES-GCM with Key Derivation Only

This algorithm suite uses a key derivation function, but lacks the ECDSA signature that provides authenticity and nonrepudiation. Use this suite when the users who encrypt data and those who decrypt it are equally trusted.

AES-GCM without Key Derivation or Signing

This algorithm suite uses the data encryption key as the AES-GCM encryption key, instead of using a key derivation function to derive a unique key. We discourage using this suite to generate ciphertext, but the SDK supports it for compatibility reasons.

For more information about how these suites are represented and used in the library, see the section called “Algorithms Reference” (p. 96).

AWS Encryption SDK Programming Languages

The AWS Encryption SDK is available for the following programming languages. For more information, see the corresponding topic.

Topics

- AWS Encryption SDK for Java (p. 12)
- AWS Encryption SDK for Python (p. 20)
- AWS Encryption SDK Command Line Interface (p. 26)

AWS Encryption SDK for Java

This topic explains how to install and use the AWS Encryption SDK for Java. For details about programming with the SDK, see the aws-encryption-sdk-java repository on GitHub and the Javadoc for the AWS Encryption SDK.

Topics

- Prerequisites (p. 12)
- Installation (p. 13)
- AWS Encryption SDK for Java Example Code (p. 13)

Prerequisites

Before you install the AWS Encryption SDK for Java, be sure you have the following prerequisites.

A Java development environment

You will need Java 8 or later. On the Oracle website, go to Java SE Downloads, and then download and install the Java SE Development Kit (JDK).

If you use the Oracle JDK, you must also download and install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files.

Bouncy Castle

Bouncy Castle provides a cryptography API for Java. If you don't have Bouncy Castle, go to Bouncy Castle latest releases to download the provider file that corresponds to your JDK.

If you use Apache Maven, Bouncy Castle is available with the following dependency definition.

```
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcprov-ext-jdk15on</artifactId>
  <version>1.58</version>
</dependency>
```

AWS SDK for Java (Optional)

Although you don't need the AWS SDK for Java to use the AWS Encryption SDK for Java, you do need it to use AWS Key Management Service (AWS KMS) as a master key provider, and to use some of the example Java code (p. 13) in this guide. For more information about installing and configuring the AWS SDK for Java, see AWS SDK for Java.

Installation

You can install the AWS Encryption SDK for Java in the following ways.

Manually

To install the AWS Encryption SDK for Java, clone or download the aws-encryption-sdk-java GitHub repository.

Using Apache Maven

The AWS Encryption SDK for Java is available through Apache Maven with the following dependency definition.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>1.3.1</version>
</dependency>
```

After you install the SDK, get started by looking at the example Java code (p. 13) in this guide and the Javadoc on GitHub.

AWS Encryption SDK for Java Example Code

The following examples show you how to use the AWS Encryption SDK for Java to encrypt and decrypt data.

Topics

- Encrypting and Decrypting Strings (p. 13)
- Encrypting and Decrypting Byte Streams (p. 15)
- Encrypting and Decrypting Byte Streams with Multiple Master Key Providers (p. 17)

Encrypting and Decrypting Strings

The following example shows you how to use the AWS Encryption SDK to encrypt and decrypt strings.

This example uses an AWS Key Management Service (AWS KMS) customer master key (CMK) as the master key. For help creating a key, see Creating Keys in the *AWS Key Management Service Developer Guide*.

To find the Amazon Resource name (ARN) of an existing CMK, go to the **Encryption keys** section of the AWS Management Console, select the region, and then click the CMK alias. You can also use the AWS KMS ListKeys operation. For details, see Viewing Keys in the AWS Key Management Service Developer Guide.

```
/*
```

```
 * Copyright 2017 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
 *
 * http://aws.amazon.com/apache2.0
 *
 * 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 key
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your KMS customer
 master
 *    key (CMK), see 'Viewing Keys' at http://docs.aws.amazon.com/kms/latest/
developerguide/viewing-keys.html
 * <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 an associated encryption context
        // to protect integrity. This sample uses placeholder values.
        //
        // For more information see:
        // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-Integrity-
of-Your-Encrypted-Data-by-Using-AWS-Key-Management
        final Map<String, String> context = Collections.singletonMap("Example", "String");

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

```
        // Decrypt the data
        final CryptoResult<String, KmsMasterKey> decryptResult = crypto.decryptString(prov,
 ciphertext);

        // Before returning the plaintext, verify that the customer master key that
        // was used in the encryption operation was the one supplied to the master key
 provider.
        if (!decryptResult.getMasterKeyIds().get(0).equals(keyArn)) {
            throw new IllegalStateException("Wrong key ID!");
        }

        // Also, verify that the encryption context in the result contains the
        // encryption context supplied to the encryptString method. Because the
        // SDK can add values to the encryption context, don't require that
        // the entire context matches.
        for (final Map.Entry<String, String> e : context.entrySet()) {
            if (!e.getValue().equals(decryptResult.getEncryptionContext().get(e.getKey())))
 {
                throw new IllegalStateException("Wrong Encryption Context!");
            }
        }

        // Now we can return the plaintext data
        System.out.println("Decrypted: " + decryptResult.getResult());
    }
}
```

Encrypting and Decrypting Byte Streams

The following example shows you how to 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 2017 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
 *
 * http://aws.amazon.com/apache2.0
 *
 * 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 file containing plaintext data to encrypt
 * </ol>
 *
 * <p>
 * This program demonstrates using a standard 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 this example, we generate a random key. In practice,
        // you would get a key from an existing store
        SecretKey cryptoKey = retrieveEncryptionKey();

        // Create a JCE master key provider using the random key and an AES-GCM encryption
 algorithm
        JceMasterKey masterKey = JceMasterKey.getInstance(cryptoKey, "Example",
 "RandomKey", "AES/GCM/NoPadding");

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

        // Create an encryption context to identify this ciphertext
        Map<String, String> context = Collections.singletonMap("Example", "FileStreaming");

        // Because the file might be to large to load into memory, we stream the data,
 instead of
        //loading it all at once.
        FileInputStream in = new FileInputStream(srcFile);
        CryptoInputStream<JceMasterKey> encryptingStream =
 crypto.createEncryptingStream(masterKey, in, context);

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

        // Decrypt the file. Verify the encryption context before returning the plaintext.
        in = new FileInputStream(srcFile + ".encrypted");
        CryptoInputStream<JceMasterKey> decryptingStream =
 crypto.createDecryptingStream(masterKey, in);
        // Does it contain the expected encryption context?
        if
 (!"FileStreaming".equals(decryptingStream.getCryptoResult().getEncryptionContext().get("Example")))
 {
            throw new IllegalStateException("Bad encryption context");
        }

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

```
        out.close();
    }

    /**
     * In practice, this key would be saved in a secure location.
     * For this demo, we generate a new random key for each operation.
     */
    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 shows you how to use the AWS Encryption SDK with more than one master key provider. Using more than one master key provider creates redundancy if 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 2017 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
 *
 * http://aws.amazon.com/apache2.0
 *
 * 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 using both KMS and an asymmetric key pair.
 *
 * <p>
 * Arguments:
 * <ol>
```

```
 * <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your KMS customer
 master
 *    key (CMK), see 'Viewing Keys' at http://docs.aws.amazon.com/kms/latest/
developerguide/viewing-keys.html
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 *
 * You might use AWS Key Management Service (KMS) for most encryption and decryption
 operations, but
 * still want the option of decrypting your data offline independently of KMS. This sample
 * demonstrates one way to do this.
 *
 * The sample encrypts data under both a KMS customer master key (CMK) and an "escrowed"
 RSA key pair
 * so that either key alone can decrypt it. You might commonly use the KMS CMK for
 decryption. However,
 * at any time, you can use the private RSA key to decrypt the ciphertext independent of
 KMS.
 *
 * This sample uses the JCEMasterKey class to generate a RSA public-private key pair
 * and saves the key pair in memory. In practice, you would store the private key in a
 secure offline
 * location, such as an offline HSM, and distribute the public key to your development
 team.
 *
 */
public class EscrowedEncryptExample {
    private static PublicKey publicEscrowKey;
    private static PrivateKey privateEscrowKey;

    public static void main(final String[] args) throws Exception {
        // This sample generates a new random key for each operation.
        // In practice, you would distribute the public key and save the private key in
 secure
        // storage.
        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 fileName) throws
 Exception {
        // Encrypt with the KMS CMK and the escrowed public key
        // 1. Instantiate the SDK
        final AwsCrypto crypto = new AwsCrypto();

        // 2. Instantiate a KMS master key provider
        final KmsMasterKeyProvider kms = new KmsMasterKeyProvider(kmsArn);

        // 3. Instantiate a JCE master key provider
        // Because the user does not have access to the private escrow key,
        // they pass in "null" for the private key parameter.
        final JceMasterKey escrowPub = JceMasterKey.getInstance(publicEscrowKey, null,
 "Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

        // 4. Combine the providers into a single master key provider
        final MasterKeyProvider<?> provider =
 MultipleProviderFactory.buildMultiProvider(kms, escrowPub);
```

```
        // 5. Encrypt the file
        // To simplify the code, we omit the encryption context. Production code should
always
        // use an encryption context. For an example, see the other SDK samples.
        final FileInputStream in = new FileInputStream(fileName);
        final FileOutputStream out = new FileOutputStream(fileName + ".encrypted");
        final CryptoOutputStream<?> encryptingStream =
crypto.createEncryptingStream(provider, out);

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

    private static void standardDecrypt(final String kmsArn, final String fileName) throws
Exception {
        // Decrypt with the KMS CMK and the escrow public key. You can use a combined
provider,
        // as shown here, or just the KMS master key provider.

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

        // 2. Instantiate a KMS master key provider
        final KmsMasterKeyProvider kms = new KmsMasterKeyProvider(kmsArn);

        // 3. Instantiate a JCE master key provider
        // Because the user does not have access to the private escrow
        // key, they pass in "null" for the private key parameter.
        final JceMasterKey escrowPub = JceMasterKey.getInstance(publicEscrowKey, null,
"Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

        // 4. Combine the providers into a single master key provider
        final MasterKeyProvider<?> provider =
MultipleProviderFactory.buildMultiProvider(kms, escrowPub);

        // 5. Decrypt the file
        // To simplify the code, we omit the encryption context. Production code should
always
        // use an encryption context. For an example, see the other SDK samples.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".decrypted");
        final CryptoOutputStream<?> decryptingStream =
crypto.createDecryptingStream(provider, out);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();
    }

    private static void escrowDecrypt(final String fileName) throws Exception {
        // You can decrypt the stream using only the private key.
        // This method does not call KMS.

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

        // 2. Instantiate a JCE master key
        // This method call uses the escrowed private key, not null
        final JceMasterKey escrowPriv = JceMasterKey.getInstance(publicEscrowKey,
privateEscrowKey, "Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

        // 3. Decrypt the file
         // To simplify the code, we omit the encryption context. Production code should
always
```

```
        // use an encryption context. For an example, see the other SDK samples.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".deescrowed");
        final CryptoOutputStream<?> decryptingStream =
 crypto.createDecryptingStream(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();

    }
}
```

AWS Encryption SDK for Python

This topic explains how to install and use the AWS Encryption SDK for Python. For details about programming with the SDK, see the aws-encryption-sdk-python repository on GitHub and the Python documentation for the AWS Encryption SDK for Python.

Topics

- Prerequisites (p. 20)
- Installation (p. 20)
- AWS Encryption SDK for Python Example Code (p. 21)

Prerequisites

Before you install the AWS Encryption SDK for Python, be sure you have the following prerequisites.

A supported version of Python

To use this SDK, you need Python 2.7, or Python 3.4 or later. To download Python, see Python downloads.

The pip installation tool for Python

If you have Python 2.7.9 or later, or Python 3.4 or later, you already have pip, although you might want to upgrade it. For more information about upgrading or installing pip, see Installation in the pip documentation.

Installation

Use pip to install the AWS Encryption SDK for Python, as shown in the following examples.

To install the latest version

```
pip install aws-encryption-sdk
```

For more details about using pip to install and upgrade packages, see Installing Packages.

The SDK requires the cryptography library on all platforms. All versions of **pip** install and build the **cryptography** library on Windows. **pip** 8.1 and later installs and builds **cryptography** on Linux. If you are using an earlier version of **pip** and your Linux environment doesn't have the tools needed to build the **cryptography** library, you need to install them. For more information, see Building cryptography on Linux.

For the latest development version of this SDK, go to the aws-encryption-sdk-python GitHub repository.

After you install the SDK, get started by looking at the example Python code (p. 21) in this guide.

AWS Encryption SDK for Python Example Code

The following examples show you how to use the AWS Encryption SDK for Python to encrypt and decrypt data.

Topics

- Encrypting and Decrypting Strings (p. 21)
- Encrypting and Decrypting Byte Streams (p. 22)
- Encrypting and Decrypting Byte Streams with Multiple Master Key Providers (p. 24)

Encrypting and Decrypting Strings

The following example shows you how to 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 2017 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.
"""

from __future__ import print_function

import aws_encryption_sdk


def cycle_string(key_arn, source_plaintext, botocore_session=None):
    """Encrypts and then decrypts a string using a KMS customer master key (CMK)

    :param str key_arn: Amazon Resource Name (ARN) of the KMS CMK
    (http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html)
    :param bytes source_plaintext: Data to encrypt
    :param botocore_session: Existing Botocore session instance
    :type botocore_session: botocore.session.Session
    """
```

```
    # Create a KMS master key provider
    kms_kwargs = dict(key_ids=[key_arn])
    if botocore_session is not None:
        kms_kwargs['botocore_session'] = botocore_session
    master_key_provider = aws_encryption_sdk.KMSMasterKeyProvider(**kms_kwargs)

    # Encrypt the plaintext source data
    ciphertext, encryptor_header = aws_encryption_sdk.encrypt(
        source=source_plaintext,
        key_provider=master_key_provider
    )
    print('Ciphertext: ', ciphertext)

    # Decrypt the ciphertext
    cycled_plaintext, decrypted_header = aws_encryption_sdk.decrypt(
        source=ciphertext,
        key_provider=master_key_provider
    )

    # Verify that the "cycled" (encrypted, then decrypted) plaintext is identical to the
 source
    # plaintext
    assert cycled_plaintext == source_plaintext

    # Verify that the encryption context used in the decrypt operation includes all key
 pairs from
    # the encrypt operation. (The SDK can add pairs, so don't require an exact match.)
    #
    # In production, always use a meaningful encryption context. In this sample, we omit
 the
    # encryption context (no key pairs).
    assert all(
        pair in decrypted_header.encryption_context.items()
        for pair in encryptor_header.encryption_context.items()
    )

    print('Decrypted: ', cycled_plaintext)
```

Encrypting and Decrypting Byte Streams

The following example shows you how to use the AWS Encryption SDK to encrypt and decrypt byte streams. This example doesn't use AWS. It uses a static, ephemeral master key provider.

```
"""
Copyright 2017 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.
"""

import filecmp
import os

import aws_encryption_sdk
```

```
from aws_encryption_sdk.internal.crypto import WrappingKey
from aws_encryption_sdk.key_providers.raw import RawMasterKeyProvider
from aws_encryption_sdk.identifiers import WrappingAlgorithm, EncryptionKeyType


class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
    """Randomly and consistently generates 256-bit keys for each unique key ID."""
    provider_id = 'static-random'

    def __init__(self, **kwargs):
        self._static_keys = {}

    def _get_raw_key(self, key_id):
        """Returns a static, randomly-generated symmetric key for the specified key ID.

        :param str key_id: Key ID
        :returns: Wrapping key that contains the specified static key
        :rtype: :class:`aws_encryption_sdk.internal.crypto.WrappingKey`
        """
        try:
            static_key = self._static_keys[key_id]
        except KeyError:
            static_key = os.urandom(32)
            self._static_keys[key_id] = static_key
        return WrappingKey(
            wrapping_algorithm=WrappingAlgorithm.AES_256_GCM_IV12_TAG16_NO_PADDING,
            wrapping_key=static_key,
            wrapping_key_type=EncryptionKeyType.SYMMETRIC
        )


def cycle_file(source_plaintext_filename):
    """Encrypts and then decrypts a file under a custom static master key provider.

    :param str source_plaintext_filename: Filename of file to encrypt
    """

    # Create a static random master key provider
    key_id = os.urandom(8)
    master_key_provider = StaticRandomMasterKeyProvider()
    master_key_provider.add_master_key(key_id)

    ciphertext_filename = source_plaintext_filename + '.encrypted'
    cycled_plaintext_filename = source_plaintext_filename + '.decrypted'

    # Encrypt the plaintext source data
    with open(source_plaintext_filename, 'rb') as plaintext, open(ciphertext_filename,
 'wb') as ciphertext:
        with aws_encryption_sdk.stream(
            mode='e',
            source=plaintext,
            key_provider=master_key_provider
        ) as encryptor:
            for chunk in encryptor:
                ciphertext.write(chunk)

    # Decrypt the ciphertext
    with open(ciphertext_filename, 'rb') as ciphertext, open(cycled_plaintext_filename,
 'wb') as plaintext:
        with aws_encryption_sdk.stream(
            mode='d',
            source=ciphertext,
            key_provider=master_key_provider
        ) as decryptor:
            for chunk in decryptor:
                plaintext.write(chunk)
```

```
    # Verify that the "cycled" (encrypted, then decrypted) plaintext is identical to the
 source
    # plaintext
    assert filecmp.cmp(source_plaintext_filename, cycled_plaintext_filename)

    # Verify that the encryption context used in the decrypt operation includes all key
 pairs from
    # the encrypt operation
        #
    # In production, always use a meaningful encryption context. In this sample, we omit
 the
    # encryption context (no key pairs).
    assert all(
        pair in decryptor.header.encryption_context.items()
        for pair in encryptor.header.encryption_context.items()
    )
    return ciphertext_filename, cycled_plaintext_filename
```

Encrypting and Decrypting Byte Streams with Multiple Master Key Providers

The following example shows you how to use the AWS Encryption SDK with more than one master key provider. Using more than one master key provider creates redundancy if 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 2017 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.
"""

import filecmp
import os

import aws_encryption_sdk
from aws_encryption_sdk.internal.crypto import WrappingKey
from aws_encryption_sdk.key_providers.raw import RawMasterKeyProvider
from aws_encryption_sdk.identifiers import WrappingAlgorithm, EncryptionKeyType
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa


class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
    provider_id = 'static-random'

    def __init__(self, **kwargs):
        self._static_keys = {}

    def _get_raw_key(self, key_id):
        """Returns a static, randomly generated, RSA key for the specified key ID.
```

```
        :param str key_id: User-defined ID for the static key
        :returns: Wrapping key that contains the specified static key
        :rtype: :class:`aws_encryption_sdk.internal.crypto.WrappingKey`
        """
        try:
            static_key = self._static_keys[key_id]
        except KeyError:
            private_key = rsa.generate_private_key(
                public_exponent=65537,
                key_size=4096,
                backend=default_backend()
            )
            static_key = private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption()
            )
            self._static_keys[key_id] = static_key
        return WrappingKey(
            wrapping_algorithm=WrappingAlgorithm.RSA_OAEP_SHA1_MGF1,
            wrapping_key=static_key,
            wrapping_key_type=EncryptionKeyType.PRIVATE
        )


def cycle_file(key_arn, source_plaintext_filename, botocore_session=None):
    """Encrypts and then decrypts a file using a KMS master key provider and a custom
 static master
    key provider. Both master key providers are used to encrypt the plaintext file, so
 either one alone
    can decrypt it.

    :param str key_arn: Amazon Resource Name (ARN) of the KMS Customer Master Key (CMK)
 (http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html)
    :param str source_plaintext_filename: Filename of file to encrypt
    :param botocore_session: existing botocore session instance
    :type botocore_session: botocore.session.Session
    """

    # "Cycled" means encrypted and then decrypted
    ciphertext_filename = source_plaintext_filename + '.encrypted'
    cycled_kms_plaintext_filename = source_plaintext_filename + '.kms.decrypted'
    cycled_static_plaintext_filename = source_plaintext_filename + '.static.decrypted'

    # Create a KMS master key provider
    kms_kwargs = dict(key_ids=[key_arn])
    if botocore_session is not None:
        kms_kwargs['botocore_session'] = botocore_session
    kms_master_key_provider = aws_encryption_sdk.KMSMasterKeyProvider(**kms_kwargs)

    # Create a static master key provider and add a master key to it
    static_key_id = os.urandom(8)
    static_master_key_provider = StaticRandomMasterKeyProvider()
    static_master_key_provider.add_master_key(static_key_id)

    # Create a master key provider that includes the KMS and static master key providers
    kms_master_key_provider.add_master_key_provider(static_master_key_provider)

    # Encrypt plaintext with both KMS and static master keys
    with open(source_plaintext_filename, 'rb') as plaintext, open(ciphertext_filename,
 'wb') as ciphertext:
        with aws_encryption_sdk.stream(
            source=plaintext,
            mode='e',
            key_provider=kms_master_key_provider
        ) as encryptor:
```

```
            for chunk in encryptor:
                ciphertext.write(chunk)

    # Decrypt the ciphertext with only the KMS master key
    with open(ciphertext_filename, 'rb') as ciphertext, open(cycled_kms_plaintext_filename,
'wb') as plaintext:
        with aws_encryption_sdk.stream(
            source=ciphertext,
            mode='d',
            key_provider=aws_encryption_sdk.KMSMasterKeyProvider(**kms_kwargs)
        ) as kms_decryptor:
            for chunk in kms_decryptor:
                plaintext.write(chunk)

    # Decrypt the ciphertext with only the static master key
    with open(ciphertext_filename, 'rb') as ciphertext,
open(cycled_static_plaintext_filename, 'wb') as plaintext:
        with aws_encryption_sdk.stream(
            source=ciphertext,
            mode='d',
            key_provider=static_master_key_provider
        ) as static_decryptor:
            for chunk in static_decryptor:
                plaintext.write(chunk)

    # Verify that the "cycled" (encrypted, then decrypted) plaintext is identical to the
source
    # plaintext
    assert filecmp.cmp(source_plaintext_filename, cycled_kms_plaintext_filename)
    assert filecmp.cmp(source_plaintext_filename, cycled_static_plaintext_filename)

    # Verify that the encryption context in the decrypt operation includes all key pairs
from the
    # encrypt operation.
    #
    # In production, always use a meaningful encryption context. In this sample, we omit
the
    # encryption context (no key pairs).
    assert all(
        pair in kms_decryptor.header.encryption_context.items()
        for pair in encryptor.header.encryption_context.items()
    )
    assert all(
        pair in static_decryptor.header.encryption_context.items()
        for pair in encryptor.header.encryption_context.items()
    )
    return ciphertext_filename, cycled_kms_plaintext_filename,
cycled_static_plaintext_filename
```

AWS Encryption SDK Command Line Interface

The AWS Encryption SDK Command Line Interface (AWS Encryption CLI) enables you to use the AWS Encryption SDK to encrypt and decrypt data interactively at the command line and in scripts. You don't need cryptography or programming expertise.

Like all implementations of the AWS Encryption SDK, the AWS Encryption CLI offers advanced data protection features. These include envelope encryption, additional authenticated data (AAD), and secure, authenticated, symmetric key algorithm suites, such as 256-bit AES-GCM with key derivation and signing.

The AWS Encryption CLI is built on the AWS Encryption SDK for Python and is supported on Linux, macOS, and Windows. You can run commands and scripts to encrypt and decrypt your data in your preferred shell on Linux or macOS, in a Command Prompt window (cmd.exe) on Windows, and in a PowerShell console on any system.

All language-specific implementations of the AWS Encryption SDK, including the AWS Encryption CLI, are interoperable. For example, you can encrypt data with the AWS Encryption SDK for Java (p. 12) and decrypt it with the AWS Encryption CLI.

This topic introduces the AWS Encryption CLI, explains how to install and use it, and provides several examples to help you get started. For a quick start, see How to Encrypt and Decrypt Your Data with the AWS Encryption CLI in the AWS Security Blog. For more detailed information, see Read The Docs, and join us in developing the AWS Encryption CLI in the aws-encryption-sdk-cli repository on GitHub.

Topics

- Installing the AWS Encryption SDK Command Line Interface (p. 27)
- How to Use the AWS Encryption SDK Command Line Interface (p. 29)
- Examples of the AWS Encryption SDK Command Line Interface (p. 36)
- AWS Encryption SDK CLI Syntax and Parameter Reference (p. 49)

Installing the AWS Encryption SDK Command Line Interface

This topic explains how to install the AWS Encryption CLI. For detailed information, see the aws-encryption-sdk-cli repository on GitHub and Read the Docs.

Topics

- Installing the Prerequisites (p. 27)
- Installing the AWS Encryption CLI (p. 28)

Installing the Prerequisites

The AWS Encryption CLI is built on the AWS Encryption SDK for Python. To use the AWS Encryption CLI, you need Python and **pip**, the Python package management tool. Python and **pip** are available on all supported platforms.

Before you install the AWS Encryption CLI, be sure that you have the following prerequisites.

Python

The AWS Encryption CLI requires Python 2.7, or Python 3.4 or later. Python is included in most Linux and macOS installations, although you might need to upgrade to one of the required versions. However, you have to install Python on Windows, if it is not already installed. To download Python, see Python downloads.

To determine whether Python is installed, at the command line, type:

```
python
```

To check the Python version, use the `-V` (uppercase V) parameter.

```
python -V
```

On Windows, you need to install Python. Then, add the path to the `Python.exe` file to the value of the **Path** environment variable.

By default, Python is installed in the all users directory or in a user profile directory (`$home` or `%userprofile%`) in the `AppData\Local\Programs\Python` subdirectory. To find the location of the `Python.exe` file on your system, check one of the following registry keys. You can use PowerShell to search the registry.

```
PS C:\> dir HKLM:\Software\Python\PythonCore\version\InstallPath
# -or-
PS C:\> dir HKCU:\Software\Python\PythonCore\version\InstallPath
```

pip

pip is the Python package manager. To install the AWS Encryption CLI and its dependencies, you need **pip** 8.1 or later.

For help installing or upgrading **pip**, see Installation in the **pip** documentation.

AWS Command Line Interface

The AWS Command Line Interface (AWS CLI) is required only if you are using AWS Key Management Service (AWS KMS) customer master keys (CMKs) with the AWS Encryption CLI. If you are using a different master key provider (p. 7), the AWS CLI is not required.

To use AWS KMS CMKs with the AWS Encryption CLI, you need to install and configure the AWS CLI. The configuration makes the credentials that you use to authenticate to AWS KMS available to the AWS Encryption CLI.

Installing the AWS Encryption CLI

Use **pip** to install the AWS Encryption CLI and the Python cryptography library that it requires.

The AWS Encryption CLI requires the **cryptography** library on all platforms. All versions of **pip** install and build the **cryptography** library on Windows and OS X.

On Linux, **pip** 8.1 and later installs and builds the **cryptography** library. If you are using an earlier version of **pip** and your Linux environment doesn't have the tools needed to build the **cryptography** library, you must install them. For more information, see Building cryptography on Linux.

To install the latest version

```
pip install aws-encryption-sdk-cli
```

To upgrade to the latest version

```
pip install --upgrade aws-encryption-sdk-cli
```

To find the version number of your AWS Encryption CLI and AWS Encryption SDK

```
aws-encryption-cli --version

aws-encryption-sdk-cli/1.1.0 aws-encryption-sdk/1.3.2
```

To install the version of the AWS Encryption CLI currently in development, see the aws-encryption-sdk-cli repository on GitHub.

For more details about using **pip** to install and upgrade Python packages, see the pip documentation.

How to Use the AWS Encryption SDK Command Line Interface

This topic explains how to use the parameters in the AWS Encryption CLI. For examples, see Examples of the AWS Encryption SDK Command Line Interface (p. 36). For complete documentation, see Read the Docs.

Topics

- How to Encrypt and Decrypt Data (p. 29)
- How to Specify a Master Key (p. 30)
- How to Provide Input (p. 32)
- How to Specify the Output Location (p. 32)
- How to Use an Encryption Context (p. 33)
- How to Store Parameters in a Configuration File (p. 34)

How to Encrypt and Decrypt Data

The AWS Encryption CLI uses the features of the AWS Encryption SDK to make it easy to encrypt and decrypt data securely.

- When you encrypt data in the AWS Encryption CLI, you specify your plaintext data and a master key (p. 6), such as an AWS Key Management Service (AWS KMS) customer master key (CMK). If you are using a custom master key provider, you need to specify the provider. You also specify output locations for the encrypted message (p. 8) and for metadata about the encryption operation. An encryption context (p. 8) is optional, but recommended.

  ```
  aws-encryption-cli --encrypt --input myPlaintextData \
                     --master-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
                     --output myEncryptedMessage \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test
  ```

  The AWS Encryption CLI gets a unique data key from the master key and encrypts your data. It returns an encrypted message (p. 8) and metadata about the operation. The encrypted message contains your encrypted data (*ciphertext*) and an encrypted copy of the data key. You don't have to worry about storing, managing, or losing the data key.

- When you decrypt data, you pass in your encrypted message, the optional encryption context, and location for the plaintext output and the metadata. If you are using a custom master key provider, you also supply the master key. If you are using an AWS KMS CMK, AWS KMS derives the master key from the encrypted message.

  ```
  aws-encryption-cli --decrypt --input myEncryptedMessage \
                     --output myPlaintextData \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test
  ```

  The AWS Encryption CLI uses the master key to decrypt the data key in the encrypted message. Then it uses the data key to decrypt your data. It returns your plaintext data and metadata about the operation.

How to Specify a Master Key

When you encrypt data in the AWS Encryption CLI, you need to specify a master key (p. 6). You can use an AWS KMS customer master key (CMK) or a master key from a custom master key provider (p. 7). The custom master key provider can be any compatible Python master key provider.

To specify a master key, use the `--master-keys` parameter (`-m`). Its value is a collection of attributes (p. 30) with the `attribute=value` format. The attributes that you use depend on the master key provider and the command.

- **AWS KMS**. In encrypt commands, you must specify a `--master-keys` parameter with a **key** attribute. The other attributes are optional. In decrypt commands, the `--master-keys` parameter is optional and it can only have a **profile** attribute.
- **Custom master key provider**. You must specify the `--master-keys` parameter in every command. The parameter value must have **key** and **provider** attributes.

You can include multiple `--master-keys` parameters (p. 31) in the same command.

Master Key Parameter Attributes

The value of the `--master-keys` parameter consists of the following attributes and their values.

If an attribute name or value includes spaces or special characters, enclose both the name and value in quotation marks. For example, `--master-keys key=12345 "provider=my cool provider"`.

Key: Specify a Master Key

Use the **key** attribute to identify a master key. The value can be any key identifier that the master key provider recognizes.

```
--master-keys key=1234abcd-12ab-34cd-56ef-1234567890ab
```

In an encrypt command, each `--master-keys` parameter value must include at least one **key** attribute and value. You can use multiple **key** attributes (p. 31) in each `--master-keys` parameter value.

```
aws-encryption-cli --encrypt --master-keys key=1234abcd-12ab-34cd-56ef-1234567890ab
 key=1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d
```

In encrypt commands that use AWS KMS CMKs, the value of **key** can be the CMK ID, its Amazon Resource Name (ARN), an alias name, or alias ARN. For example, this encrypt command uses an alias ARN in the value of the **key** attribute.

```
aws-encryption-cli --encrypt --master-keys key=arn:aws:kms:us-
west-2:111122223333:alias/ExampleAlias
```

In decrypt commands that use a custom master key provider, **key** and **provider** attributes are required. The key attribute is not permitted in decrypt commands that use an AWS KMS CMK.

```
aws-encryption-cli --decrypt --master-keys provider='myProvider' key='100101'
```

Provider: Specify the Master Key Provider

The **provider** attribute identifies the master key provider (p. 7). The default value is `aws-kms`, which represents AWS KMS. If you are using a different master key provider, the **provider** attribute is required.

```
--master-keys key=12345 provider=my_custom_provider
```

For more information about using custom (non-AWS KMS) master key providers, see the **Advanced Configuration** topic in the README file for the AWS Encryption SDK CLI repository.

Region: Specify an AWS Region

Use the **region** attribute to specify the AWS Region of an AWS KMS CMK. This attribute is valid only in encrypt commands and only when the master key provider is AWS KMS.

```
--encrypt --master-keys key=alias/primary-key region=us-east-2
```

AWS Encryption CLI commands use the AWS Region that is specified in the **key** attribute value if it includes a region, such as an ARN. if the **key** value specifies a AWS Region, the **region** attribute is ignored.

The **region** attribute takes precedence over other region specifications. If you don't use a region attribute, AWS Encryption CLI commands uses the AWS Region specified in your AWS CLI named profile, if any, or your default profile.

Profile: Specify a Named Profile

Use the **profile** attribute to specify an AWS CLI named profile. Named profiles can include credentials and an AWS Region. This attribute is valid only when the master key provider is AWS KMS.

```
--master-keys key=alias/primary-key profile=admin-1
```

You can use the **profile** attribute to specify alternate credentials in encrypt and decrypt commands. In an encrypt command, the AWS Encryption CLI uses the AWS Region in the named profile only when the **key** value does not include a region and there is no **region** attribute. In a decrypt command, the AWS Region in the name profile is ignored.

How to Specify Multiple Master Keys

You can specify multiple master keys in each command.

If you specify more than one master key, the first master key generates (and encrypts) the data key that is used to encrypt your data. The other master keys only encrypt the data key. The resulting encrypted message (p. 8) contains the encrypted data ("ciphertext") and a collection of encrypted data keys, one encrypted by each master key. Any of the master keys can decrypt one data key and then decrypt the data.

There are two ways to specify multiple master keys:

- Include multiple **key** attributes in a `--master-keys` parameter value.

```
$cmk_oregon=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$cmk_ohio=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef

--master-keys key=$cmk_oregon key=$cmk_ohio
```

- Include multiple `--master-keys` parameters in the same command. Use this syntax when the attribute values that you specify do not apply to all of the master keys in the command.

```
--master-keys region=us-east-2 key=alias/primary_CMK \
```

```
--master-keys region=us-west-1 key=alias/primary_CMK
```

How to Provide Input

The encrypt operation in the AWS Encryption CLI takes plaintext data as input and returns an encrypted message (p. 8). The decrypt operation takes an encrypted message as input and returns plaintext data.

The `--input` parameter (`-i`) , which tells the AWS Encryption CLI where to find the input, is required in all AWS Encryption CLI commands.

You can provide input in any of the following ways:

- Use a file.

```
--input myData.txt
```

- Use a file name pattern.

```
--input testdir/*.xml
```

- Use a directory or directory name pattern. When the input is a directory, the `--recursive` parameter (`-r`, `-R`) is required.

```
--input testdir --recursive
```

- Pipe input to the command (stdin). Use a value of `-` for the `--input` parameter. (The `--input` parameter is always required.)

```
echo 'Hello World' | aws-encryption-cli --encrypt --input -
```

How to Specify the Output Location

The `--output` parameter tells the AWS Encryption CLI where to write the results of the encryption or decryption operation. It is required in every AWS Encryption CLI command. The AWS Encryption CLI creates a new output file for every input file in the operation.

If an output file already exists, by default, the AWS Encryption CLI prints a warning, then overwrites the file. To prevent overwriting, use the `--interactive` parameter, which prompts you for confirmation before overwriting, or `--no-overwrite`, which skips the input if the output would cause an overwrite. To suppress the overwrite warning, use `--quiet`. To capture errors and warnings from the AWS Encryption CLI, use the `2>&1` redirection operator to write them to the output stream.

> **Note**
> Commands that overwrite output files begin by deleting the output file. If the command fails, the output file might already be deleted.

You can the output location in several ways.

- Specify a file name. If you specify a path to the file, all directories in the path must exist before the command runs.

```
--output myEncryptedData.txt
```

- Specify a directory. The output directory must exist before the command runs.

If the input contains subdirectories, the command reproduces the subdirectories under the specified directory.

```
--output Test
```

When the output location is a directory (without file names), the AWS Encryption CLI creates output file names based on the input file names plus a suffix. Encrypt operations append `.encrypted` to the input file name and the decrypt operations append `.decrypted`. To change the suffix, use the `--suffix` parameter.

For example, if you encrypt `file.txt`, the encrypt command creates `file.txt.encrypted`. If you decrypt `file.txt.encrypted`, the decrypt command creates `file.txt.encrypted.decrypted`.

- Write to the command line (stdout). Enter a value of - for the `--output` parameter. You can use `--output -` to pipe output to another command or program.

```
--output -
```

How to Use an Encryption Context

The AWS Encryption CLI lets you provide an encryption context in encrypt and decrypt commands. It is not required, but it is a cryptographic best practice that we recommend.

An *encryption context* is a type of arbitrary, non-secret *additional authenticated data*. In the AWS Encryption CLI, the encryption context consists of a collection of `name=value` pairs. You can use any content in the pairs, including information about the files, data that helps you to find the encryption operation in logs, or data that your grants or policies require.

In an Encrypt Command

The encryption context that you specify in an encrypt command, along with any additional encryption context that the encryption components add, is cryptographically bound to the encrypted data. It is also included (in plaintext) in the encrypted message (p. 8) that the command returns. If you are using an AWS KMS customer master key (CMK), the encryption context also might appear in plaintext in audit records and logs, such as AWS CloudTrail.

The following example shows a encryption context with three `name=value` pairs.

```
--encryption-context purpose=test dept=IT class=confidential
```

In a Decrypt Command

In a decrypt command, the encryption context helps you to confirm that you are decrypting the right encrypted message.

You are not required to provide an encryption context in a decrypt command, even if an encryption context was used on encrypt. However, if you do, the AWS Encryption CLI verifies that every element in the encryption context of the decrypt command matches an element in the encryption context of the encrypted message. If any element does not match, the decrypt command fails.

For example, the following command decrypts the encrypted message only if its encryption context includes `dept=IT`.

```
aws-encryption-cli --decrypt --encryption-context dept=IT ...
```

An encryption context is an important part of your security strategy. However, when choosing an encryption context, remember that its values are not secret. Do not include any confidential data in the encryption context.

To specify an encryption context:

- In an **encrypt** command, use the `--encryption-context` parameter with one or more `name=value` pairs. Use a space to separate each pair.

```
--encryption-context name=value [name=value] ...
```

- In a **decrypt** command, the `--encryption-context` parameter value can include `name=value` pairs, `name` elements (with no values), or a combination of both.

```
--encryption-context name[=value] [name] [name=value] ...
```

If the `name` or `value` in a `name=value` pair includes spaces or special characters, enclose the entire pair in quotation marks.

```
--encryption-context "department=software engineering" "AWS Region=us-west-2"
```

For example, this encrypt command includes an encryption context with two pairs, `purpose=test` and `dept=23`.

```
aws-encryption-cli --encrypt --encryption-context purpose=test dept=23 ...
```

These decrypt command would succeed. The encryption context in each commands is a subset of the original encryption context.

```
\\ Any one or both of the encryption context pairs
aws-encryption-cli --decrypt --encryption-context dept=23 ...

\\ Any one or both of the encryption context names
aws-encryption-cli --decrypt --encryption-context purpose ...

\\ Any combination of names and pairs
aws-encryption-cli --decrypt --encryption-context dept purpose=test ...
```

However, these decrypt commands would fail. The encryption context in the encrypted message does not contain the specified elements.

```
aws-encryption-cli --decrypt --encryption-context dept=Finance ...
aws-encryption-cli --decrypt --encryption-context scope ...
```

How to Store Parameters in a Configuration File

You can save time and avoid typing errors by saving frequently used AWS Encryption CLI parameters and values in configuration files.

A *configuration file* is a text file that contains parameters and values for an AWS Encryption CLI command. When you refer to a configuration file in a AWS Encryption CLI command, the reference is replaced by the parameters and values in the configuration file. The effect is the same is if you typed the file content at the command line. A configuration file can have any name and it can be located in any directory that the current user can access.

The following example configuration file, `cmk.conf`, specifies two AWS KMS CMKs in different regions.

```
--master-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
--master-keys key=arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
```

To use the configuration file in a command, prefix the file name with an at sign (`@`). In a PowerShell console, use a backtick character to escape the at sign (`` `@ ``).

This example command uses the `cmk.conf` file in an encrypt command.

Bash

```
$ aws-encryption-cli -e @cmk.conf -i hello.txt -o testdir
```

PowerShell

```
PS C:\> aws-encryption-cli -e `@cmk.conf -i .\Hello.txt -o .\TestDir
```

Configuration File Rules

The rules for using configuration files are as follows:

- You can include multiple parameters in each configuration file and list them in any order. List each parameter with its values (if any) on a separate line.
- Use # to add a comment to all or part of a line.
- You can include references to other configuration files. Do not use a backtick to escape the `@` sign, even in PowerShell.
- If you use quotes in a configuration file, the quoted text cannot span multiple lines.

For example, this is the contents of an example `encrypt.conf` file.

```
# Archive Files
--encrypt
--output /archive/logs
--recursive
--interactive
--encryption-context class=unclassified dept=IT
--suffix  # No suffix
--metadata-output ~/metadata
@caching.conf  # Use limited caching
```

You can also include multiple configuration files in a command. This example command uses both the `encrypt.conf` and `master-keys.conf` configurations files.

Bash

```
$  aws-encryption-cli -i /usr/logs @encrypt.conf @master-keys.conf
```

PowerShell

```
PS C:\> aws-encryption-cli -i $home\Test\*.log `@encrypt.conf `@master-keys.conf
```

Next: Try the AWS Encryption CLI examples (p. 36)

Examples of the AWS Encryption SDK Command Line Interface

Use the following examples to try the AWS Encryption CLI on the platform you prefer. For help with master keys and other parameters, see How to Use the AWS Encryption SDK Command Line Interface (p. 29). For a quick reference, see AWS Encryption SDK CLI Syntax and Parameter Reference (p. 49).

Topics

- Encrypting a File (p. 36)
- Decrypting a File (p. 38)
- Encrypting All Files in a Directory (p. 39)
- Decrypting All Files in a Directory (p. 40)
- Encrypting and Decrypting on the Command Line (p. 42)
- Using Multiple Master Keys (p. 43)
- Encrypting and Decrypting in Scripts (p. 45)
- Using Data Key Caching (p. 47)

Encrypting a File

This example uses the AWS Encryption CLI to encrypt the contents of the `hello.txt` file, which contains a "Hello World" string.

When you run an encrypt command on a file, the AWS Encryption CLI gets the contents of the file, generates a unique data key (p. 6), encrypts the file contents under the data key, and then writes the encrypted message (p. 8) to a new file.

The first command saves the Amazon Resource Name (ARN) of an AWS KMS customer master key (CMK) in the `$cmkArn` variable.

The second command encrypts the file contents. The command uses the `--encrypt` parameter to specify the operation and the `--input` parameter to indicate the file to encrypt. The `--master-keys` parameter (p. 30), and its required **key** attribute, tell the command to use the master key represented by the CMK ARN.

The command uses the `--metadata-output` parameter to specify a text file for the metadata about the encryption operation. As a best practice, the command uses the `--encryption-context` parameter to specify an encryption context (p. 33).

The value of the `--output` parameter, a dot (.), tells the command to write the output file to the current directory.

Bash

```
\\ To run this example, replace the fictitious CMK ARN with a valid value.
```

```
$ cmkArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --master-keys key=$cmkArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .
```

PowerShell

```
# To run this example, replace the fictitious CMK ARN with a valid value.
PS C:\> $CmkArn = arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

PS C:\> aws-encryption-cli --encrypt `
                           --input Hello.txt `
                           --master-keys key=$CmkArn `
                           --metadata-output $home\Metadata.txt `
                           --encryption-context purpose=test `
                           --output .
```

When the encrypt command succeeds, it does not return any output. To determine whether the command succeeded, check the Boolean value in the `$?` variable. When the command succeeds, the value of `$?` is `0` (Bash) or `True` (PowerShell). When the command fails, the value of `$?` is non-zero (Bash) or `False` (PowerShell).

Bash

```
$ echo $?
0
```

PowerShell

```
PS C:\> $?
True
```

You can also use a directory listing command to see that the encrypt command created a new file, `hello.txt.encrypted`. Because the encrypt command did not specify a file name for the output, the AWS Encryption CLI wrote the output to a file with the same name as the input file plus a `.encrypted` suffix. To use a different suffix, or suppress the suffix, use the `--suffix` parameter.

The `hello.txt.encrypted` file contains an encrypted message (p. 8) that includes the ciphertext of the `hello.txt` file, an encrypted copy of the data key, and additional metadata, including the encryption context.

Bash

```
$  ls
hello.txt   hello.txt.encrypted
```

PowerShell

```
PS C:\> dir
```

```
    Directory: C:\TestCLI

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
```

Decrypting a File

This example uses the AWS Encryption CLI to decrypt the contents of the `Hello.txt.encrypted` file that was encrypted in the previous example.

The decrypt command uses the `--decrypt` parameter to indicate the operation and `--input` parameter to identify the file to decrypt. The value of the `--output` parameter is a dot that represents the current directory.

This command does not have a `--master-keys` parameter. A `--master-keys` parameter is required in decrypt commands only when you are using a custom master key provider. If you are using an AWS KMS CMK, you cannot specify a master key, because AWS KMS derives it from the encrypted message.

The `--encryption-context` parameter is optional in the decrypt command, even when an encryption context (p. 33) is provided in the encrypt command. In this case, the decrypt command uses the same encryption context that was provided in the encrypt command. Before decrypting, the AWS Encryption CLI verifies that the encryption context in the encrypted message includes a `purpose=test` pair. If it does not, the decrypt command fails.

The `--metadata-output` parameter specifies a file for metadata about the decryption operation. The value of the `--output` parameter, a dot (.), writes the output file to the current directory.

Bash

```
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

PowerShell

```
PS C:\> aws-encryption-cli --decrypt `
                           --input Hello.txt.encrypted `
                           --encryption-context purpose=test `
                           --metadata-output $home\Metadata.txt `
                           --output .
```

When a decrypt command succeeds, it does not return any output. To determine whether the command succeeded, get the value of the `$?` variable. You can also use a directory listing command to see that the command created a new file with a `.decrypted` suffix. To see the plaintext content, use a command to get the file content, such as `cat` or Get-Content.

Bash

```
$  ls
hello.txt  hello.txt.encrypted  hello.txt.encrypted.decrypted

$  cat hello.txt.encrypted.decrypted
```

```
Hello World
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/17/2017   1:01 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
-a----        9/17/2017   1:08 PM             11 Hello.txt.encrypted.decrypted


PS C:\> Get-Content Hello.txt.encrypted.decrypted
Hello World
```

Encrypting All Files in a Directory

This example uses the AWS Encryption CLI to encrypt the contents of all of the files in a directory.

When a command affects multiple files, the AWS Encryption CLI processes each file individually. It gets the file contents, gets a unique data key (p. 6) for the file from the master key, encrypts the file contents under the data key, and writes the results to a new file in the output directory. As a result, you can decrypt the output files independently.

This listing of the `TestDir` directory shows the plaintext files that we want to encrypt.

Bash

```
$  ls testdir
cool-new-thing.py  hello.txt  employees.csv
```

PowerShell

```
PS C:\> dir C:\TestDir

    Directory: C:\TestDir

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/12/2017   3:11 PM           2139 cool-new-thing.py
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:44 PM             46 Employees.csv
```

The first command saves the Amazon Resource Name (ARN) of an AWS KMS customer master key (CMK) in the `$cmkArn` variable.

The second command encrypts the content of files in the `TestDir` directory and writes the files of encrypted content to the `TestEnc` directory. If the `TestEnc` directory doesn't exist, the command fails. Because the input location is a directory, the `--recursive` parameter is required.

The `--master-keys` parameter (p. 30), and its required **key** attribute, specify the master key. The encrypt command includes an encryption context (p. 33), `dept=IT`. When you specify an encryption context in a command that encrypts multiple files, the same encryption context is used for all of the files.

The command also has a `--metadata-output` parameter to tell the AWS Encryption CLI where to write the metadata about the encryption operations. The AWS Encryption CLI writes one metadata record for each file that was encrypted.

When the command completes, the AWS Encryption CLI writes the encrypted files to the `TestEnc` directory, but it does not return any output.

The final command lists the files in the `TestEnc` directory. There is one output file of encrypted content for each input file of plaintext content. Because the command did not specify an alternate suffix, the encrypt command appended `.encrypted` to each of the input file names.

Bash

```
# To run this example, replace the fictitious CMK ARN with a valid master key
 identifier.
$  cmkArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --encrypt \
                     --input testdir --recursive\
                     --master-keys key=$cmkArn \
                     --encryption-context dept=IT \
                     --metadata-output ~/metadata \
                     --output testenc

$ ls testenc
cool-new-thing.py.encrypted  employees.csv.encrypted  hello.txt.encrypted
```

PowerShell

```
# To run this example, replace the fictitious CMK ARN with a valid master key
 identifier.
PS C:\> $cmkArn = arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

PS C:\> aws-encryption-cli --encrypt `
                           --input .\TestDir --recursive `
                           --master-keys key=$cmkArn `
                           --encryption-context dept=IT `
                           --metadata-output .\Metadata\Metadata.txt `
                           --output .\TestEnc

PS C:\> dir .\TestEnc

    Directory: C:\TestEnc

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

Decrypting All Files in a Directory

This example decrypts all files in a directory. It starts with the files in the `TestEnc` directory that were encrypted in the previous example.

Bash

```
$  ls testenc
```

```
cool-new-thing.py.encrypted  hello.txt.encrypted  employees.csv.encrypted
```

PowerShell

```
PS C:\> dir C:\TestEnc

    Directory: C:\TestEnc

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

This decrypt command decrypts all of the files in the TestEnc directory and writes the plaintext files to the TestDec directory. Because the encrypted files were encrypted under an AWS KMS CMK, there is no `--master-keys` parameter in the command. The command uses the `--interactive` parameter to tell the AWS Encryption CLI to prompt you before overwriting a file with the same name.

This command also uses the encryption context that was provided when the files were encrypted. When decrypting multiple files, the AWS Encryption CLI checks the encryption context of every file. If the encryption context check on any file fails, the AWS Encryption CLI rejects the file, writes a warning, records the failure in the metadata, and then continues checking the remaining files. If the AWS Encryption CLI fails to decrypt a file for any other reason, the entire decrypt command fails immediately.

In this example, the encrypted messages in all of the input files contain the `dept=IT` encryption context element. However, if you were decrypting messages with different encryption contexts, you might still be able to verify part of the encryption context. For example, if some messages had an encryption context of `dept=finance` and others had `dept=IT`, you could verify that the encryption context always contains a `dept` name without specifying the value. If you wanted to be more specific, you could decrypt the files in separate commands.

The decrypt command does not return any output, but you can use a directory listing command to see that it created new files with the `.decrypted` suffix. To see the plaintext content, use a command to get the file content.

Bash

```
$ aws-encryption-cli --decrypt --input testenc --recursive \
                     --encryption-context dept=IT \
                     --metadata-output ~/metadata \
                     --output testdec --interactive

$ ls testdec
cool-new-thing.py.encrypted.decrypted  hello.txt.encrypted.decrypted
 employees.csv.encrypted.decrypted
```

PowerShell

```
PS C:\> aws-encryption-cli --decrypt `
                           --input C:\TestEnc --recursive `
                           --encryption-context dept=IT `
                           --metadata-output $home\Metadata.txt `
                           --output C:\TestDec --interactive

PS C:\> dir .\TestDec
```

```
    Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        10/8/2017   4:57 PM           2139 cool-new-thing.py.encrypted.decrypted
-a----        10/8/2017   4:57 PM             46 Employees.csv.encrypted.decrypted
-a----        10/8/2017   4:57 PM             11 Hello.txt.encrypted.decrypted
```

Encrypting and Decrypting on the Command Line

These examples show you how to pipe input to commands (stdin) and write output to the command line (stdout). They explain how to represent stdin and stdout in a command and how to use the built-in Base64 encoding tools to prevent the shell from misinterpreting non-ASCII characters.

This example pipes a plaintext string to an encrypt command and saves the encrypted message in a variable. Then, it pipes the encrypted message in the variable to a decrypt command, which writes its output to the pipeline (stdout).

The example consists of three commands:

- The first command saves the Amazon Resource Name (ARN) of an AWS KMS customer master key (CMK) in the `$cmkArn` variable.

  Bash

  ```
  $  cmkArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
  ```

  PowerShell

  ```
  PS C:\> $cmkArn = arn:aws:kms:us-
  west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
  ```

- The second command pipes the `Hello World` string to the encrypt command and saves the result in the `$encrypted` variable.

  The `--input` and `--output` parameters are required in all AWS Encryption CLI commands. To indicate that input is being piped to the command (stdin), use a hyphen (-) for the value of the `--input` parameter. To send the output to the command line (stdout), use a hyphen for the value of the `--output` parameter.

  The `--encode` parameter Base64-encodes the output before returning it. This prevents the shell from misinterpreting the non-ASCII characters in the encrypted message.

  Because this command is just a proof of concept, we omit the encryption context and suppress the metadata (`-S`).

  Bash

  ```
  $ encrypted=$(echo 'Hello World' | aws-encryption-cli --encrypt -S \
                                                        --input - --output - --encode \
                                                        --master-keys key=$cmkArn )
  ```

  PowerShell

  ```
  PS C:\> $encrypted = 'Hello World' | aws-encryption-cli --encrypt -S `
                                                          --input - --output - --encode
   `
                                                          --master-keys key=$cmkArn
  ```

- The third command pipes the encrypted message in the `$encrypted` variable to the decrypt command.

  This decrypt command uses `--input -` to indicate that input is coming from the pipeline (stdin) and `--output -` to send the output to the pipeline (stdout). (The input parameter takes the location of the input, not the actual input bytes, so you cannot use the `$encrypted` variable as the value of the `--input` parameter.)

  Because the output was encrypted and then encoded, the decrypt command uses the `--decode` parameter to decode Base64-encoded input before decrypting it. You can also use the `--decode` parameter to decode Base64-encoded input before encrypting it.

  Again, the command omits the encryption context and suppresses the metadata (-`S`).

  Bash

  ```
  $  echo $encrypted | aws-encryption-cli --decrypt --input - --output - --decode -S
  Hello World
  ```

  PowerShell

  ```
  PS C:\> $encrypted | aws-encryption-cli --decrypt --input - --output - --decode -S
  Hello World
  ```

You can also perform the encrypt and decrypt operations in a single command without the intervening variable.

As in the previous example, the `--input` and `--output` parameters have a `-` value and the command uses the `--encode` parameter to encode the output and the `--decode` parameter to decode the input.

Bash

```
$  cmkArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$  echo 'Hello World' |
          aws-encryption-cli --encrypt --master-keys key=$cmkArn --input - --output -
 --encode -S |
          aws-encryption-cli --decrypt  --input - --output - --decode -S
Hello World
```

PowerShell

```
PS C:\> $cmkArn = arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

PS C:\> 'Hello World' |
               aws-encryption-cli --encrypt --master-keys key=$cmkArn --input - --
output - --encode -S |
               aws-encryption-cli --decrypt --input - --output - --decode -S
Hello World
```

Using Multiple Master Keys

This example shows how to use multiple master keys when encrypting and decrypting data in the AWS Encryption CLI.

When you use multiple master keys to encrypt data, any one of the master keys can be used to decrypt the data. This strategy assures that you can decrypt the data even if one of the master keys is unavailable. If you are storing the encrypted data in multiple AWS Regions, this strategy lets you use a master key in the same Region to decrypt the data.

When you encrypt with multiple master keys, the first master key plays a special role. It generates the data key that is used to encrypt the data. The remaining master keys encrypt the plaintext data key. The resulting encrypted message (p. 8) includes the encrypted data and a collection of encrypted data keys, one for each master key. Although the first master key generated the data key, any of the master keys can decrypt one of the data keys, which can be used to decrypt the data.

Encrypting with Three Master Keys

This example command uses three master keys to encrypt the `Finance.log` file, one in each of three AWS Regions.

It writes the encrypted message to the `Archive` directory. The command uses the `--suffix` parameter with no value to suppress the suffix, so the input and output files names will be the same.

The command uses the `--master-keys` parameter with three **key** attributes. You can also use multiple `--master-keys` parameters in the same command.

To encrypt the log file, the AWS Encryption CLI asks the first master key in the list, `$cmk1`, to generate the data key that it uses to encrypt the data. Then, it uses each of the other master keys to encrypt the plaintext copy of the data key. The encrypted message in the output file includes all three of the encrypted data keys.

Bash

```
$ cmk1=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$ cmk2=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
$ cmk3=arn:aws:kms:ap-southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d

$ aws-encryption-cli --encrypt --input /logs/finance.log \
                               --output /archive --suffix \
                               --encryption-context class=log \
                               --metadata-output ~/metadata \
                               --master-keys key=$cmk1 key=$cmk2 key=$cmk3
```

PowerShell

```
PS C:\> $cmk1 = arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
PS C:\> $cmk2 = arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
PS C:\> $cmk3 = arn:aws:kms:ap-
southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d

PS C:\> aws-encryption-cli --encrypt --input D:\Logs\Finance.log `
                           --output D:\Archive --suffix `
                           --encryption-context class=log `
                           --metadata-output $home\Metadata.txt `
                           --master-keys key=$cmk1 key=$cmk2 key=$cmk3
```

This command decrypts the encrypted copy of the `Finance.log` file and writes it to a `Finance.log.clear` file in the `Finance` directory.

When you decrypt data that was encrypted under AWS KMS CMKs, you cannot tell AWS KMS to use a particular CMK to decrypt the data. The **key** attribute of the `--master-keys` parameter is not valid in a decrypt command with the `aws-kms` provider. The AWS Encryption CLI can use any of the CMKs that were used to encrypt the data, provided that the AWS credentials you are using have permission to call the Decrypt API on the master key. For more information, see Authentication and Access Control for AWS KMS.

Bash

```
$ aws-encryption-cli --decrypt --input /archive/finance.log \
                     --output /finance --suffix '.clear' \
                     --metadata-output ~/metadata \
                     --encryption-context class=log
```

PowerShell

```
PS C:\> aws-encryption-cli --decrypt `
                           --input D:\Archive\Finance.log `
                           --output D:\Finance --suffix '.clear' `
                           --metadata-output .\Metadata\Metadata.txt `
                           --encryption-context class=log
```

Encrypting and Decrypting in Scripts

This example shows how to use the AWS Encryption CLI in scripts. You can write scripts that just encrypt and decrypt data, or scripts that encrypt or decrypt as part of a data management process.

In this example, the script gets a collection of log files, compresses them, encrypts them, and then copies the encrypted files to an Amazon S3 bucket. This script processes each file separately, so that you can decrypt and expand them independently.

When you compress and encrypt files, be sure to compress before you encrypt. Properly encrypted data is not compressible.

> **Warning**
> Be careful when compressing data that includes both secrets and data that might be controlled by a malicious actor. The final size of the compressed data might inadvertently reveal sensitive information about its contents.

You can find the complete scripts in the Examples directory of the aws-encryption-sdk-cli repository in GitHub.

PowerShell

```
#Requires -Modules AWSPowerShell, Microsoft.PowerShell.Archive
Param
(
    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String[]]
    $FilePath,

    [Parameter()]
    [Switch]
    $Recurse,

    [Parameter(Mandatory=$true)]
    [String]
    $masterKeyID,

    [Parameter()]
    [String]
    $masterKeyProvider = 'aws-kms',
```

```
    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $ZipDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $EncryptDirectory,

    [Parameter()]
    [String]
    $EncryptionContext,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $MetadataDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-S3Bucket -BucketName $_})]
    [String]
    $S3Bucket,

    [Parameter()]
    [String]
    $S3BucketFolder
)

BEGIN {}
PROCESS {
    if ($files = dir $FilePath -Recurse:$Recurse)
    {

        # Step 1: Compress
        foreach ($file in $files)
        {
            $fileName = $file.Name
            try
            {
                Microsoft.PowerShell.Archive\Compress-Archive -Path $file.FullName -
DestinationPath $ZipDirectory\$filename.zip
            }
            catch
            {
                Write-Error "Zip failed on $file.FullName"
            }

            # Step 2: Encrypt
            if (-not (Test-Path "$ZipDirectory\$filename.zip"))
            {
                Write-Error "Cannot find zipped file: $ZipDirectory\$filename.zip"
            }
            else
            {
                # 2>&1 captures command output
                $err = (aws-encryption-cli -e -i "$ZipDirectory\$filename.zip" `
                                           -o $EncryptDirectory `
                                           -m key=$masterKeyID provider=
$masterKeyProvider `
                                           -c $EncryptionContext `
                                           --metadata-output $MetadataDirectory `
                                           -v) 2>&1

                # Check error status
```

```
                if ($? -eq $false)
                {
                    # Write the error
                    $err
                }
                elseif (Test-Path "$EncryptDirectory\$fileName.zip.encrypted")
                {
                    # Step 3: Write to S3 bucket
                    if ($S3BucketFolder)
                    {
                        Write-S3Object -BucketName $S3Bucket -File "$EncryptDirectory\
$fileName.zip.encrypted" -Key "$S3BucketFolder/$fileName.zip.encrypted"


                    }
                    else
                    {
                        Write-S3Object -BucketName $S3Bucket -File "$EncryptDirectory\
$fileName.zip.encrypted"
                    }
                }
            }
        }
    }
}
```

Using Data Key Caching

This example uses data key caching (p. 55) in a command that encrypts a large number of files.

By default, the AWS Encryption CLI (and other versions of the AWS Encryption SDK) generates a unique data key for each file that it encrypts. Although using a unique data key for each operation is a cryptographic best practice, limited reuse of data keys is acceptable for some situations. If you are considering data key caching, consult with a security engineer to understand the security requirements of your application and determine security thresholds that are right for you.

In this example, data key caching speeds up the encryption operation by reducing the frequency of requests to the master key provider.

The command in this example encrypts a large directory with multiple subdirectories that contain a total of approximately 800 small log files. The first command saves the ARN of the CMK in a `cmkARN` variable. The second command encrypts all of the files in the input directory (recursively) and writes them to an archive directory. The command uses the `--suffix` parameter to specify the `.archive` suffix.

The `--caching` parameter enables data key caching. The **capacity** attribute, which limits the number of data keys in the cache, is set to 1, because serial file processing never uses more than one data key at a time. The **max_age** attribute, which determines how long the cached data key can used, is set to 10 seconds.

The optional **max_messages_encrypted** attribute is set to 10 messages, so a single data key is never used to encrypt more than 10 files. Limiting the number of files encrypted by each data key reduces the number of files that would be affected in the unlikely event that a data key was compromised.

To run this command on log files that your operating system generates, you might need administrator permissions (`sudo` in Linux; **Run as Administrator** in Windows).

Bash

```
$  cmkArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```

```
$  aws-encryption-cli --encrypt \
                      --input /var/log/httpd --recursive \
                      --output ~/archive --suffix .archive \
                      --master-keys key=$cmkArn \
                      --encryption-context class=log \
                      --suppress-metadata \
                      --caching capacity=1 max_age=10 max_messages_encrypted=10
```

PowerShell

```
PS C:\> $cmkArn = arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

PS C:\> aws-encryption-cli --encrypt `
                           --input C:\Windows\Logs --recursive `
                           --output $home\Archive --suffix '.archive' `
                           --master-keys key=$cmkARN `
                           --encryption-context class=log `
                           --suppress-metadata `
                           --caching capacity=1 max_age=10 max_messages_encrypted=10
```

To test the effect of data key caching, this example uses the Measure-Command cmdlet in PowerShell. When you run this example without data key caching, it takes about 25 seconds to complete. This process generates a new data key for each file in the directory.

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive' `
                                            --master-keys key=$cmkARN `
                                            --encryption-context class=log `
                                            --suppress-metadata }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 25
Milliseconds      : 453
Ticks             : 254531202
TotalDays         : 0.000294596298611111
TotalHours        : 0.00707031116666667
TotalMinutes      : 0.42421867
TotalSeconds      : 25.4531202
TotalMilliseconds : 25453.1202
```

Data key caching makes the process quicker, even when you limit each data key to a maximum of 10 files. The command now takes less than 12 seconds to complete and reduces the number of calls to the master key provider to 1/10 of the original value.

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive' `
                                            --master-keys key=$cmkARN `
                                            --encryption-context class=log `
                                            --suppress-metadata `
                                            --caching capacity=1 max_age=10
 max_messages_encrypted=10}


Days              : 0
Hours             : 0
```

```
Minutes           : 0
Seconds           : 11
Milliseconds      : 813
Ticks             : 118132640
TotalDays         : 0.000136727592592593
TotalHours        : 0.00328146222222222
TotalMinutes      : 0.196887733333333
TotalSeconds      : 11.813264
TotalMilliseconds : 11813.264
```

If you eliminate the `max_messages_encrypted` restriction, all files are encrypted under the same data key. This change increases the risk of reusing data keys without making the process much faster. However, it reduces the number of calls to the master key provider to 1.

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive' `
                                            --master-keys key=$cmkARN `
                                            --encryption-context class=log `
                                            --suppress-metadata `
                                            --caching capacity=1 max_age=10}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 10
Milliseconds      : 252
Ticks             : 102523367
TotalDays         : 0.000118661304398148
TotalHours        : 0.00284787130555556
TotalMinutes      : 0.170872278333333
TotalSeconds      : 10.2523367
TotalMilliseconds : 10252.3367
```

AWS Encryption SDK CLI Syntax and Parameter Reference

This topic provides syntax diagrams and brief parameter descriptions to help you use the AWS Encryption SDK Command Line Interface (CLI). For help with master keys and other parameters, see How to Use the AWS Encryption SDK Command Line Interface (p. 29). For example, see Examples of the AWS Encryption SDK Command Line Interface (p. 36). For complete documentation, see Read the Docs.

Topics

- AWS Encryption CLI Syntax (p. 49)
- AWS Encryption CLI Command Line Parameters (p. 51)
- Advanced Parameters (p. 54)

AWS Encryption CLI Syntax

These AWS Encryption CLI syntax diagrams show the syntax for each task that you perform with the AWS Encryption CLI.

Get Help

To get the full AWS Encryption CLI syntax with parameter descriptions, use `--help` or `-h`.

```
aws-encryption-cli (--help | -h)
```

Get the Version

To get the version number of your AWS Encryption CLI installation, use `--version`. Be sure to include the version when you ask questions, report problems, or share tips about using the AWS Encryption CLI.

```
aws-encryption-cli --version
```

Encrypt Data

The following syntax diagram shows the parameters that an encrypt command uses.

```
aws-encryption-cli --encrypt
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite] [--suffix
 [<suffix>]] [--encode]
                   --master-keys  [--master-keys ...]
                       key=<keyID> [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]
                   --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata
                   [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                   [--algorithm <algorithm_suite>]
                   [--caching <attributes>]
                   [--frame-length <length>]
                   [-v | -vv | -vvv | -vvvv]
                   [--quiet]
```

Decrypt Data

The following syntax diagram shows the parameters that a decrypt command uses.

```
aws-encryption-cli --decrypt
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite]  [--suffix
 [<suffix>]] [--encode]
                   --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata
                   [--master-keys [--master-keys ...]
                       [key=<keyID>] [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]]
                   [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                   [--caching <attributes>]
                   [--max-length <length>]
                   [-v | -vv | -vvv | -vvvv]
                   [--quiet]
```

Use Configuration Files

You can refer to configuration files that contain parameters and their values. This is equivalent to typing the parameters and values in the command. For an example, see How to Store Parameters in a Configuration File (p. 34).

```
aws-encryption-cli @<configuration_file>
```

```
# In a PowerShell console, use a backtick to escape the @.
aws-encryption-cli `@<configuration_file>
```

AWS Encryption CLI Command Line Parameters

This list provides a basic description of the AWS Encryption CLI command parameters. For a complete description, see the aws-encryption-sdk-cli documentation.

--encrypt (-e)

Encrypts the input data. Every command must have an `--encrypt` or `--decrypt` parameter.

--decrypt (-d)

Decrypts the input data. Every command must have an `--encrypt` or `--decrypt` parameter.

--master-keys (-m)

Specifies the master keys (p. 6) used in encryption and decryption operations. You can use multiple master keys parameters in each command.

The `--master-keys` parameter is required in encrypt commands. It is required in decrypt commands only when you are using a custom master key provider.

Attributes: The value of the `--master-keys` parameter consists of the following attributes. The format is `attribute_name=value`.

key

Identifies the master key. The format is a **key**=ID pair.

The **key** attribute is required in all encrypt commands. When you use an AWS KMS customer master key (CMK) in an encrypt command, the value of the **key** attribute can be a CMK ID or Amazon Resource Name (ARN), an alias, or an alias ARN.

The key attribute is required in decrypt commands when the master key provider is not AWS KMS. The **key** attribute is not permitted in commands that decrypt data that was encrypted under an AWS KMS CMK.

You can specify multiple **key** attributes in each `--master-keys` parameter value. However, any **provider**, **region**, and **profile** attributes apply to all master keys in the parameter value. To specify master keys with different attribute values, use multiple `--master-keys` parameters in the command.

provider

Identifies the master key provider (p. 7). The format is a **provider**=ID pair. The default value, **aws-kms**, represents AWS KMS. This attribute is required only when the master key provider is not AWS KMS.

region

Identifies the AWS Region of an AWS KMS CMK. This attribute is valid only for AWS KMS CMKs. It is used only when the **key** identifier does not specify a region; otherwise, it is ignored. When it is used, it overrides the default region in the AWS CLI named profile.

profile

Identifies an AWS CLI named profile. This attribute is valid only for AWS KMS CMKs. The region in the profile is used only when the key identifier does not specify a region and there is no **region** attribute in the command.

--input (-i)

Specifies the location of the data to encrypt or decrypt. This parameter is required. The value can be a path to a file or directory, or a file name pattern. If you are piping input to the command (stdin), use `-`.

If the input does not exist, the command completes successfully without error or warning.

--recursive (-r, -R)

Performs the operation on files in the input directory and its subdirectories. This parameter is required when the value of `--input` is a directory.

--decode

Decodes Base64-encoded input.

If you are decrypting a message that was encrypted and then encoded, you must decode the message before decrypting it. This parameter does that for you.

For example, if you used the `--encode` parameter in an encrypt command, use the `--decode` parameter in the corresponding decrypt command. You can also use this parameter to decode Base64-encoded input before you encrypt it.

--output (-o)

Specifies a destination for the output. This parameter is required. The value can be a file name, an existing directory, or `-`, which writes output to the command line (stdout).

If the specified output directory does not exist, the command fails. If the input contains subdirectories, the AWS Encryption CLI reproduces the subdirectories under the output directory that you specify.

By default, the AWS Encryption CLI overwrites files with the same name. To change that behavior, use the `--interactive` or `--no-overwrite` parameters. To suppress the overwrite warning, use the `--quiet` parameter.

Note
If a command that would overwrite an output file fails, the output file is deleted.

--interactive

Prompts before overwriting the file.

--no-overwrite

Does not overwrite files. Instead, if the output file exists, the AWS Encryption CLI skips the corresponding input.

--suffix

Specifies a custom file name suffix for files that the AWS Encryption CLI creates. To indicate no suffix, use the parameter with no value (`--suffix`).

By default, when the `--output` parameter does not specify a file name, the output file name has the same name as the input file name plus the suffix. The suffix for encrypt commands is `.encrypted`. The suffix for decrypt commands is `.decrypted`.

--encode

Applies Base64 (binary to text) encoding to the output. Encoding prevents the shell host program from misinterpreting non-ASCII characters in output text.

Use this parameter when writing encrypted output to stdout (`--output -`), especially in a PowerShell console, even when you are piping the output to another command or saving it in a variable.

--metadata-output

Specifies a location for metadata about the cryptographic operations. Enter a path and file name. If the directory does not exist, the command fails. To write the metadata to the command line (stdout), use `-`.

You cannot write command output (`--output`) and metadata output (`--metadata-output`) to stdout in the same command. Also, when the value of `--input` or `--output` is a directory (without file names), you cannot write the metadata output to the same directory or to any subdirectory of that directory.

If you specify an existing file, by default, the AWS Encryption CLI appends new metadata records to any content in the file. This feature lets you create a single file that contains the metadata for all of your cryptographic operations. To overwrite the content in an existing file, use the `--overwrite-metadata` parameter.

The AWS Encryption CLI returns a JSON-formatted metadata record for each encryption or decryption operation that the command performs. Each metadata record includes the full paths to the input and output file, the encryption context, the algorithm suite, and other valuable information that you can use to review the operation and verify that it meets your security standards.

--overwrite-metadata

Overwrites the content in the metadata output file. By default, the `--metadata-output` parameter appends metadata to any existing content in the file.

--suppress-metadata (-S)

Suppresses the metadata about the encryption or decryption operation.

--encryption-context (-c)

Specifies an encryption context (p. 33) for the operation. This parameter is not required, but it is recommended.

- In an `--encrypt` command, enter one or more `name=value` pairs. Use spaces to separate the pairs.
- In a decrypt command, enter `name=value` pairs, `name` elements with no values, or both.

If the `name` or `value` in a `name=value` pair includes spaces or special characters, enclose the entire pair in quotation marks. For example, `--encryption-context "department=software development"`.

--help (-h)

Prints usage and syntax at the command line.

--version

Gets the version of the AWS Encryption CLI.

-v | -vv | -vvv | -vvvv

Displays verbose information, warning, and debugging messages. The detail in the output increases with the number of `v`s in the parameter. The most detailed setting (`-vvvv`) returns debugging-level data from the AWS Encryption CLI and all of the components that it uses.

--quiet (-q)

Suppresses warning messages, such as the message that appears when you overwrite an output file.

Advanced Parameters

--algorithm

Specifies an alternate algorithm suite (p. 7). This parameter is optional and valid only in encrypt commands. By default, the AWS Encryption CLI uses the default algorithm suite for the AWS Encryption SDK, which is AES-GCM with an HKDF, an ECDSA signature, and a 256-bit encryption key. This algorithm suite is recommended for most encryption operations. For a list of valid values, see the values for the `algorithm` parameter in Read the Docs.

--frame-length

Creates output with specified frame length. Enter a value in bytes. This parameter is optional and valid only in encrypt commands.

--max-length

Indicates the maximum frame size (or maximum content length for non-framed messages) in bytes to read from encrypted messages. This parameter is optional and valid only in decrypt commands. It is designed to protect you from decrypting extremely large malicious ciphertext.

--caching

Enables the data key caching (p. 55) feature, which reuses data keys, instead of generating a new data key for each input file. This parameter supports an advanced scenario. Be sure to read the Data Key Caching (p. 55) documentation before using this feature.

The `--caching` parameter has the following attributes.

capacity (required)

Determines the maximum number of entries in the cache.

max_age (required)

Determine how long cache entries are used, beginning when they are added to the cache.

max_messages_encrypted

Determines the maximum number of messages that a cached entry can encrypt.

max_bytes_encrypted

Determines the maximum number of bytes that a cached entry can encrypt.

Data Key Caching

Data key caching stores data keys (p. 6) and related cryptographic material (p. 65) in a cache. When you encrypt or decrypt data, the AWS Encryption SDK looks for a matching data key in the cache. If it finds a match, it uses the cached data key rather than generating a new one. Data key caching can improve performance, reduce cost, and help you stay within service limits as your application scales.

Your application can benefit from data key caching if:

- It can reuse data keys.
- It generates numerous data keys.
- Your cryptographic operations are unacceptably slow, expensive, limited, or resource-intensive.

Caching can reduce your use of cryptographic services, such as AWS Key Management Service (AWS KMS). If you are hitting your AWS KMS requests-per-second limit, caching can help. Your application can use cached keys to service some of your data key requests instead of calling AWS KMS. (You can also create a case in the AWS Support Center to raise the limit for your account.)

The AWS Encryption SDK helps you to create and manage your data key cache. It provides a LocalCryptoMaterialsCache (p. 64) and a caching cryptographic materials manager (p. 65) that interacts with the cache and enforces security thresholds (p. 60) that you set. Working together, these components help you to benefit from the efficiency of reusing data keys while maintaining the security of your system.

Data key caching is an optional feature of the AWS Encryption SDK that you should use cautiously. By default, the AWS Encryption SDK generates a new data key for every encryption operation. This technique supports cryptographic best practices, which discourage excessive reuse of data keys. In general, use data key caching only when it is required to meet your performance goals. Then, use the data key caching security thresholds (p. 60) to ensure that you use the minimum amount of caching required to meet your cost and performance goals.

For a detailed discussion of these security tradeoffs, see AWS Encryption SDK: How to Decide if Data Key Caching is Right for Your Application in the AWS Security Blog.

Topics

- How to Implement Data Key Caching (p. 55)
- Setting Cache Security Thresholds (p. 60)
- Data Key Caching Details (p. 61)
- Data Key Caching Example (p. 66)

How to Implement Data Key Caching

This topic shows you how to implement data key caching in your application. It takes you through the process step by step. Then, it combines the steps in a simple example that uses data key caching in an operation to encrypt a string.

Topics

- Implement Data Key Caching: Step-by-Step (p. 56)
- Data Key Caching Example: Encrypt a String (p. 58)

Implement Data Key Caching: Step-by-Step

These step-by-step instructions show you how to create the components that you need to implement data key caching.

- Create a data key cache (p. 64), such as a LocalCryptoMaterialsCache.

  Java

  ```
  //Cache capacity (maximum number of entries) is required
  int MAX_CACHE_SIZE = 10;

  CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(MAX_CACHE_SIZE);
  ```

  Python

  ```
  # Cache capacity (maximum number of entries) is required
  MAX_CACHE_SIZE = 10

  cache = LocalCryptoMaterialsCache(MAX_CACHE_SIZE)
  ```

- Create a master key provider (p. 7). This example uses an AWS Key Management Service (AWS KMS) master key provider.

  Java

  ```
  //Create a KMS master key provider
  //  The input is the Amazon Resource Name (ARN)
  //  of a KMS customer master key (CMK)

  MasterKeyProvider<KmsMasterKey> keyProvider = new KmsMasterKeyProvider(kmsCmkArn);
  ```

  Python

  ```
  # Create a KMS master key provider
  #  The input is the Amazon Resource Name (ARN)
  #  of a KMS customer master key (CMK)

  key_provider = aws_encryption_sdk.KMSMasterKeyProvider(key_ids=[kms_cmk_arn])
  ```

- Create a caching cryptographic materials manager (p. 65) (caching CMM).

  Associate your caching CMM with your cache and master key provider. Then, set cache security thresholds (p. 60) on the caching CMM.

Java

```
/*
 * Security thresholds
 *   Max entry age is required.
 *   Max messages (and max bytes) per entry are optional
 */
int MAX_ENTRY_AGE_SECONDS = 60;
int MAX_ENTRY_MSGS = 10;

//Create a caching CMM
CryptoMaterialsManager cachingCmm =
    CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                                 .withCache(cache)
                                 .withMaxAge(MAX_ENTRY_AGE_SECONDS, TimeUnit.SECONDS)
                                 .withMessageUseLimit(MAX_ENTRY_MSGS)
    .build();
```

Python

```
# Security thresholds
#   Max entry age is required.
#   Max messages (and max bytes) per entry are optional
#
MAX_ENTRY_AGE_SECONDS = 60.0
MAX_ENTRY_MESSAGES = 10

# Create a caching CMM
caching_cmm = CachingCryptoMaterialsManager(
    master_key_provider=key_provider,
    cache=cache,
    max_age=MAX_ENTRY_AGE_SECONDS,
    max_messages_encrypted=MAX_ENTRY_MESSAGES
)
```

That's all you need to do. Then, let the AWS Encryption SDK manage the cache for you, or add your own cache management logic.

When you want to use data key caching in a call to encrypt or decrypt data, specify your caching CMM instead of a master key provider or other CMM.

Note
If you are encrypting data streams, or any data of unknown size, be sure to specify the data size in the request. The AWS Encryption SDK does not use data key caching when encrypting data of unknown size.

Java

```
// When the call to encryptData specifies a caching CMM,
// the encryption operation uses the data key cache
//
final AwsCrypto encryptionSdk = new AwsCrypto();
byte[] message = encryptionSdk.encryptData(cachingCmm, plaintext_source).getResult();
```

Python

```
# When the call to encrypt specifies a caching CMM,
# the encryption operation uses the data key cache
#
encrypted_message, header = aws_encryption_sdk.encrypt(
```

```
    source=plaintext_source,
    materials_manager=caching_cmm
)
```

Data Key Caching Example: Encrypt a String

This simple code example uses data key caching when encrypting a string. It combines the code from the step-by-step procedure (p. 56) into test code that you can run.

The example creates a LocalCryptoMaterialsCache (p. 64) and a master key provider (p. 7) for an AWS KMS customer master key (CMK). Then, it uses the cache and master key provider to create a caching CMM with appropriate security thresholds (p. 60). The encryption request specifies the caching CMM, the plaintext data to encrypt, and an encryption context (p. 66).

To run the example, you need to supply the Amazon Resource Name (ARN) of a KMS CMK. Be sure that you have permission to use the CMK to generate a data key.

For more detailed, real-world examples of creating and using a data key cache, see Data Key Caching Example in Java (p. 68) and Data Key Caching Example in Python (p. 72).

Java

```
/*
 * Copyright 2017 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
 *
 * http://aws.amazon.com/apache2.0
 *
 * 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.
 */


import java.nio.charset.StandardCharsets;
import java.util.Collections;
import java.util.Map;
import java.util.concurrent.TimeUnit;

import javax.xml.bind.DatatypeConverter;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoMaterialsManager;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.CryptoMaterialsCache;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kms.KmsMasterKey;
import com.amazonaws.encryptionsdk.kms.KmsMasterKeyProvider;

/**
 * <p>
 * Encrypts a string using an AWS KMS customer master key (CMK) and data key caching
 *
 * <p>
 * Arguments:
```

```
 * <ol>
 * <li>KMS CMK ARN: To find the Amazon Resource Name of your AWS KMS customer master
 key (CMK),
 *     see 'Viewing Keys' at http://docs.aws.amazon.com/kms/latest/developerguide/
viewing-keys.html
 * <li>Max entry age: Maximum time (in seconds) that a cached entry can be used
 * <li>Cache capacity: Maximum number of entries in the cache
 * </ol>
 */
public class SimpleDataKeyCachingExample {
    /*
     * Security thresholds
     *   Max entry age is required.
     *   Max messages (and max bytes) per data key are optional
     */
    private static final int MAX_ENTRY_MSGS = 100;

    public static byte[] encryptWithCaching(String kmsCmkArn, int maxEntryAge, int
 cacheCapacity) {
        // Plaintext data to be encrypted
        byte[] myData = "My plaintext data".getBytes(StandardCharsets.UTF_8);

        // Encryption context
        final Map<String, String> encryptionContext =
 Collections.singletonMap("purpose", "test");

        // Create a master key provider
        MasterKeyProvider<KmsMasterKey> keyProvider = new
 KmsMasterKeyProvider(kmsCmkArn);

        // Create a cache
        CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(cacheCapacity);

        // Create a caching CMM
        CryptoMaterialsManager cachingCmm =

 CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                                        .withCache(cache)
                                        .withMaxAge(maxEntryAge, TimeUnit.SECONDS)
                                        .withMessageUseLimit(MAX_ENTRY_MSGS)
            .build();

        // When the call to encryptData specifies a caching CMM,
        // the encryption operation uses the data key cache
        //
        final AwsCrypto encryptionSdk = new AwsCrypto();
        return encryptionSdk.encryptData(cachingCmm, myData,
 encryptionContext).getResult();
}
```

Python

```
# Copyright 2017 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
#
# http://aws.amazon.com/apache2.0/
#
# 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.
```

```
"""Example of basic configuration and use of data key caching."""
import aws_encryption_sdk


def encrypt_with_caching(kms_cmk_arn, max_age_in_cache, cache_capacity):
    """Encrypts a string using an AWS KMS customer master key (CMK) and data key
 caching.

    :param str kms_cmk_arn: Amazon Resource Name (ARN) of the KMS customer master key
    :param float max_age_in_cache: Maximum time in seconds that a cached entry can be
 used
    :param int cache_capacity: Maximum number of entries in the cache
    """
    # Data to be encrypted
    my_data = 'My plaintext data'

    # Security thresholds
    #   Max messages (and max bytes) per data key are optional
    MAX_ENTRY_MESSAGES = 100

    # Create an encryption context.
    encryption_context = {'purpose': 'test'}

    # Create a master key provider for the KMS master key
    key_provider = aws_encryption_sdk.KMSMasterKeyProvider(key_ids=[kms_cmk_arn])

    # Create a cache
    cache = aws_encryption_sdk.LocalCryptoMaterialsCache(cache_capacity)

    # Create a caching CMM
    caching_cmm = aws_encryption_sdk.CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=max_age_in_cache,
        max_messages_encrypted=MAX_ENTRY_MESSAGES
    )

    # When the encrypt request specifies a caching CMM,
    # the encryption operation uses the data key cache
    encrypted_message, _header = aws_encryption_sdk.encrypt(
        source=my_data,
        materials_manager=caching_cmm,
        encryption_context=encryption_context
    )

    return encrypted_message
```

Setting Cache Security Thresholds

When you implement data key caching, you need to configure the security thresholds that the caching CMM (p. 65) enforces.

The security thresholds help you to limit how long each cached data key is used and how much data is protected under each data key. The caching CMM returns cached data keys only when the cache entry conforms to all of the security thresholds. If the cache entry exceeds any threshold, the entry is not used for the current operation and it is evicted from the cache.

As a rule, use the minimum amount of caching that is required to meet your cost and performance goals.

The AWS Encryption SDK only caches data keys that are encrypted by using a key derivation function. Also, it establishes upper limits for the threshold values. These restrictions ensure that data keys are not reused beyond their cryptographic limits. However, because your plaintext data keys are cached (in memory, by default), try to minimize the time that the keys are saved . Also, try to limit the data that might be exposed if a key is compromised.

For examples of setting cache security thresholds, see AWS Encryption SDK: How to Decide if Data Key Caching is Right for Your Application in the AWS Security Blog.

> **Note**
> The caching CMM enforces all of the following thresholds. If you do not specify an optional value, the caching CMM uses the default value.
> To disable data key caching temporarily, do not set the cache capacity (p. 64) or security thresholds to 0. Instead, use the *null cryptographic materials cache* (NullCryptoMaterialsCache) that the AWS Encryption SDK provides. The NullCryptoMaterialsCache returns a miss for every get request and does not respond to put requests. For more information, see the SDK for your programming language (p. 12).

Maximum age (required)

Determines how long a cached entry can be used, beginning when it was added. This value is required. Enter a value greater than 0. There is no maximum value.

The LocalCryptoMaterialsCache tries to evict cache entries as soon as possible after they reach the maximum age value. Other conforming caches might perform differently.

Use the shortest interval that still allows your application to benefit from the cache. You can use the maximum age threshold like a key rotation policy. Use it to limit reuse of data keys, minimize exposure of cryptographic materials, and evict data keys whose policies might have changed while they were cached.

Maximum messages encrypted (optional)

Specifies the maximum number of messages that a cached data key can encrypt. This value is optional. Enter a value between 1 and 2^32 messages. The default value is 2^32 messages.

Set the number of messages protected by each cached key to be large enough to get value from reuse, but small enough to limit the number of messages that might be exposed if a key is compromised.

Maximum bytes encrypted (optional)

Specifies the maximum number of bytes that a cached data key can encrypt. This value is optional. Enter a value between 0 and 2^63 - 1. The default value is 2^63 - 1. A value of 0 lets you encrypt empty message strings.

The first use of each data key (before caching) is exempt from this threshold. Also, to enforce this threshold, requests to encrypt data of unknown size, such as streamed data with no length specifier, do not use the data key cache.

The bytes in the current request are included when evaluating this threshold. If the bytes processed, plus current bytes, exceed the threshold, the cached data key is evicted from the cache, even though it might have been used on a smaller request.

Data Key Caching Details

Most applications can use the default implementation of data key caching without writing custom code. This section describes the default implementation and some details about options.

Topics

- How Data Key Caching Works (p. 62)
- Creating a Cryptographic Materials Cache (p. 64)
- Creating a Caching Cryptographic Materials Manager (p. 65)
- What Is in a Data Key Cache Entry? (p. 65)
- Encryption Context: How to Select Cache Entries (p. 66)

How Data Key Caching Works

When you use data key caching in a request to encrypt or decrypt data, the AWS Encryption SDK first searches the cache for a data key that matches the request. If it finds a valid match, it uses the cached data key to encrypt the data. Otherwise, it generates a new data key, just as it would without the cache.

In addition to a cache, data key caching uses a caching cryptographic materials manager (p. 65) (caching CMM). The caching CMM is a specialized cryptographic materials manager (CMM) (p. 7) that interacts with a cache (p. 64) and an underlying CMM (p. 7) or master key provider (p. 7). The caching CMM caches the data keys that its underlying CMM (or master key provider) returns. The caching CMM also enforces cache security thresholds that you set.

To prevent the wrong data key from being selected from the cache, each caching CMM requires that several properties of each cached data key match the materials request, as follows:

- For encryption material requests, the cached entry and the request must have the same algorithm suite (p. 7), encryption context (p. 66) (even when empty), and partition name (a string that identifies the caching CMM).
- For decryption material requests, the cached entry and the request must have the same algorithm suite (p. 7), encryption context (p. 66) (even when empty), and partition name (a string that identifies the caching CMM).

> **Note**
> The AWS Encryption SDK caches data keys only when the algorithm suite (p. 7) uses a key derivation function.
> Data key caching is not used for data of unknown size, such as streamed data. This allows the caching CMM to properly enforce the maximum bytes threshold (p. 60). To avoid this behavior, add the data length to the encryption request.

The following workflows show how a request to encrypt data is processed with and without data key caching. They show how the caching components that you create, including the cache and the caching CMM, are used in the process.

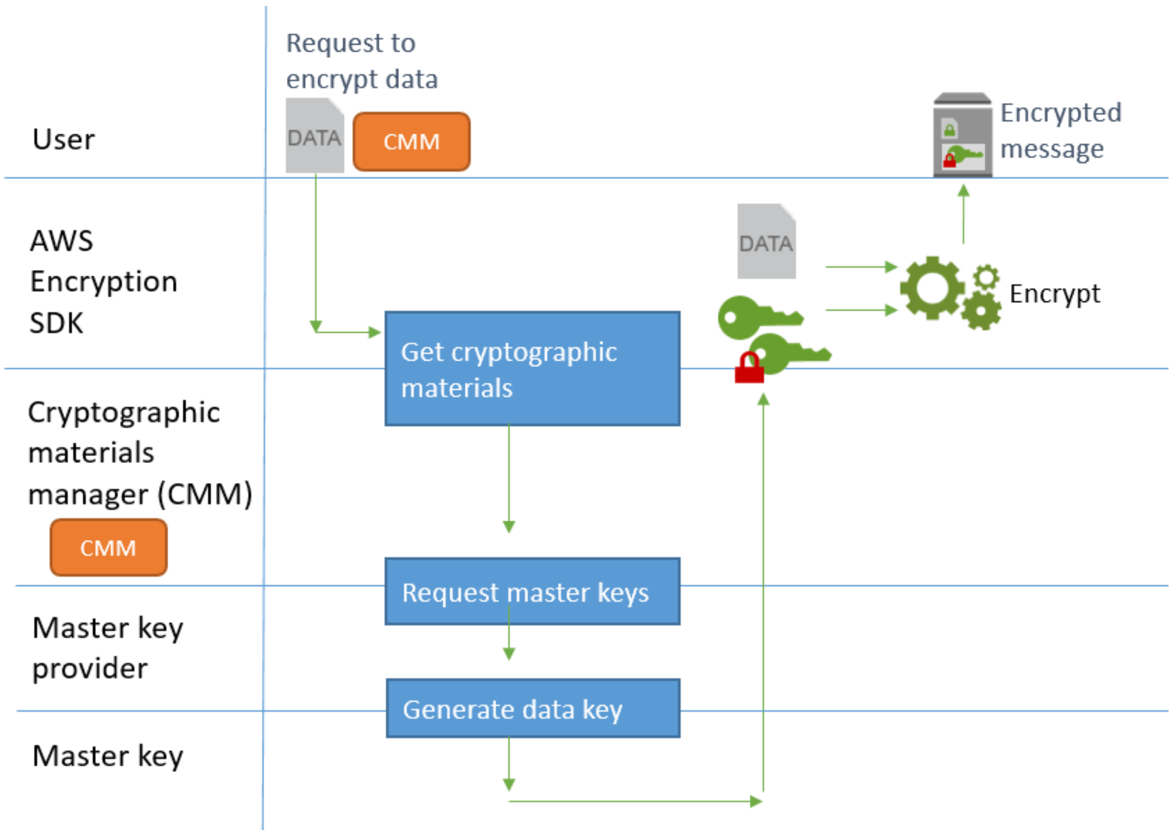
Encrypt Data without Caching

To generate a data key without caching:

1. An application asks the AWS Encryption SDK to encrypt data.

   The request specifies a cryptographic materials manager (CMM) or master key provider. If you specify a master key provider, the AWS Encryption SDK creates a default CMM that interacts with the master key provider you specified.
2. The AWS Encryption SDK asks the CMM for a data key to encrypt the data (get cryptographic materials).
3. The CMM asks its master key provider for master keys (p. 6) (or objects that represent master keys). Then, it uses the master keys to generate a new data key (p. 6). This might involve a call to a cryptographic service, such as AWS Key Management Service (AWS KMS). The CMM returns plaintext and encrypted copies of the data key to the AWS Encryption SDK.

4. The AWS Encryption SDK uses the plaintext data key to encrypt the data and it returns an encrypted message (p. 8) to the user.



Encrypt Data with Caching

To generate a data key with data key caching:

1. An application asks the AWS Encryption SDK to encrypt data.

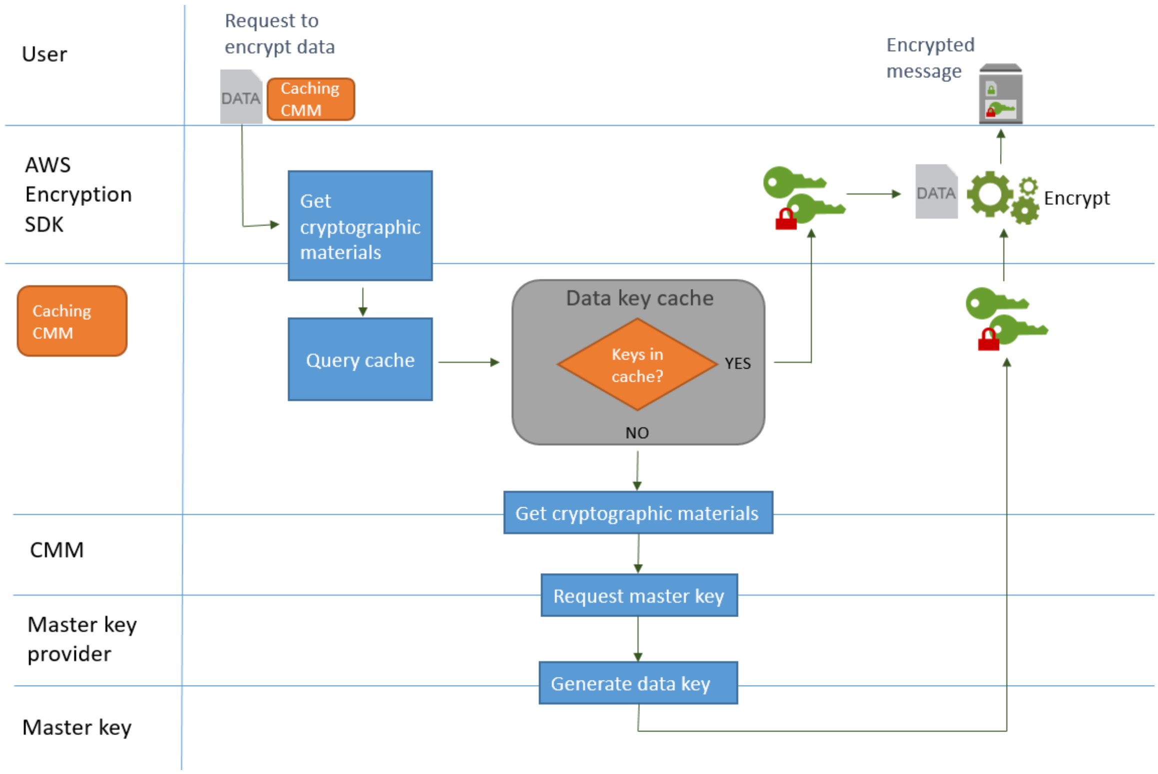
   The request specifies a caching cryptographic materials manager (caching CMM) (p. 65) that is associated with a default cryptographic materials manager (CMM) or a master key provider. If you specify a master key provider, the SDK creates a default CMM for you.
2. The SDK asks the specified caching CMM for a data key to encrypt the data (get cryptographic materials).
3. The caching CMM requests a data key from the cache.
   a. If the cache finds a match, it updates the age and use values of the matched cache entry, and returns the cached data key to the caching CMM.

      If the cache entry conforms to its security thresholds (p. 60), the caching CMM returns it to the SDK. Otherwise, it tells the cache to evict the entry and proceeds as though there was no match.
   b. If the cache cannot find a valid match, the caching CMM asks its underlying CMM to generate a new data key.

      The CMM gets master keys (or objects that represent master keys) from its master key provider and it uses them to generate a new data key. This might involve a call to a service, such as AWS Key Management Service. The CMM returns the plaintext and encrypted copies of the data key to the caching CMM.

The caching CMM saves the new data key in the cache.

4. The caching CMM returns plaintext and encrypted copies of the data key to the AWS Encryption SDK.
5. The AWS Encryption SDK uses the data key to encrypt the data and it returns an encrypted message (p. 8) to the user.



Creating a Cryptographic Materials Cache

The AWS Encryption SDK defines the requirements for a cryptographic materials cache used in data key caching. It also provides *LocalCryptoMaterialsCache*, a configurable, in-memory, least recently used (LRU) cache, and a null cryptographic materials cache for testing.

LocalCryptoMaterialsCache includes logic for basic cache management, including adding, evicting, and matching cached entries, and maintaining the cache. You don't need to write any custom cache management logic. You can use LocalCryptoMaterialsCache as is, customize it, or substitute any compatible cache.

When you create a LocalCryptoMaterialsCache, you set its *capacity*, that is, the maximum number of entries that the cache can hold. This setting helps you to design an efficient cache with limited data key reuse.

The AWS Encryption SDK also provides a *null cryptographic materials cache* (NullCryptoMaterialsCache). The NullCryptoMaterialsCache returns a miss for all get operations and does not respond to put operations. You can use the NullCryptoMaterialsCache in testing or to temporarily disable caching in an application that includes caching code.

In the AWS Encryption SDK, each cryptographic materials cache is associated with a caching cryptographic materials manager (p. 65) (caching CMM). The caching CMM gets data keys from the cache, puts data keys in the cache, and enforces security thresholds (p. 60) that you set. When you create a caching CMM, you specify the cache that it uses and the underlying CMM or master key provider that generates the data keys that it caches.

Creating a Caching Cryptographic Materials Manager

To enable data key caching, you create a cache (p. 64) and a *caching cryptographic materials manager* (caching CMM). Then, in your requests to encrypt or decrypt data, you specify a caching CMM, instead of a standard cryptographic materials manager (CMM) (p. 7) or master key provider (p. 7) .

There are two types of CMMs. Both get data keys (and related cryptographic material), but in different ways, as follows:

- A CMM is associated with a master key provider. When the SDK asks the CMM for data keys (get encryption materials), the CMM gets master keys (or objects that represent master keys) from its master key provider. Then, it uses the master keys to generate, encrypt, or decrypt the data keys.

- A caching CMM is associated with one cache, such as a LocalCryptoMaterialsCache (p. 64), and a CMM or master key provider. (If you specify a master key provider, the SDK creates a default CMM for the master key provider.) When the SDK asks the caching CMM for data keys, the caching CMM tries to get them from the cache. If it cannot find a valid, matching data key, the caching CMM asks its underlying CMM for the data keys. Then, it caches those data keys before returning them to the caller.

The caching CMM also enforces security thresholds (p. 60) that you set for each cache entry. Because the security thresholds are set in and enforced by the caching CMM, you can use any compatible cache, even if the cache is not designed for sensitive material.

For details about creating and managing CMMs and caching CMMs in your application, see the SDK for your programming language (p. 12).

What Is in a Data Key Cache Entry?

Data key caching stores data keys and related cryptographic materials in a cache. Each entry includes the elements listed below. You might find this information useful when you're deciding whether to use the data key caching feature, and when you're setting security thresholds on a caching cryptographic materials manager (caching CMM).

Cached Entries for Encryption Requests

The entries that are added to a data key cache as a result of a encryption operation include the following elements:

- Plaintext data key
- Encrypted data keys (one or more)
- Encryption context (p. 66)
- Message signing key (if one is used)
- Algorithm suite (p. 7)
- Metadata, including usage counters for enforcing security thresholds

Cached Entries for Decryption Requests

The entries that are added to a data key cache as a result of a decryption operation include the following elements:

- Plaintext data key
- Signature verification key (if one is used)
- Metadata, including usage counters for enforcing security thresholds

Encryption Context: How to Select Cache Entries

You can specify an encryption context in any request to encrypt data. However, the encryption context plays a special role in data key caching. It lets you create subgroups of data keys in your cache, even when the data keys originate from the same caching CMM.

An encryption context (p. 8) is a set of key-value pairs that contain arbitrary nonsecret data. During encryption, the encryption context is cryptographically bound to the encrypted data so that the same encryption context is required to decrypt the data. In the AWS Encryption SDK, the encryption context is stored in the encrypted message (p. 8) along with the encrypted data and data keys.

When you use a data key cache, you can also use the encryption context to select particular cached data keys for your encryption operations. The encryption context is saved in the cache entry with the data key (it's part of the cache entry ID). Cached data keys are reused only when their encryption contexts match. If you want to reuse certain data keys for an encryption request, specify the same encryption context. If you want to avoid those data keys, specify a different encryption context.

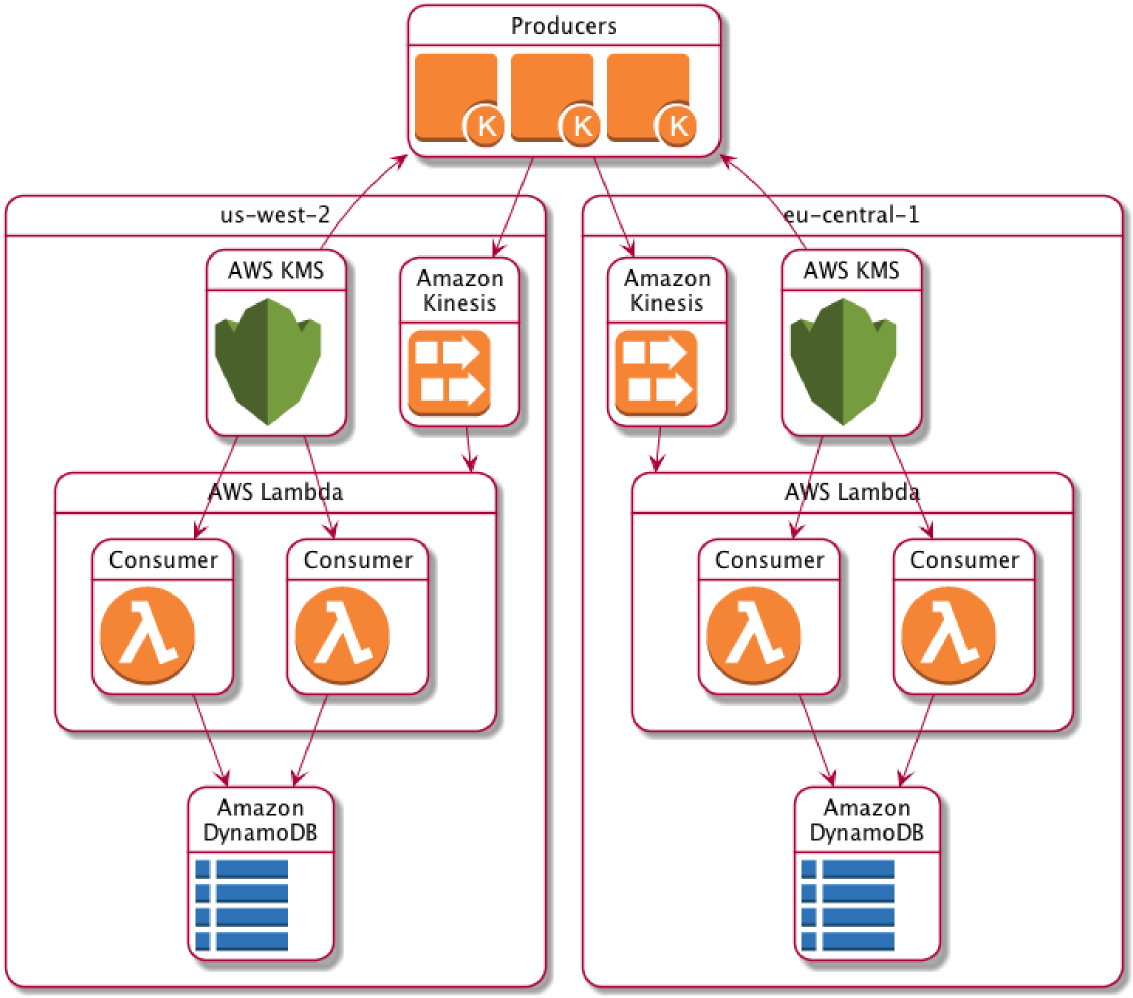
The encryption context is always optional, but recommended. If you don't specify an encryption context in your request, an empty encryption context is included in the cache entry identifier and matched to each request.

Data Key Caching Example

This example uses data key caching (p. 55) with a LocalCryptoMaterialsCache (p. 64) to speed up an application in which data generated by multiple devices is encrypted and stored in different regions.

In this scenario, multiple data producers generate data, encrypt it, and write to a Kinesis stream in each region. AWS Lambda functions (consumers) decrypt the streams and write plaintext data to a DynamoDB table in the region. Data producers and consumers use the AWS Encryption SDK and a KMS master key provider (p. 7). To reduce calls to KMS, each producer and consumer has their own LocalCryptoMaterialsCache.

You can find the source code for these examples in Java (p. 68) and Python (p. 72). The sample also includes a AWS CloudFormation template that defines the resources for the samples.



LocalCryptoMaterialsCache Results

The following table shows that LocalCryptoMaterialsCache reduces the total calls to KMS (per second per region) in this example to 1% of its original value.

Producer requests

	Requests per second per client			Clients per region	Average requests per second per region
	Generate data key (us-west-2)	Encrypt data key (eu-central-1)	Total (per region)		
No cache	1	1	1	500	500
LocalCryptoMaterialsCache	1 rps / 100 uses	1 rps / 100 uses	1 rps / 100 uses	500	5

Consumer requests

	Requests per second per client			Client per region	Average requests per second per region
	Decrypt data key	Producers	Total		
No cache	1 rps per producer	500	500	2	1,000
LocalCryptoMaterialsCache	1 rps per producer / 100 uses	500	5	2	10

Data Key Caching Example in Java

This code sample creates a basic implementation of data key caching with a LocalCryptoMaterialsCache (p. 64) in Java. For details about the Java implementation of the AWS Encryption SDK, see AWS Encryption SDK for Java (p. 12).

The code creates two instances of a LocalCryptoMaterialsCache; one for data producers that are encrypting data and another for data consumers (Lambda functions) that are decrypting data. For implementation details, see the Javadoc for the AWS Encryption SDK.

Producer

The producer gets a map, converts it to JSON, uses the AWS Encryption SDK to encrypt it, and pushes the ciphertext record to a Kinesis stream in each region.

The code defines a caching cryptographic materials manager (p. 65) (caching CMM) and associates it with a LocalCryptoMaterialsCache (p. 64) and an underlying KMS master key provider (p. 7). The caching CMM caches the data keys (and related cryptographic materials (p. 65)) from the master key provider. It also interacts with the cache on behalf of the SDK and enforces security thresholds that you set.

Because the call to the `encryptData` method specifies a caching CMM, instead of a regular cryptographic materials manager (CMM) (p. 7) or master key provider, the method will use data key caching.

```
/*
 * Copyright 2017 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
 *
 * http://aws.amazon.com/apache2.0
 *
 * 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.kinesisdatakeycaching;

import java.nio.ByteBuffer;
import java.util.ArrayList;
```

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kms.KmsMasterKey;
import com.amazonaws.encryptionsdk.kms.KmsMasterKeyProvider;
import com.amazonaws.encryptionsdk.multi.MultipleProviderFactory;
import com.amazonaws.regions.Region;
import com.amazonaws.services.kinesis.AmazonKinesis;
import com.amazonaws.services.kinesis.AmazonKinesisClientBuilder;
import com.amazonaws.util.json.Jackson;

/**
 * Pushes data to Kinesis Streams in multiple regions.
 */
public class MultiRegionRecordPusher {
    private static long MAX_ENTRY_AGE_MILLISECONDS = 300000;
    private static long MAX_ENTRY_USES = 100;
    private static int MAX_CACHE_ENTRIES = 100;
    private final String streamName_;
    private ArrayList<AmazonKinesis> kinesisClients_;
    private CachingCryptoMaterialsManager cachingMaterialsManager_;
    private AwsCrypto crypto_;

    /**
     * Creates an instance of this object with Kinesis clients for all target regions
     * and a cached key provider containing KMS master keys in all target regions.
     */
    public MultiRegionRecordPusher(final Region[] regions, final String kmsAliasName, final
 String streamName){
        streamName_ = streamName;
        crypto_ = new AwsCrypto();
        kinesisClients_ = new ArrayList<AmazonKinesis>();

        DefaultAWSCredentialsProviderChain credentialsProvider = new
 DefaultAWSCredentialsProviderChain();
        ClientConfiguration clientConfig = new ClientConfiguration();

        // Build KmsMasterKey and AmazonKinesisClient objects for each target region
        List<KmsMasterKey> masterKeys = new ArrayList<KmsMasterKey>();
        for (Region region : regions) {
            kinesisClients_.add(AmazonKinesisClientBuilder.standard()
                    .withCredentials(credentialsProvider)
                    .withRegion(region.getName())
                    .build());

            KmsMasterKey regionMasterKey = new KmsMasterKeyProvider(
                credentialsProvider,
                region,
                clientConfig,
                kmsAliasName
            ).getMasterKey(kmsAliasName);

            masterKeys.add(regionMasterKey);
        }

        // Collect KmsMasterKey objects into single provider and add cache
```

```
        MasterKeyProvider<?> masterKeyProvider =
 MultipleProviderFactory.buildMultiProvider(
                KmsMasterKey.class,
                masterKeys
        );

        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()
            .withMasterKeyProvider(masterKeyProvider)
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .withMessageUseLimit(MAX_ENTRY_USES)
            .build();
    }

    /**
     * JSON serializes and encrypts the received record data and pushes it to all target
 streams.
     */
    public void putRecord(final Map<Object, Object> data){
        String partitionKey = UUID.randomUUID().toString();
        Map<String, String> encryptionContext = new HashMap<String, String>();
        encryptionContext.put("stream", streamName_);

        // JSON serialize data
        String jsonData = Jackson.toJsonString(data);

        // Encrypt data
        CryptoResult<byte[], ?> result = crypto_.encryptData(
            cachingMaterialsManager_,
            jsonData.getBytes(),
            encryptionContext
        );
        byte[] encryptedData = result.getResult();

        // Put records to Kinesis stream in all regions
        for (AmazonKinesis regionalKinesisClient : kinesisClients_) {
            regionalKinesisClient.putRecord(
                streamName_,
                ByteBuffer.wrap(encryptedData),
                partitionKey
            );
        }
    }
}
```

Consumer

The data consumer is an AWS Lambda function that is triggered by Kinesis events. It decrypts and deserializes each record, and writes the plaintext record to a DynamoDB table in the same region.

Like the producer code, the consumer code enables data key caching by using a caching cryptographic materials manager (caching CMM) in calls to the `decryptData` method.

```
/*
 * Copyright 2017 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
 *
 * http://aws.amazon.com/apache2.0
 *
 * 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.kinesisdatakeycaching;

import java.io.UnsupportedEncodingException;
import java.nio.ByteBuffer;
import java.util.concurrent.TimeUnit;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kms.KmsMasterKey;
import com.amazonaws.encryptionsdk.kms.KmsMasterKeyProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRecord;
import com.amazonaws.util.BinaryUtils;

/**
 * Decrypts all incoming Kinesis records and writes records to DynamoDB.
 */
public class LambdaDecryptAndWrite {
    private static final long MAX_ENTRY_AGE_MILLISECONDS = 600000;
    private static final int MAX_CACHE_ENTRIES = 100;
    private CachingCryptoMaterialsManager cachingMaterialsManager_;
    private AwsCrypto crypto_;
    private Table table_;

    /**
     * Because the cache is used only for decryption, the code doesn't set
     * the max bytes or max message security thresholds that are are enforced
     * only on on data keys used for encryption.
     */
    public LambdaDecryptAndWrite() {
     String cmkArn = System.getenv("CMK_ARN");
        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()
            .withMasterKeyProvider(new KmsMasterKeyProvider(cmkArn))
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .build();

        crypto_ = new AwsCrypto();
        String tableName = System.getenv("TABLE_NAME");
        DynamoDB dynamodb = new DynamoDB(AmazonDynamoDBClientBuilder.defaultClient());
        table_ = dynamodb.getTable(tableName);
    }

    /**
     *
     * @param event
     * @param context
     */
    public void handleRequest(KinesisEvent event, Context context) throws
 UnsupportedEncodingException{
        for (KinesisEventRecord record : event.getRecords()) {
            ByteBuffer ciphertextBuffer = record.getKinesis().getData();
            byte[] ciphertext = BinaryUtils.copyAllBytesFrom(ciphertextBuffer);

            // Decrypt and unpack record
```

```
            CryptoResult<byte[], ?> plaintextResult =
 crypto_.decryptData(cachingMaterialsManager_, ciphertext);

            // Verify the encryption context value
            String streamArn = record.getEventSourceARN();
            String streamName = streamArn.substring(streamArn.indexOf("/") + 1);
            if (!streamName.equals(plaintextResult.getEncryptionContext().get("stream"))) {
                throw new IllegalStateException("Wrong Encryption Context!");
            }

            // Write record to DynamoDB
            String jsonItem = new String(plaintextResult.getResult(), "UTF-8");
            System.out.println(jsonItem);
            table_.putItem(Item.fromJSON(jsonItem));
        }
    }
}
```

Data Key Caching Example in Python

This code sample creates a basic implementation of data key caching with a LocalCryptoMaterialsCache (p. 64) in Python. For details about the Python implementation of the AWS Encryption SDK, see AWS Encryption SDK for Python (p. 20).

The code creates two instances of a LocalCryptoMaterialsCache; one for data producers that are encrypting data and another for data consumers (Lambda functions) that are decrypting data. For implementation details, see the Python documentation for the AWS Encryption SDK.

Producer

The producer gets a map, converts it to JSON, uses the AWS Encryption SDK to encrypt it, and pushes the ciphertext record to an Kinesis stream in each region.

The code defines a caching cryptographic materials manager (p. 65) (caching CMM) and associates it with a LocalCryptoMaterialsCache (p. 64) and an underlying KMS master key provider (p. 7). The caching CMM caches the data keys (and related cryptographic materials (p. 65)) from the master key provider. It also interacts with the cache on behalf of the SDK and enforces security thresholds that you set.

Because the call to the `encrypt` method specifies a caching CMM, instead of a regular cryptographic materials manager (CMM) (p. 7) or master key provider, the method will use data key caching.

```
"""
Copyright 2017 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.
"""
import json
import uuid

from aws_encryption_sdk import encrypt, KMSMasterKeyProvider,
 CachingCryptoMaterialsManager, LocalCryptoMaterialsCache
```

```
from aws_encryption_sdk.key_providers.kms import KMSMasterKey
import boto3


class MultiRegionRecordPusher(object):
    """Pushes data to Kinesis Streams in multiple regions."""
    CACHE_CAPACITY = 100
    MAX_ENTRY_AGE_SECONDS = 300.0
    MAX_ENTRY_MESSAGES_ENCRYPTED = 100

    def __init__(self, regions, kms_alias_name, stream_name):
        self._kinesis_clients = []
        self._stream_name = stream_name

        # Set up KMSMasterKeyProvider with cache
        _key_provider = KMSMasterKeyProvider()

        # Add MasterKey and Kinesis client for each region
        for region in regions:
            self._kinesis_clients.append(boto3.client('kinesis', region_name=region))
            regional_master_key = KMSMasterKey(
                client=boto3.client('kms', region_name=region),
                key_id=kms_alias_name
            )
            _key_provider.add_master_key_provider(regional_master_key)

        cache = LocalCryptoMaterialsCache(capacity=self.CACHE_CAPACITY)
        self._materials_manager = CachingCryptoMaterialsManager(
            master_key_provider=_key_provider,
            cache=cache,
            max_age=self.MAX_ENTRY_AGE_SECONDS,
            max_messages_encrypted=self.MAX_ENTRY_MESSAGES_ENCRYPTED
        )

    def put_record(self, record_data):
        """JSON serializes and encrypts the received record data and pushes it to all
 target streams.

        :param dict record_data: Data to write to stream
        """
        # Kinesis partition key to randomize write load across stream shards
        partition_key = uuid.uuid4().hex

        encryption_context = {'stream': self._stream_name}

        # JSON serialize data
        json_data = json.dumps(record_data)

        # Encrypt data
        encrypted_data, _header = encrypt(
            source=json_data,
            materials_manager=self._materials_manager,
            encryption_context=encryption_context
        )

        # Put records to Kinesis stream in all regions
        for client in self._kinesis_clients:
            client.put_record(
                StreamName=self._stream_name,
                Data=encrypted_data,
                PartitionKey=partition_key
            )
```

Consumer

The data consumer is an AWS Lambda function that is triggered by Kinesis events. It decrypts and deserializes each record, and writes the plaintext record to a DynamoDB table in the same region.

Like the producer code, the consumer code enables data key caching by using a caching cryptographic materials manager (caching CMM) in calls to the `decrypt` method.

```
"""
Copyright 2017 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.
"""
import base64
import json
import logging
import os

from aws_encryption_sdk import decrypt, KMSMasterKeyProvider,
 CachingCryptoMaterialsManager, LocalCryptoMaterialsCache
import boto3

_LOGGER = logging.getLogger(__name__)
_is_setup = False
CACHE_CAPACITY = 100
MAX_ENTRY_AGE_SECONDS = 600.0

def setup():
    """Sets up clients that should persist across Lambda invocations."""
    global materials_manager
    key_provider = KMSMasterKeyProvider()
    cache = LocalCryptoMaterialsCache(capacity=CACHE_CAPACITY)

    #  Because the cache is used only for decryption, the code doesn't set
    #   the max bytes or max message security thresholds that are are enforced
    #   only on on data keys used for encryption.
    materials_manager = CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=MAX_ENTRY_AGE_SECONDS
    )
    global table
    table_name = os.environ.get('TABLE_NAME')
    table = boto3.resource('dynamodb').Table(table_name)
    global _is_setup
    _is_setup = True


def lambda_handler(event, context):
    """Decrypts all incoming Kinesis records and writes records to DynamoDB."""
    _LOGGER.debug('New event:')
    _LOGGER.debug(event)
    if not _is_setup:
        setup()
```

```
    with table.batch_writer() as batch:
        for record in event.get('Records', []):
            # Record data base64-encoded by Kinesis
            ciphertext = base64.b64decode(record['kinesis']['data'])

            # Decrypt and unpack record
            plaintext, header = decrypt(
                source=ciphertext,
                materials_manager=materials_manager
            )
            item = json.loads(plaintext)

            # Verify the encryption context value
            stream_name = record['eventSourceARN'].split('/', 1)[1]
            if stream_name != header.encryption_context['stream']:
                raise ValueError('Wrong Encryption Context!')

            # Write record to DynamoDB
            batch.put_item(Item=item)
```

LocalCryptoMaterialsCache Example AWS CloudFormation Template

This AWS CloudFormation template sets up all the necessary AWS resources to replicate this example.

```
Parameters:
    SourceCodeBucket:
        Type: String
        Description: S3 bucket containing Lambda source code zip files
    PythonLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    PythonLambdaObjectVersionId:
        Type: String
        Description: S3 version id for S3 key containing Python Lambda source code zip file
    JavaLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    JavaLambdaObjectVersionId:
        Type: String
        Description: S3 version id for S3 key containing Python Lambda source code zip file
    KeyAliasSuffix:
        Type: String
        Description: 'Suffix to use for KMS CMK Alias (ie: alias/<KeyAliasSuffix>)'
    StreamName:
        Type: String
        Description: Name to use for Kinesis Stream
Resources:
    InputStream:
        Type: AWS::Kinesis::Stream
        Properties:
            Name: !Ref StreamName
            ShardCount: 2
    PythonLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
```

```
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    PythonLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: PythonLambdaAccess
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    PythonLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Python consumer
            Runtime: python2.7
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt PythonLambdaRole.Arn
            Handler: aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref PythonLambdaS3Key
                S3ObjectVersion: !Ref PythonLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref PythonLambdaOutputTable
    PythonLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
```

```
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:${AWS::AccountId}:stream/
${InputStream}
            FunctionName: !Ref PythonLambdaFunction
            StartingPosition: TRIM_HORIZON
    JavaLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    JavaLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: JavaLambdaAccess
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    JavaLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Java consumer
            Runtime: java8
            MemorySize: 512
```

```
            Timeout: 90
            Role: !GetAtt JavaLambdaRole.Arn
            Handler:
 com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref JavaLambdaS3Key
                S3ObjectVersion: !Ref JavaLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref JavaLambdaOutputTable
                    CMK_ARN: !GetAtt RegionKinesisCMK.Arn
    JavaLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:${AWS::AccountId}:stream/
${InputStream}
            FunctionName: !Ref JavaLambdaFunction
            StartingPosition: TRIM_HORIZON
    RegionKinesisCMK:
        Type: AWS::KMS::Key
        Properties:
            Description: Used to encrypt data passing through Kinesis Stream in this region
            Enabled: true
            KeyPolicy:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            AWS: !Sub arn:aws:iam::${AWS::AccountId}:root
                        Action:
                            # Data plane actions
                            - kms:Encrypt
                            - kms:GenerateDataKey
                            # Control plane actions
                            - kms:CreateAlias
                            - kms:DeleteAlias
                            - kms:DescribeKey
                            - kms:DisableKey
                            - kms:EnableKey
                            - kms:PutKeyPolicy
                            - kms:ScheduleKeyDeletion
                            - kms:UpdateAlias
                            - kms:UpdateKeyDescription
                        Resource: '*'
                    -
                        Effect: Allow
                        Principal:
                            AWS:
                                - !GetAtt PythonLambdaRole.Arn
                                - !GetAtt JavaLambdaRole.Arn
                        Action: kms:Decrypt
                        Resource: '*'
    RegionKinesisCMKAlias:
        Type: AWS::KMS::Alias
        Properties:
            AliasName: !Sub alias/${KeyAliasSuffix}
            TargetKeyId: !Ref RegionKinesisCMK
```

Frequently Asked Questions

- How is the AWS Encryption SDK different from the AWS SDKs? (p. 79)
- How is the AWS Encryption SDK different from the Amazon S3 encryption client? (p. 79)
- Which cryptographic algorithms are supported by the AWS Encryption SDK, and which one is the default? (p. 79)
- How is the initialization vector (IV) generated and where is it stored? (p. 80)
- How is each data key generated, encrypted, and decrypted? (p. 80)
- How do I keep track of the data keys that were used to encrypt my data? (p. 80)
- How does the AWS Encryption SDK store encrypted data keys with their encrypted data? (p. 80)
- How much overhead does the AWS Encryption SDK's message format add to my encrypted data? (p. 80)
- Can I use my own master key provider? (p. 80)
- Can I encrypt data under more than one master key? (p. 80)
- Which data types can I encrypt with the AWS Encryption SDK? (p. 81)
- How does the AWS Encryption SDK encrypt and decrypt input/output (I/O) streams? (p. 81)

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

The AWS SDKs provide libraries for interacting with Amazon Web Services (AWS). They integrate with AWS Key Management Service (AWS KMS) to generate, encrypt, and decrypt data keys. However, in most cases you can't use them to directly encrypt or decrypt raw data.

The AWS Encryption SDK provides an encryption library that optionally integrates with AWS KMS as a master key provider. The AWS Encryption SDK builds on the AWS SDKs to do the following things:

- Generate, encrypt, and decrypt data keys
- Use those data keys to encrypt and decrypt your raw data
- Store the encrypted data keys with the corresponding encrypted data in a single object

You can also use the AWS Encryption SDK with no AWS integration by defining a custom master key provider.

How is the AWS Encryption SDK different from the Amazon S3 encryption client?

The Amazon S3 encryption client in the AWS SDK for Java, AWS SDK for Ruby, and AWS SDK for .NET provides encryption and decryption for data that you store in Amazon Simple Storage Service (Amazon S3). These clients are tightly coupled to Amazon S3 and are intended for use only with data stored there.

The AWS Encryption SDK provides encryption and decryption for data that you can store anywhere. The AWS Encryption SDK and the Amazon S3 encryption client are not compatible because they produce ciphertexts with different data formats.

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

The AWS Encryption SDK uses the Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM), known as AES-GCM. The SDK supports 256-bit, 192-bit, and 128-bit encryption keys. In all cases, the length of the initialization vector (IV) is 12 bytes; the length of the authentication tag is 16 bytes. By default, the SDK uses the data key as an input to the HMAC-based extract-and-expand key derivation function (HKDF) to derive the AES-GCM encryption key, and also adds an Elliptic Curve Digital Signature Algorithm (ECDSA) signature.

For information about choosing which algorithm to use, see Supported Algorithm Suites (p. 10).

For implementation details about the supported algorithms, see Algorithms Reference (p. 96).

How is the initialization vector (IV) generated and where is it stored?

In previous releases, the AWS Encryption SDK randomly generated a unique IV value for each encryption operation. The SDK now uses a deterministic method to construct a different IV value for each frame so that every IV is unique within its message. The SDK stores the IV in the encrypted message that it returns. For more information, see AWS Encryption SDK Message Format Reference (p. 82).

How is each data key generated, encrypted, and decrypted?

The method depends on the master key provider and the implementation of its master keys. When AWS KMS is the master key provider, the SDK uses the AWS KMS GenerateDataKey API operation to generate each data key in both plaintext and encrypted forms. It uses the Decrypt operation to decrypt the data key. AWS KMS encrypts and decrypts the data key by using the customer master key (CMK) that you specified when configuring the master key provider.

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

The AWS Encryption SDK does this for you. When you encrypt data, the SDK encrypts the data key and stores the encrypted key along with the encrypted data in the encrypted message (p. 8) that it returns. When you decrypt data, the AWS Encryption SDK extracts the encrypted data key from the encrypted message, decrypts it, and then uses it to decrypt the data.

How does the AWS Encryption SDK store encrypted data keys with their encrypted data?

The encryption operations in the AWS Encryption SDK return an encrypted message (p. 8), a single data structure that contains the encrypted data and its encrypted data keys. 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 known as a *footer*. The message header contains the encrypted data keys 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 AWS Encryption SDK Message Format Reference (p. 82).

How much overhead does the AWS Encryption SDK's message format add to my encrypted data?

The amount of overhead added by the AWS Encryption SDK depends on several factors, including the following:

- The size of the plaintext data
- Which of the supported algorithms is used
- Whether additional authenticated data (AAD) is provided, and the length of that AAD
- The number and type of master key providers
- The frame size (when framed data (p. 88) is used)

When you use the AWS Encryption SDK with its default configuration, with one CMK in AWS KMS as the master key, with no AAD, and encrypt nonframed data, the overhead is approximately 600 bytes. In general, you can reasonably assume that the AWS Encryption SDK adds overhead of 1 KB or less, not including the provided AAD. For more information, see AWS Encryption SDK Message Format Reference (p. 82).

Can I use my own master key provider?

Yes. The implementation details vary depending on which of the supported programming languages (p. 12) you use. However, all supported languages allow you to define custom cryptographic materials managers (CMMs) (p. 7), master key providers, and master keys.

Can I encrypt data under more than one master key?

Yes. You can encrypt the data key with additional master keys to add redundancy in case a master key is in a different region or is unavailable for decryption.

To encrypt data under multiple master keys, create a master key provider with multiple master keys. You can see examples of this pattern in the example code for Java (p. 17) and Python (p. 24).

When you encrypt data by using a master key provider that returns multiple master keys, the AWS Encryption SDK encrypts the data that you pass to the encryption methods with a data key and encrypts that data key with the same master key. Then, it encrypts the data with the other master keys that the master key provider returned. The resulting message includes the encrypted data and one encrypted data key for each master key. The resulting message can be decrypted by using any one of the master keys used in the encryption operation.

Which data types can I encrypt with the AWS Encryption SDK?

The AWS Encryption SDK can encrypt raw bytes (byte arrays), I/O streams (byte streams), and strings. We provide example code for each of the supported programming languages (p. 12).

How does the AWS Encryption SDK encrypt and decrypt input/output (I/O) 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 it can read ciphertext from an underlying stream and decrypt it before returning the result. We provide example code for encrypting and decrypting streams for each of the supported programming languages (p. 12).

AWS Encryption SDK Reference

The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.

To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 12).

The AWS Encryption SDK uses the supported algorithms (p. 10) to return a single data structure or *message* that contains encrypted data and the corresponding encrypted data keys. The following topics explain the algorithms and the data structure. Use this information to build libraries that can read and write ciphertexts that are compatible with this SDK.

Topics

- AWS Encryption SDK Message Format Reference (p. 82)
- Body Additional Authenticated Data (AAD) Reference for the AWS Encryption SDK (p. 90)
- AWS Encryption SDK Message Format Examples (p. 91)
- AWS Encryption SDK Algorithms Reference (p. 96)
- AWS Encryption SDK Initialization Vector Reference (p. 98)

AWS Encryption SDK Message Format Reference

The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.

To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 12).

The encryption operations in the AWS Encryption SDK return a single data structure or *message* that contains the encrypted data (ciphertext) and all encrypted data keys. 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, 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 there is one).

Topics

- Header Structure (p. 83)
- Body Structure (p. 87)
- Footer Structure (p. 89)

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 (bytes)
Version (p. 83)	1
Type (p. 83)	1
Algorithm ID (p. 83)	2
Message ID (p. 84)	16
AAD Length (p. 84)	2 When the encryption context (p. 8) is empty, the AAD Length is 0.
AAD (p. 84)	Variable. Equal to the value specified in the previous 2 bytes (AAD Length). When the encryption context (p. 8) is empty, there is no AAD field in the header.
Encrypted Data Key Count (p. 85)	2
Encrypted Data Key(s) (p. 85)	Variable. Determined by the number of encrypted data keys and the length of each.
Content Type (p. 86)	1
Reserved (p. 86)	4
IV Length (p. 86)	1
Frame Length (p. 86)	4
Header Authentication (p. 86)	Variable. Determined by the algorithm (p. 96) that generated the message.

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. The only supported type 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. For more information about the algorithms, see AWS Encryption SDK Algorithms Reference (p. 96).

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 a data key with multiple encrypted messages.
- 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.

When the encryption context (p. 8) is empty, the AAD Length is 0.

AAD

The additional authenticated data. The AAD is an encoding of the encryption context (p. 8), 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 encryption context is empty, there is no AAD field in the header.

When the algorithms with signing (p. 96) are used, the encryption context must contain the key-value pair `{'aws-crypto-public-key', Qtxt}`. Qtxt represents the elliptic curve point Q compressed according to SEC 1 version 2.0 and then base64-encoded. The encryption context can contain additional values, but the maximum length of the constructed AAD is 2^16 - 1 bytes.

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. The bytes are appended in the order shown.

AAD Structure

Field	Length (bytes)
Key-Value Pair Count (p. 84)	2
Key Length (p. 84)	2
Key (p. 85)	Variable. Equal to the value specified in the previous 2 bytes (Key Length).
Value Length (p. 85)	2
Value (p. 85)	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. The maximum number of key-value pairs in the AAD is 2^16 - 1.

When there is no encryption context or the encryption context is empty, this field is not present in the AAD structure.

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 (p. 85)	2
Key Provider ID (p. 85)	Variable. Equal to the value specified in the previous 2 bytes (Key Provider ID Length).
Key Provider Information Length (p. 85)	2
Key Provider Information (p. 85)	Variable. Equal to the value specified in the previous 2 bytes (Key Provider Information Length).
Encrypted Data Key Length (p. 86)	2
Encrypted Data Key (p. 86)	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, the following are true:

- This value contains the Amazon Resource Name (ARN) of the AWS KMS customer master key (CMK).
- This value is always the full CMK ARN, regardless of which key identifier (key ID, alias, etc.) was specified when calling the master key provider.

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 the data encryption key encrypted by the key provider.

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

Framed content is broken into equal-length parts; each part is encrypted separately. 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. This value is determined by the IV bytes value of the algorithm (p. 96) that generated the message.

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.

Header Authentication

The header authentication is determined by the algorithm (p. 96) that generated the message. The header authentication is calculated over the entire header. It consists of an IV and an authentication tag. The bytes are appended in the order shown.

Header Authentication Structure

Field	Length, in bytes
IV (p. 86)	Variable. Determined by the IV bytes value of the algorithm (p. 96) that generated the message.
Authentication Tag (p. 87)	Variable. Determined by the authentication tag bytes value of the algorithm (p. 96) that generated the message.

IV

The initialization vector (IV) used to calculate the header authentication tag.

Authentication Tag

The authentication value for the header. It is used to authenticate the entire contents of the header.

Body Structure

The message body contains the encrypted data, called the *ciphertext*. 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 (p. 87)
- Framed Data (p. 88)

Non-Framed Data

Non-framed data is encrypted in a single blob with a unique IV and body AAD (p. 90). 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 (p. 87)	Variable. Equal to the value specified in the IV Length (p. 86) byte of the header.
Encrypted Content Length (p. 87)	8
Encrypted Content (p. 87)	Variable. Equal to the value specified in the previous 8 bytes (Encrypted Content Length).
Authentication Tag (p. 87)	Variable. Determined by the algorithm implementation (p. 96) used.

IV

The initialization vector (IV) to use with the encryption algorithm (p. 96).

Encrypted Content Length

The length of the encrypted content, or *ciphertext*. It is an 8-byte value interpreted as a 64-bit unsigned integer that specifies the number of bytes that contain the encrypted content.

Technically, the maximum allowed value is 2^63 - 1, or 8 exbibytes (8 EiB). However, in practice the maximum value is 2^36 - 32, or 64 gibibytes (64 GiB), due to restrictions imposed by the implemented algorithms (p. 96).

Note
The Java implementation of this SDK further restricts this value to 2^31 - 1, or 2 gibibytes (2 GiB), due to restrictions in the language.

Encrypted Content

The encrypted content (ciphertext) as returned by the encryption algorithm (p. 96).

Authentication Tag

The authentication value for the body. It is used to authenticate the message body.

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 body AAD (p. 90).

There are two kinds of frames: regular and final. A final frame is always used. When the length of the data is an exact multiple of the frame length, the final frame contains no data—that is, it has a content length of 0. When the length of the data is less than the frame length, only a final frame is written.

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, in bytes
Sequence Number (p. 88)	4
IV (p. 88)	Variable. Equal to the value specified in the IV Length (p. 86) byte of the header.
Encrypted Content (p. 88)	Variable. Equal to the value specified in the Frame Length (p. 86) of the header.
Authentication Tag (p. 88)	Variable. Determined by the algorithm used, as specified in the Algorithm ID (p. 83) of the header.

Sequence Number

The frame sequence number. It is an incremental counter number for the frame. It is a 4-byte value interpreted as a 32-bit unsigned integer.

Framed data must start at sequence number 1. Subsequent 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 (IV) for the frame. The SDK uses a deterministic method to construct a different IV for each frame in the message. Its length is specified by the algorithm suite (p. 96) used.

Encrypted Content

The encrypted content (ciphertext) for the frame, as returned by the encryption algorithm (p. 96).

Authentication Tag

The authentication value for the frame. It is used to authenticate the entire frame.

Framed Body Structure, Final Frame

Field	Length, in bytes
Sequence Number End (p. 89)	4
Sequence Number (p. 89)	4

Field	Length, in bytes
IV (p. 89)	Variable. Equal to the value specified in the IV Length (p. 86) byte of the header.
Encrypted Content Length (p. 89)	4
Encrypted Content (p. 89)	Variable. Equal to the value specified in the previous 4 bytes (Encrypted Content Length).
Authentication Tag (p. 89)	Variable. Determined by the algorithm used, as specified in the Algorithm ID (p. 83) 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. It is a 4-byte value interpreted as a 32-bit unsigned integer.

Framed data must start at sequence number 1. Subsequent 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 (IV) for the frame. The SDK uses a deterministic method to construct a different IV for each frame in the message. The length of the IV length is specified by the algorithm suite (p. 96).

Encrypted Content Length

The length of the encrypted content. 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 (ciphertext) for the frame, as returned by the encryption algorithm (p. 96).

Authentication Tag

The authentication value for the frame. It is used to authenticate the entire frame.

Footer Structure

When the algorithms with signing (p. 96) are used, the message format contains a footer. The message footer contains a signature calculated over the message header and 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 (p. 90)	2
Signature (p. 90)	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.

Body Additional Authenticated Data (AAD) Reference for the AWS Encryption SDK

The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.

To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 12).

You must provide additional authenticated data (AAD) to the AES-GCM algorithm (p. 96) for each cryptographic operation. This is true for both framed and non-framed body data (p. 87). For more information about AAD, see the definition section in the Galois/Counter Mode of Operation (GCM) specification.

The following table describes the fields that form the body AAD. The bytes are appended in the order shown.

Body AAD Structure

Field	Length, in bytes
Message ID (p. 90)	16
Body AAD Content (p. 90)	Variable. See **Body AAD Content** in the following list.
Sequence Number (p. 91)	4
Content Length (p. 91)	8

Message ID

The same Message ID (p. 84) value set in the message header.

Body AAD Content

A UTF-8 encoded value determined by the type of body data used.

For non-framed data (p. 87), use the value `AWSKMSEncryptionClient Single Block`.

For regular frames in framed data (p. 88), use the value `AWSKMSEncryptionClient Frame`.

For the final frame in framed data (p. 88), use the value `AWSKMSEncryptionClient Final Frame`.

Sequence Number

A 4-byte value interpreted as a 32-bit unsigned integer.

For framed data (p. 88), this is the frame sequence number.

For non-framed data (p. 87), use the value 1, encoded as the 4 bytes `00 00 00 01` in hexadecimal notation.

Content Length

The length, in bytes, of the plaintext data provided to the algorithm for encryption. It is an 8-byte value interpreted as a 64-bit unsigned integer.

AWS Encryption SDK Message Format Examples

The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.

To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 12).

The following topics show examples of the AWS Encryption SDK message format. Each example shows the raw bytes, in hexadecimal notation, followed by a description of what those bytes represent.

Topics

- Non-Framed Data (p. 91)
- Framed Data (p. 93)

Non-Framed Data

The following example shows the message format for non-framed data.

```
+--------+
| Header |
+--------+
01                                          Version (1.0)
80                                          Type (128, customer authenticated encrypted
 data)
0378                                        Algorithm ID (see Algorithms Reference)
B8929B01 753D4A45 C0217F39 404F70FF         Message ID (random 128-bit value)
008E                                        AAD Length (142)
0004                                        AAD Key-Value Pair Count (4)
0005                                        AAD Key-Value Pair 1, Key Length (5)
30746869 73                                 AAD Key-Value Pair 1, Key ("0This")
0002                                        AAD Key-Value Pair 1, Value Length (2)
6973                                        AAD Key-Value Pair 1, Value ("is")
0003                                        AAD Key-Value Pair 2, Key Length (3)
31616E                                      AAD Key-Value Pair 2, Key ("1an")
000A                                        AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                      AAD Key-Value Pair 2, Value ("encryption")
0008                                        AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                           AAD Key-Value Pair 3, Key ("2context")
0007                                        AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                             AAD Key-Value Pair 3, Value ("example")
0015                                        AAD Key-Value Pair 4, Key Length (21)
```

```
6177732D 63727970 746F2D70 75626C69        AAD Key-Value Pair 4, Key ("aws-crypto-public-
key")
632D6B65 79
0044                                       AAD Key-Value Pair 4, Value Length (68)
41734738 67473949 6E4C5075 3136594B        AAD Key-Value Pair 4, Value
 ("AsG8gG9InLPu16YKlqXTOD+nykG8YqHAhqecj8aXfD2e5B4gtVE73dZkyClA+rAMOQ==")
6C715854 4F442B6E 796B4738 59714841
68716563 6A386158 66443265 35423467
74564537 33645A6B 79436C41 2B72414D
4F513D3D
0002                                       Encrypted Data Key Count (2)
0007                                       Encrypted Data Key 1, Key Provider ID Length (7)
6177732D 6B6D73                            Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                       Encrypted Data Key 1, Key Provider Information
 Length (75)
61726E3A 6177733A 6B6D733A 75732D77        Encrypted Data Key 1, Key Provider Information
 ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                       Encrypted Data Key 1, Encrypted Data Key Length
 (167)
01010200 7857A1C1 F7370545 4ECA7C83        Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C28 4116449A
0F2A0383 659EF802 0110803B B23A8133
3A33605C 48840656 C38BCB1F 9CCE7369
E9A33EBE 33F46461 0591FECA 947262F3
418E1151 21311A75 E575ECC5 61A286E0
3E2DEBD5 CB005D
0007                                       Encrypted Data Key 2, Key Provider ID Length (7)
6177732D 6B6D73                            Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                       Encrypted Data Key 2, Key Provider Information
 Length (78)
61726E3A 6177733A 6B6D733A 63612D63        Encrypted Data Key 2, Key Provider Information
 ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                       Encrypted Data Key 2, Encrypted Data Key Length
 (167)
01010200 78FAFFFB D6DE06AF AC72F79B        Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040CB2 A820D0CC
76616EF2 A6B30D02 0110803B 8073D0F1
FDD01BD9 B0979082 099FDBFC F7B13548
3CC686D7 F3CF7C7A CCC52639 122A1495
71F18A46 80E2C43F A34C0E58 11D05114
2A363C2A E11397
01                                         Content Type (1, non-framed data)
00000000                                   Reserved
0C                                         IV Length (12)
00000000                                   Frame Length (0, non-framed data)
734C1BBE 032F7025 84CDA9D0                 IV
2C82BB23 4CBF4AAB 8F5C6002 622E886C        Authentication Tag
+------+
```

```
| Body |
+-------+
D39DD3E5 915E0201 77A4AB11                IV
00000000 0000028E                         Encrypted Content Length (654)
E8B6F955 B5F22FE4 FD890224 4E1D5155       Encrypted Content
5871BA4C 93F78436 1085E4F8 D61ECE28
59455BD8 D76479DF C28D2E0B BDB3D5D3
E4159DFE C8A944B6 685643FC EA24122B
6766ECD5 E3F54653 DF205D30 0081D2D8
55FCDA5B 9F5318BC F4265B06 2FE7C741
C7D75BCC 10F05EA5 0E2F2F40 47A60344
ECE10AA7 559AF633 9DE2C21B 12AC8087
95FE9C58 C65329D1 377C4CD7 EA103EC1
31E4F48A 9B1CC047 EE5A0719 704211E5
B48A2068 8060DF60 B492A737 21B0DB21
C9B21A10 371E6179 78FAFB0B BAAEC3F4
9D86E334 701E1442 EA5DA288 64485077
54C0C231 AD43571A B9071925 609A4E59
B8178484 7EB73A4F AAE46B26 F5B374B8
12B0000C 8429F504 936B2492 AAF47E94
A5BA804F 7F190927 5D2DF651 B59D4C2F
A15D0551 DAEBA4AF 2060D0D5 CB1DA4E6
5E2034DB 4D19E7CD EEA6CF7E 549C86AC
46B2C979 AB84EE12 202FD6DF E7E3C09F
C2394012 AF20A97E 369BCBDA 62459D3E
C6FFB914 FEFD4DE5 88F5AFE1 98488557
1BABBAE4 BE55325E 4FB7E602 C1C04BEE
F3CB6B86 71666C06 6BF74E1B 0F881F31
B731839B CF711F6A 84CA95F5 958D3B44
E3862DF6 338E02B5 C345CFF8 A31D54F3
6920AA76 0BF8E903 552C5A04 917CCD11
D4E5DF5C 491EE86B 20C33FE1 5D21F0AD
6932E67C C64B3A26 B8988B25 CFA33E2B
63490741 3AB79D60 D8AEFBE9 2F48E25A
978A019C FE49EE0A 0E96BF0D D6074DDB
66DFF333 0E10226F 0A1B219C BE54E4C2
2C15100C 6A2AA3F1 88251874 FDC94F6B
9247EF61 3E7B7E0D 29F3AD89 FA14A29C
76E08E9B 9ADCDF8C C886D4FD A69F6CB4
E24FDE26 3044C856 BF08F051 1ADAD329
C4A46A1E B5AB72FE 096041F1 F3F3571B
2EAFD9CB B9EB8B83 AE05885A 8F2D2793
1E3305D9 0C9E2294 E8AD7E3B 8E4DEC96
6276C5F1 A3B7E51E 422D365D E4C0259C
50715406 822D1682 80B0F2E5 5C94
65B2E942 24BEEA6E A513F918 CCEC1DE3      Authentication Tag
+--------+
| Footer |
+--------+
0067                                     Signature Length (103)
30650230 7229DDF5 B86A5B64 54E4D627      Signature
CBE194F1 1CC0F8CF D27B7F8B F50658C0
BE84B355 3CED1721 A0BE2A1B 8E3F449E
1BEB8281 023100B2 0CB323EF 58A4ACE3
1559963B 889F72C3 B15D1700 5FB26E61
331F3614 BC407CEE B86A66FA CBF74D9E
34CB7E4B 363A38
```

Framed Data

The following example shows the message format for framed data.

```
+--------+
| Header |
```

```
+--------+
01                                          Version (1.0)
80                                          Type (128, customer authenticated encrypted
 data)
0378                                        Algorithm ID (see Algorithms Reference)
6E7C0FBD 4DF4A999 717C22A2 DDFE1A27         Message ID (random 128-bit value)
008E                                        AAD Length (142)
0004                                        AAD Key-Value Pair Count (4)
0005                                        AAD Key-Value Pair 1, Key Length (5)
30746869 73                                 AAD Key-Value Pair 1, Key ("0This")
0002                                        AAD Key-Value Pair 1, Value Length (2)
6973                                        AAD Key-Value Pair 1, Value ("is")
0003                                        AAD Key-Value Pair 2, Key Length (3)
31616E                                      AAD Key-Value Pair 2, Key ("1an")
000A                                        AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                      AAD Key-Value Pair 2, Value ("encryption")
0008                                        AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                           AAD Key-Value Pair 3, Key ("2context")
0007                                        AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                             AAD Key-Value Pair 3, Value ("example")
0015                                        AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69         AAD Key-Value Pair 4, Key ("aws-crypto-public-
key")
632D6B65 79
0044                                        AAD Key-Value Pair 4, Value Length (68)
416A4173 7569326F 7430364C 4B77715A         AAD Key-Value Pair 4, Value
 ("AjAsui2ot06LKwqZXDJnU/Aqc2vD+0OkpOZ1cc8Tg2qd7rs5aLTg7lvfUEW/86+/5w==")
58444A6E 552F4171 63327644 2B304F6B
704F5A31 63633854 67327164 37727335
614C5467 376C7666 5545572F 38362B2F
35773D3D
0002                                        EncryptedDataKeyCount (2)
0007                                        Encrypted Data Key 1, Key Provider ID Length (7)
6177732D 6B6D73                             Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                        Encrypted Data Key 1, Key Provider Information
 Length (75)
61726E3A 6177733A 6B6D733A 75732D77         Encrypted Data Key 1, Key Provider Information
 ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                        Encrypted Data Key 1, Encrypted Data Key Length
 (167)
01010200 7857A1C1 F7370545 4ECA7C83         Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C3F F02C897B
7A12EB19 8BF2D802 0110803B 24003D1F
A5474FBC 392360B5 CB9997E0 6A17DE4C
A6BD7332 6BF86DAB 60D8CCB8 8295DBE9
4707E356 ADA3735A 7C52D778 B3135A47
9F224BF9 E67E87
0007                                        Encrypted Data Key 2, Key Provider ID Length (7)
6177732D 6B6D73                             Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                        Encrypted Data Key 2, Key Provider Information
 Length (78)
61726E3A 6177733A 6B6D733A 63612D63         Encrypted Data Key 2, Key Provider Information
 ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
```

```
372D6265 33343335 62343233 6666
00A7                                    Encrypted Data Key 2, Encrypted Data Key Length
 (167)
01010200 78FAFFFB D6DE06AF AC72F79B     Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C36 CD985E12
D218B674 5BBC6102 0110803B 0320E3CD
E470AA27 DEAB660B 3E0CE8E0 8B1A89E4
57DCC69B AAB1294F 21202C01 9A50D323
72EBAAFD E24E3ED8 7168E0FA DB40508F
556FBD58 9E621C
02                                      Content Type (2, framed data)
00000000                                Reserved
0C                                      IV Length (12)
00000100                                Frame Length (256)
4ECBD5C0 9899CA65 923D2347              IV
0B896144 0CA27950 CA571201 4DA58029     Authentication Tag
+------+
| Body |
+------+
00000001                                Frame 1, Sequence Number (1)
6BD3FE9C ADBCB213 5B89E8F1              Frame 1, IV
1F6471E0 A51AF310 10FA9EF6 F0C76EDF     Frame 1, Encrypted Content
F5AFA33C 7D2E8C6C 9C5D5175 A212AF8E
FBD9A0C3 C6E3FB59 C125DBF2 89AC7939
BDEE43A8 0F00F49E ACBBD8B2 1C785089
A90DB923 699A1495 C3B31B50 0A48A830
201E3AD9 1EA6DA14 7F6496DB 6BC104A4
DEB7F372 375ECB28 9BF84B6D 2863889F
CB80A167 9C361C4B 5EC07438 7A4822B4
A7D9D2CC 5150D414 AF75F509 FCE118BD
6D1E798B AEBA4CDB AD009E5F 1A571B77
0041BC78 3E5F2F41 8AF157FD 461E959A
BB732F27 D83DC36D CC9EBC05 00D87803
57F2BB80 066971C2 DEEA062F 4F36255D
E866C042 E1382369 12E9926B BA40E2FC
A820055F FB47E428 41876F14 3B6261D9
5262DB34 59F5D37E 76E46522 E8213640
04EE3CC5 379732B5 F56751FA 8E5F26AD     Frame 1, Authentication Tag
00000002                                Frame 2, Sequence Number (2)
F1140984 FF25F943 959BE514              Frame 2, IV
216C7C6A 2234F395 F0D2D9B9 304670BF     Frame 2, Encrypted Content
A1042608 8A8BCB3F B58CF384 D72EC004
A41455B4 9A78BAC9 36E54E68 2709B7BD
A884C1E1 705FF696 E540D297 446A8285
23DFEE28 E74B225A 732F2C0C 27C6BDA2
7597C901 65EF3502 546575D4 6D5EBF22
1FF787AB 2E38FD77 125D129C 43D44B96
778D7CEE 3C36625F FF3A985C 76F7D320
ED70B1F3 79729B47 E7D9B5FC 02FCE9F5
C8760D55 7779520A 81D54F9B EC45219D
95941F7E 5CBAEAC8 CEC13B62 1464757D
AC65B6EF 08262D74 44670624 A3657F7F
2A57F1FD E7060503 AC37E197 2F297A84
DF1172C2 FA63CF54 E6E2B9B6 A86F582B
3B16F868 1BBC5E4D 0B6919B3 08D5ABCF
FECDC4A4 8577F08B 99D766A1 E5545670
A61F0A3B A3E45A84 4D151493 63ECA38F     Frame 2, Authentication Tag
FFFFFFFF                                Final Frame, Sequence Number End
00000003                                Final Frame, Sequence Number (3)
35F74F11 25410F01 DD9E04BF              Final Frame, IV
0000008E                                Final Frame, Encrypted Content Length (142)
F7A53D37 2F467237 6FBD0B57 D1DFE830     Final Frame, Encrypted Content
```

```
B965AD1F A910AA5F 5EFFFFF4 BC7D431C
BA9FA7C4 B25AF82E 64A04E3A A0915526
88859500 7096FABB 3ACAD32A 75CFED0C
4A4E52A3 8E41484D 270B7A0F ED61810C
3A043180 DF25E5C5 3676E449 0986557F
C051AD55 A437F6BC 139E9E55 6199FD60
6ADC017D BA41CDA4 C9F17A83 3823F9EC
B66B6A5A 80FDB433 8A48D6A4 21CB
811234FD 8D589683 51F6F39A 040B3E3B        Final Frame, Authentication Tag
+--------+
| Footer |
+--------+
0066                                       Signature Length (102)
30640230 085C1D3C 63424E15 B2244448        Signature
639AED00 F7624854 F8CF2203 D7198A28
758B309F 5EFD9D5D 2E07AD0B 467B8317
5208B133 02301DF7 2DFC877A 66838028
3C6A7D5E 4F8B894E 83D98E7C E350F424
7E06808D 0FE79002 E24422B9 98A0D130
A13762FF 844D
```

AWS Encryption SDK Algorithms Reference

> The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.
>
> To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 12).

To build your own library that can read and write ciphertexts that are compatible with the AWS Encryption SDK, you should understand how the SDK implements the supported algorithms to encrypt raw data. The SDK supports nine algorithm suites. An implementation specifies the encryption algorithm and mode, encryption key length, key derivation algorithm (if one applies), and signature algorithm (if one applies). The following table contains an overview of each implementation. By default, the SDK uses the first implementation in the following table. The list that follows the table provides more information.

AWS Encryption SDK Algorithm Suites

Algorithm ID (in 2-byte hex)	Algorithm Name	Data Key Length (in bits)	Algorithm Mode	IV Length (in bytes)	Authenticat Tag Length (in bytes)	Key Derivation Algorithm	Signature Algorithm
`03 78`	AES	256	GCM	12	16	HKDF with SHA-384	ECDSA with P-384 and SHA-384
`03 46`	AES	192	GCM	12	16	HKDF with SHA-384	ECDSA with P-384 and SHA-384
`02 14`	AES	128	GCM	12	16	HKDF with SHA-256	ECDSA with

Algorithm ID (in 2-byte hex)	Algorithm Name	Data Key Length (in bits)	Algorithm Mode	IV Length (in bytes)	Authenticat Tag Length (in bytes)	Key Derivation Algorithm	Signature Algorithm
							P-256 and SHA-256
`01 78`	AES	256	GCM	12	16	HKDF with SHA-256	Not applicable
`01 46`	AES	192	GCM	12	16	HKDF with SHA-256	Not applicable
`01 14`	AES	128	GCM	12	16	HKDF with SHA-256	Not applicable
`00 78`	AES	256	GCM	12	16	Not applicable	Not applicable
`00 46`	AES	192	GCM	12	16	Not applicable	Not applicable
`00 14`	AES	128	GCM	12	16	Not applicable	Not applicable

Algorithm ID

A 2-byte value that uniquely identifies an algorithm's implementation. This value is stored in the ciphertext's message header (p. 83).

Algorithm Name

The encryption algorithm used. For all algorithm suites, the SDK uses the Advanced Encryption Standard (AES) encryption algorithm.

Data Key Length

The length of the data key. The SDK supports 256-bit, 192-bit, and 128-bit keys. The data key is generated by a master key. For some implementations, this data key is used as input to an HMAC-based extract-and-expand key derivation function (HKDF). The output of the HKDF is used as the data encryption key in the encryption algorithm. For more information, see **Key Derivation Algorithm** in this list.

Algorithm Mode

The mode used with the encryption algorithm. For all algorithm suites, the SDK uses Galois/Counter Mode (GCM).

IV Length

The length of the initialization vector (IV) used with AES-GCM. For more information about the IVs that the AWS Encryption SDK uses, see the Initialization Vector Reference (p. 98).

Authentication Tag Length

The length of the authentication tag used with AES-GCM.

Key Derivation Algorithm

The HMAC-based extract-and-expand key derivation function (HKDF) used to derive the data encryption key. The SDK uses the HKDF defined in RFC 5869, with the following specifics:

- The hash function used is either SHA-384 or SHA-256, as specified by the algorithm ID.

- For the extract step:
  - No salt is used. Per the RFC, this means that the salt is set to a string of zeros. The string length is equal to the length of the hash function output; that is, 48 bytes for SHA-384 and 32 bytes for SHA-256.
  - The input keying material is the data key received from the master key provider.
- For the expand step:
  - The input pseudorandom key is the output from the extract step.
  - The input info is a concatenation of the algorithm ID followed by the message ID.
  - The length of the output keying material is the **Data Key Length** described previously. This output is used as the data encryption key in the encryption algorithm.

Signature Algorithm

The signature algorithm used to generate a signature over the ciphertext header and body. The SDK uses the Elliptic Curve Digital Signature Algorithm (ECDSA) with the following specifics:

- The elliptic curve used is either the P-384 or P-256 curve, as specified by the algorithm ID. These curves are defined in Digital Signature Standard (DSS) (FIPS PUB 186-4).
- The hash function used is SHA-384 (with the P-384 curve) or SHA-256 (with the P-256 curve).

AWS Encryption SDK Initialization Vector Reference

> The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.
>
> To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 12).

The AWS Encryption SDK supplies the initialization vectors (IVs) that are required by all supported algorithm suites (p. 96). The SDK uses frame sequence numbers to construct an IV so that no two frames in the same message can have the same IV.

Each 96-bit (12-byte) IV is constructed from two big-endian byte arrays concatenated in the following order:

- 64 bits: 0 (reserved for future use)
- 32 bits: Frame sequence number. For the header authentication tag, this value is all zeroes.

Before the introduction of data key caching (p. 55), the AWS Encryption SDK always used a new data key to encrypt each message, and it generated all IVs randomly. Randomly generated IVs were cryptographically safe because data keys were never reused. When the SDK introduced data key caching, which intentionally reuses data keys, we changed the way the SDK generates IVs.

Using deterministic IVs that cannot repeat within a message significantly increases the number of invocations that can safely be executed under a single data key. In addition, data keys that are cached always use an algorithm suite with a key derivation function. Using a deterministic IV with a pseudo-random key derivation function to derive encryption keys from a data key allows the AWS Encryption SDK to encrypt 2^32 messages without exceeding cryptographic bounds.

Document History for the AWS Encryption SDK Developer Guide

This topic describes significant updates to the *AWS Encryption SDK Developer Guide*.

Topics

- Recent Updates (p. 99)
- Earlier Updates (p. 99)

Recent Updates

The following table describes significant changes to this documentation since November 2017. In addition to major changes listed here, we also update the documentation frequently to improve the descriptions and examples, and to address the feedback that you send to us. To be notified about significant changes, use the link in the upper right corner to subscribe to the RSS feed.

Latest documentation update: June 8, 2018 (minor)

update-history-change	update-history-description	update-history-date
New release	Added documentation of the new command line interface for the AWS Encryption SDK.	November 20, 2017

Earlier Updates

The following table describes significant changes to the *AWS Encryption SDK Developer Guide* before November 2017.

Change	Description	Date
New release	Added the *Data Key Caching* (p. 55) chapter for the new feature. Added the the section called "Initialization Vector Reference" (p. 98) topic that explains that the SDK changed from generating random IVs to constructing deterministic IVs. Added the the section called "Concepts" (p. 5) topic to explain concepts, including the new cryptographic materials manager.	July 31, 2017

Change	Description	Date
Update	Expanded the Message Format Reference (p. 82) documentation into a new AWS Encryption SDK Reference (p. 82) section. Added a section about the AWS Encryption SDK's Supported Algorithm Suites (p. 10).	March 21, 2017
New release	The AWS Encryption SDK now supports the Python (p. 20) programming language, in addition to Java (p. 12).	March 21, 2017
Initial release	Initial release of the AWS Encryption SDK and this documentation.	March 22, 2016