AWS Encryption SDK: Developer Guide
Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.
# Table of Contents

What is the AWS Encryption SDK? ........................................................................................................... 1
   Compatibility with encryption libraries and services ................................................................. 2
   Learning more .............................................................................................................................. 2
   Sending feedback .......................................................................................................................... 3
   How the SDK works ..................................................................................................................... 3
      Symmetric key encryption ......................................................................................................... 3
      Envelope encryption .................................................................................................................. 4
   AWS Encryption SDK encryption workflows ............................................................................ 5

Concepts ................................................................................................................................................. 6
   Data keys ...................................................................................................................................... 7
   Master key .................................................................................................................................... 7
   Cryptographic materials manager .............................................................................................. 7
   Master key provider (Java and Python) ...................................................................................... 8
   Keyring (C and JavaScript) ......................................................................................................... 8
   Algorithm suite ........................................................................................................................... 8
   Encryption context ..................................................................................................................... 9
   Encrypted message ................................................................................................................... 10

Getting started ......................................................................................................................................... 11
   Using keyrings ............................................................................................................................. 12
      How keyrings work .................................................................................................................. 12
      Keyring compatibility ............................................................................................................ 14
   AWS KMS keyrings ................................................................................................................... 14
      Required permissions for AWS KMS keyrings ...................................................................... 15
      Identifying CMKs in an AWS KMS keyring .......................................................................... 15
      Encrypting with an AWS KMS keyring .................................................................................. 15
      Decrypting with an AWS KMS keyring .................................................................................. 17
      Using an AWS KMS discovery keyring .................................................................................. 18
      Using an AWS KMS regional discovery keyring ................................................................... 20
   Raw AES keyrings ....................................................................................................................... 21
   Raw RSA keyrings ....................................................................................................................... 21
   Multi-keyrings ............................................................................................................................ 22

Supported algorithm suites .................................................................................................................... 25
   Recommended: AES-GCM with key derivation and signing ..................................................... 25
   Other supported algorithm suites ............................................................................................. 26

Programming languages ..................................................................................................................... 27
   C ................................................................................................................................................. 27
      Install and build ....................................................................................................................... 28
      Using the C SDK ..................................................................................................................... 35
      Examples ................................................................................................................................. 38
   Java ............................................................................................................................................. 43
      Prerequisites ............................................................................................................................ 43
      Installation ............................................................................................................................... 44
      Example code .......................................................................................................................... 44
   JavaScript ..................................................................................................................................... 51
      Compatibility ............................................................................................................................ 52
      Installation ............................................................................................................................... 53
      Modules ................................................................................................................................... 54
      Examples ................................................................................................................................. 55
   Python ......................................................................................................................................... 60
      Prerequisites ............................................................................................................................. 60
      Installation ............................................................................................................................... 60
      Example code .......................................................................................................................... 61
   Command line interface .............................................................................................................. 67
      Installing the CLI ..................................................................................................................... 68
What is the AWS Encryption SDK?

The AWS Encryption SDK is a client-side encryption library designed to make it easy for everyone to encrypt and decrypt data using industry standards and best practices. 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 is provided free of charge under the Apache 2.0 license.

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?

With the AWS Encryption SDK, you define a master key provider (p. 8) (Java or Python) or a keyring (C or JavaScript) that determines which master keys you use to protect your data. Then you encrypt and decrypt your data using straightforward methods provided by the AWS Encryption SDK. The AWS Encryption SDK does the rest.

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. 25).

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. 10) 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.

Some language implementations of the AWS Encryption SDK require an AWS SDK, but the AWS Encryption SDK doesn't require an AWS account and it doesn't depend on any AWS service. You need an AWS account only if you choose to use AWS Key Management Service (AWS KMS) customer master keys to protect your data.

Compatibility with encryption libraries and services

The AWS Encryption SDK is supported in several programming languages (p. 27). All language implementations are interoperable. You can encrypt with one language implementation and decrypt with another. Interoperability might be subject to language constraints. If so, these constraints are described in the topic about the language implementation. Also, when encrypting and decrypting, you must use compatible keyrings, or master keys and master key providers. For details, see the section called “Keyring compatibility” (p. 14).

However, the AWS Encryption SDK cannot interoperate with other libraries. Because each library returns encrypted data in a different format, you cannot encrypt with one library and decrypt with another.

DynamoDB Encryption Client and Amazon S3 client-side encryption

The AWS Encryption SDK cannot decrypt data encrypted by the DynamoDB Encryption Client or Amazon S3 client-side encryption. And these libraries cannot decrypt the encrypted message (p. 10) the AWS Encryption SDK returns.

AWS Key Management Service (AWS KMS)

The AWS Encryption SDK can use AWS KMS customer master keys (CMKs) and data keys to protect your data. For example, you can configure the AWS Encryption SDK to encrypt your data under one or more CMKs in your AWS account. However, you must use the AWS Encryption SDK to decrypt that data.

The AWS Encryption SDK cannot decrypt the ciphertext that the AWS KMS Encrypt or ReEncrypt operations return. Similarly, the AWS KMS Decrypt operation cannot decrypt the encrypted message (p. 10) the AWS Encryption SDK returns.

The AWS Encryption SDK supports only symmetric CMKs. You cannot use an asymmetric CMK for encryption or signing in the AWS Encryption SDK. The AWS Encryption SDK generates its own ECDSA signing keys for algorithm suites (p. 25) that sign messages.

For help deciding which library or service to use, see How to Choose an Encryption Tool or Service in AWS Cryptographic Services and Tools.

Learning more

For more information about the AWS Encryption SDK and client-side encryption, try these sources.

- To get started quickly, see Getting started (p. 11).
- For information about how this SDK works, see How the SDK works (p. 3).
• For help with the terms and concepts used in this SDK, see Concepts in the AWS Encryption SDK (p. 6).
• For detailed technical information, see the Reference (p. 133).
• For the technical specification for the AWS Encryption SDK, see the aws-encryption-sdk-specification repository in GitHub.
• 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.

• C: See AWS Encryption SDK for C (p. 27), the AWS Encryption SDK C documentation, and the aws-encryption-sdk-c repository on GitHub.
• Command Line Interface: See AWS Encryption SDK command line interface (p. 67), Read the Docs for the AWS Encryption CLI, and the aws-encryption-sdk-cli repository on GitHub.
• Java: See AWS Encryption SDK for Java (p. 43), the AWS Encryption SDK Javadoc, and the aws-encryption-sdk-java repository on GitHub.
• Python: See AWS Encryption SDK for Python (p. 60), the AWS Encryption SDK Python documentation, and the aws-encryption-sdk-python repository on GitHub.
• JavaScript: See the section called “JavaScript” (p. 51) and the aws-encryption-sdk-javascript repository on GitHub.

Sending feedback

We welcome your feedback! If you have a question or comment, or an issue to report, please use the following resources.

• If you discover a potential security vulnerability in the AWS Encryption SDK, please notify AWS security. Do not create a public GitHub issue.
• To provide feedback on the AWS Encryption SDK, file an issue in the GitHub repository for the programming language you are using.
• To provide feedback on this documentation, use the Feedback link on this page. You can also file an issue or contribute to aws-encryption-sdk-docs, the open-source repository for this documentation on GitHub.

How the AWS Encryption SDK works

The AWS Encryption SDK uses envelope encryption to protect your data and the corresponding data keys.

Topics

• Symmetric key encryption (p. 3)
• Envelope encryption (p. 4)
• AWS Encryption SDK encryption workflows (p. 5)

Symmetric key encryption

To encrypt data, the AWS Encryption SDK submits an encryption key, known as a data key and the plaintext data that you provide to an encryption algorithm. The encryption algorithm uses those inputs
to encrypt the data. Then, the AWS Encryption SDK returns an encrypted message (p. 10) that includes the encrypted data, an encrypted copy of the data key, and the encryption context (p. 9), if you used one.

To decrypt the encrypted message, the AWS Encryption SDK submits the data key and the encrypted message that the SDK returned to a decryption algorithm. The decryption algorithm 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.

![Symmetric key encryption and decryption diagram](image)

### Envelope encryption

The security of your encrypted data depends in part 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. 7). 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. 10).

**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 or wrapping 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 keyrings (p. 8) and master key providers (p. 8), 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. 10). They show how the SDK uses the components that you create, including a master key provider (p. 8) and master key (p. 7) (Java and Python) or keyring (p. 8) (C or JavaScript) 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 that encrypt and decrypt strings and byte streams in each supported programming language, see the examples in the Programming languages (p. 27) section.

1. Your application passes plaintext data to one of the encryption methods. We also recommend that you pass in an optional, non-secret encryption context (p. 9).
2. The encryption method asks the cryptographic materials manager (p. 7) (CMM) for encryption materials.

The CMM is a component that assembles the data keys, signing keys, and other encryption materials. The AWS Encryption SDK provides a default CMM and a CMM that manages data key
You can also create custom CMMs for your applications. Typically, you don't create a default CMM explicitly. When you specify a master key provider or keyring, the AWS Encryption SDK creates a default CMM for you.

3. The CMM requests encryption materials from the master key provider or keyring. The response includes a plaintext data key and the same data key encrypted under the master keys. The CMM returns these encryption materials to the encryption method.

4. The encryption method uses the plaintext data key to encrypt the data, and then discards the plaintext data key. If you provided an encryption context, the encryption method also cryptographically binds the encryption context to the encrypted data.

5. The encryption method returns an encrypted message (p. 10) that contains the encrypted data, the encrypted data key, and other metadata, including the encryption context, if one was used.

How the SDK decrypts an encrypted message

The SDK provides methods that decrypt the encrypted message (p. 10) and return plaintext strings, byte arrays, or byte streams. For code examples in each supported programming language, see the examples in the Programming languages (p. 27) section.

You must use the same master key provider and keyring to decrypt that you used to encrypt, or a compatible one.

1. Your application passes an encrypted message (p. 10) to a decryption method.

   To indicate the source of the data keys (p. 7) that were used to encrypt your data, your request specifies a cryptographic materials manager (CMM), or a master key provider or keyring. If you specify a master key provider or keyring, the AWS Encryption SDK creates a default CMM for you.

2. The decryption method asks the CMM for cryptographic materials to decrypt the encrypted message. It passes in information from the encrypted message, including the encrypted data keys.

3. In Java and Python, to get decryption materials, the Default CMM asks its master key provider for a master key that can decrypt one of the encrypted data keys. Other CMMs might use different techniques to get the decryption materials. In C and JavaScript, the CMM asks the keyring for decryption materials. The keyring uses its wrapping keys to decrypt one of the encrypted data keys.

   The response includes the decryption materials, including the plaintext data key.

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, and provides a glossary and reference.

Topics
- Data keys (p. 7)
- Master key (p. 7)
- Cryptographic materials manager (p. 7)
- Master key provider (Java and Python) (p. 8)
- Keyring (C and JavaScript) (p. 8)
- Algorithm suite (p. 8)
Data keys

A data key is an encryption key that the AWS Encryption SDK uses to encrypt your data. Each data key is a byte array that conforms to the requirements for cryptographic keys. Unless you’re using data key caching (p. 97), the AWS Encryption SDK uses a unique data key to encrypt each message.

To protect your data keys, the AWS Encryption SDK encrypts them under one or more key encryption keys known as master keys (p. 7) or wrapping keys. After the AWS Encryption SDK uses your plaintext data keys to encrypt your data, it removes them from memory as soon as possible. Then it stores the encrypted data keys with the encrypted data in the encrypted message (p. 10) that the encrypt operations return.

When you use the AWS Encryption SDK, you do not need to generate, implement, extend, protect or use data keys or master keys. The AWS Encryption SDK does that work for you when you call the encrypt and decrypt operations.

However, when you select your master key provider (Java and Python) or keyring (C and JavaScript), you determine the source of your master keys. You also determine whether your plaintext data key is encrypted by one or multiple master keys.

Tip
In the AWS Encryption SDK, we distinguish data keys from data encryption keys. Several of the supported algorithm suites (p. 8), 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, also known as a wrapping key, is an encryption key that is used to encrypt data keys. Each plaintext data key can be encrypted under one or more master keys.

When you use the AWS Encryption SDK, you do not need to generate, implement, extend, protect or use data keys or master keys. The AWS Encryption SDK does that work for you when you call the encrypt and decrypt operations. However, in the Java and Python implementations, you need to specify a cryptographic materials manager (p. 7) or a master key provider (p. 8) that supplies master keys. In the C and JavaScript implementations, you specify a keyring (p. 8) that interacts with wrapping keys for you and returns data keys.

The AWS Encryption SDK provides several commonly used master keys or wrapping keys, such as AWS Key Management Service (AWS KMS) symmetric customer master keys (CMKs), raw AES-GCM (Advanced Encryption Standard/Galois Counter Mode) keys, and RSA keys. You can also extend or implement your own master keys and wrapping keys.

Cryptographic materials manager

The cryptographic materials manager (CMM) assembles 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 or write a custom CMM. You can specify a CMM, but you never interact with it directly. The encryption and decryption methods handle it for you.
The default CMM gets the encryption or decryption materials from the master key provider (Java and Python) or keyring (C or JavaScript) that you specify. This might involve a call to a cryptographic service, such as AWS Key Management Service (AWS KMS).

You can specify a CMM and master key provider or keyring, but it's not required. If you specify a master key provider or keyring, the AWS Encryption SDK creates a Default CMM for you.

Because the CMM acts as a liaison between the SDK and a keyring or master key provider, it is an ideal point for customization and extension, such as support for policy enforcement and caching. The AWS Encryption SDK provides a caching CMM to support data key caching. (p. 97)

Master key provider (Java and Python)

In the AWS Encryption SDK for Java and the AWS Encryption SDK for Python, a master key provider returns master keys, or objects that identify or represent master keys. Each master key is associated with one master key provider, but a master key provider typically provides multiple master keys.

When you use the Java and Python implementations of the AWS Encryption SDK, you need to specify a cryptographic materials manager (p. 7) (CMM) or a master key provider, but you do not need to design or implement your own master key provider. If you specify a master key provider, the SDK creates a Default CMM for you based on the master key provider that you specify.

Master key providers in Java and Python are compatible with keyrings in the AWS Encryption SDK for C and the AWS Encryption SDK for JavaScript, subject to language constraints. However, you must specify the same key material and use a keyring that is compatible with the master key provider. For details, see Keyring compatibility (p. 14).

Keyring (C and JavaScript)

A keyring generates, encrypts, and decrypts data keys. Each keyring (p. 12) is typically associated with a wrapping key or a service that provides and protects wrapping keys. You can use the keyrings that the AWS Encryption SDK provides or write your own compatible custom keyrings. Keyrings are supported only in the AWS Encryption SDK for C and the AWS Encryption SDK for JavaScript.

You can use a single keyring or combine keyrings of the same type or a different type into a multi-keyring. The multi-keyring (p. 22) returns a copy of the data key encrypted by each of the wrapping keys in each of the keyrings that comprise the multi-keyring. When you use a multi-keyring to encrypt data, you can decrypt the data using a keyring configured with any one of the wrapping keys in the multi-keyring.

For details about working with keyrings, see Using keyrings (p. 12).

Keyrings in C and JavaScript are compatible with master key providers in the AWS Encryption SDK for Java and the AWS Encryption SDK for Python. However, you must specify the same key material and use a keyring that is compatible with the master key provider, subject to language constraints. Any minor incompatibility due to language constraints is explained in the topic about the language implementation. For details, see Keyring compatibility (p. 14).

Algorithm suite

The AWS Encryption SDK supports several algorithm suites (p. 25). All of the supported suites use Advanced Encryption Standard (AES) as the primary algorithm, and combine it with other algorithms 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
Encryption context

To improve the security of your cryptographic operations, include an encryption context in all requests to encrypt data. Using an encryption context is optional, but it is a cryptographic best practice that we recommend.

An encryption context is a set of name-value pairs that contain arbitrary, non-secret additional authenticated 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. When you encrypt data, the encryption context is cryptographically bound to the encrypted data so that the same encryption context is required to decrypt the data. The AWS Encryption SDK includes the encryption context in plaintext in the header of the encrypted message (p. 10) that it returns.

The encryption context that the AWS Encryption SDK uses consists of the encryption context that you specify and a public key pair that the cryptographic materials manager (p. 7) (CMM) adds. Specifically, whenever you use an encryption algorithm with signing (p. 147), the CMM adds a name-value pair to the encryption context that consists of a reserved name, aws-crypto-public-key, and a value that represents the public verification key. The aws-crypto-public-key name in the encryption context is reserved by the AWS Encryption SDK and cannot be used as a name in any other pair in the encryption context. For details, see AAD (p. 135) in the Message Format Reference.

The following example encryption context consists of two encryption context pairs specified in the request and the public key pair that the CMM adds.

```
  "Purpose"="Test", "Department"="IT", aws-crypto-public-key=<public key>
```

To decrypt the data, you pass in the encrypted message. Because the AWS Encryption SDK can extract the encryption context from the encrypted message header, you are not required to provide the encryption context separately. However, the encryption context can help you to confirm that you are decrypting the correct encrypted message.

- In the AWS Encryption SDK Command Line Interface (p. 67) (CLI), if you provide an encryption context in a decrypt command, the CLI verifies that the values are present in the encryption context of the encrypted message before it returns the plaintext data.
- In other languages, the decrypt response includes the encryption context and the plaintext data. The decrypt function in your application should always verify that the encryption context in the decrypt response includes the encryption context in the encrypt request (or a subset) 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. 10) 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.

For examples of verifying an encryption context in your code, see:

- C – Encrypting and Decrypting Strings (p. 38)
Encrypted message

When you encrypt data with the AWS Encryption SDK, it returns an encrypted message.

An encrypted message is a portable formatted data structure (p. 133) 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. Encrypt operations in the AWS Encryption SDK return an encrypted message and decrypt operations take an encrypted message as input.

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. 133).
Getting started with the AWS Encryption SDK

To use the AWS Encryption SDK, you need to configure keyring (p. 8) or master key provider (p. 8) with encryption keys. If you don't have a key infrastructure, we recommend using AWS Key Management Service (AWS KMS). Many of the code examples 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, the AWS SDK for Python (Boto), the AWS SDK for JavaScript, or the AWS SDK for C++, which you use with the AWS Encryption SDK for C. 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 key ID or Amazon Resource Name (ARN) of the CMK. For help finding the ID or ARN of a CMK, see Finding the Key ID and ARN 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, JavaScript, Python or C++ (for C), 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. 27) that you want to use.
Using keyrings

The AWS Encryption SDK for C and AWS Encryption SDK for JavaScript use keyrings to perform envelope encryption. Keyrings generate, encrypt, and decrypt data keys. The keyrings that you use determines the source of the unique data keys that protect each message, and the wrapping keys (p. 7) that encrypt that data key. You specify a keyring when encrypting and the same or a different keyring when decrypting. You can use the keyrings that the SDK provides or write your own compatible custom keyrings.

You can use each keyring individually or combine keyrings into a multi-keyring (p. 22). Although most keyrings can generate, encrypt, and decrypt data keys, you might create a keyring that performs only one particular operation, such as a keyring that only generates data keys, and use that keyring in combination with others.

We recommend that you use a keyring that protects your wrapping keys and performs cryptographic operations within a secure boundary, such as the AWS KMS keyring, which uses AWS Key Management Service (AWS KMS) customer master keys (CMKs) that never leave AWS KMS unencrypted. You can also write a keyring that uses wrapping keys that are stored in your hardware security modules (HSMs) or protected by other master key services. For details, see the Keyring Interface topic in the AWS Encryption SDK Specification.

This topic explains how to use the keyring feature of the AWS Encryption SDK and how to choose a keyring. For examples of creating and using keyrings, see the C (p. 27) and JavaScript (p. 51) topics.

Topics
- How keyrings work (p. 12)
- Keyring compatibility (p. 14)
- AWS KMS keyrings (p. 14)
- Raw AES keyrings (p. 21)
- Raw RSA keyrings (p. 21)
- Multi-keyrings (p. 22)

How keyrings work

One of the most important tasks you perform when using the AWS Encryption SDK for C or JavaScript is selecting and configuring a keyring. A keyring (p. 8) generates, encrypts, and decrypts data keys. Each keyring is typically associated with a wrapping key or a service that provides and protects wrapping keys. You can use the keyrings that the AWS Encryption SDK provides or write your own compatible custom keyrings. For help in choosing a keyring, see the following sections that describe each keyring.

Keyrings make it easier for you to determine which wrapping keys are used to encrypt and decrypt your data. Keyrings take the place of master key providers in the Java and Python implementations of the AWS Encryption SDK. Despite this architectural difference, all of the language implementations are fully interoperable, subject to language constraints. However, you must configure the keyring and master key provider with the same or corresponding wrapping keys. For details, see Keyring compatibility (p. 14).

You instantiate and configure your keyring, but you don't interact with it directly. The cryptographic materials manager (p. 7) (CMM) interacts with the keyring for you.

When you encrypt data, the CMM asks the keyring for encryption materials. The keyring returns a plaintext key and a copy of the key that's encrypted by each of the wrapping keys in the keyring. The AWS Encryption SDK uses the plaintext key to encrypt the data, and stores the encrypted data keys with the data in the encrypted message (p. 10) that it returns.
When you decrypt data, the CMM passes in the encryption keys from the encrypted message and asks the keyring to decrypt any one of them. The keyring uses its wrapping keys to decrypt one of the encrypted data keys and returns a plaintext data key. The AWS Encryption SDK uses the plaintext data key to decrypt the data. If none of the wrapping keys in the keyring can decrypt any of the encrypted data keys, the decrypt operation fails.

You can use a single keyring or also combine keyrings of the same type or a different type into a multi-keyring (p. 22). When you encrypt data, the multi-keyring returns a copy of the data key encrypted by...
all of the wrapping keys in all of the keyrings that comprise the multi-keyring. You can decrypt the data using a keyring configured with any one of the wrapping keys in the multi-keyring.

**Keyring compatibility**

Although the Java, Python, C, and JavaScript implementations of the AWS Encryption SDK have some architectural differences, they are designed to be fully compatible, subject to language constraints. You can encrypt your data using one programming language implementation and decrypt it with any other language implementation. However, you must use the same or corresponding master keys to encrypt and decrypt your data keys. For information about language constraints, see the topic about each language implementation, such as the section called “Compatibility” (p. 52) in the AWS Encryption SDK for JavaScript topic.

The following table shows which master keys and master key providers are compatible with the keyrings provided in the AWS Encryption SDK for C and the AWS Encryption SDK for JavaScript. Any minor incompatibility due to language constraints is explained in the topic about the language implementation.

**Compatible Keyrings and Master Key Providers**

<table>
<thead>
<tr>
<th>Keyring: C and JavaScript</th>
<th>Master key provider: Java and Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS KMS keyring (p. 14)</td>
<td>KmsMasterKey (Java)</td>
</tr>
<tr>
<td></td>
<td>KmsMasterKeyProvider (Java)</td>
</tr>
<tr>
<td></td>
<td>KmsMasterKey (Python)</td>
</tr>
<tr>
<td></td>
<td>KmsMasterKeyProvider (Python)</td>
</tr>
<tr>
<td>Raw AES keyring (p. 21)</td>
<td>When they are used with symmetric encryption keys:</td>
</tr>
<tr>
<td></td>
<td>JceMasterKey (Java)</td>
</tr>
<tr>
<td></td>
<td>RawMasterKey (Python)</td>
</tr>
<tr>
<td>Raw RSA keyring (p. 21)</td>
<td>When they are used with asymmetric encryption keys:</td>
</tr>
<tr>
<td></td>
<td>JceMasterKey (Java)</td>
</tr>
<tr>
<td></td>
<td>RawMasterKey (Python)</td>
</tr>
</tbody>
</table>

When decrypting, the Java and Python implementations behave like the AWS KMS discovery keyring (p. 18), that is, they don't limit the CMKs that can be used to decrypt the encrypted data keys. Also, the Java and Python SDKs don't supply an equivalent for the AWS KMS regional discovery keyring (p. 20), but you can create your own custom keyrings.

**AWS KMS keyrings**

An AWS KMS keyring uses AWS Key Management Service (AWS KMS) symmetric customer master keys (CMKS) to generate, encrypt, and decrypt data keys. AWS KMS protects your master keys and performs cryptographic operations within the FIPS boundary. We recommend that you use a AWS KMS keyring, or a keyring with similar security properties, whenever possible.

**Note**

All mentions of KMS keyrings in the AWS Encryption SDK refer to AWS KMS keyrings.

An AWS KMS keyring can have a *generator key*, which is the CMK that generates the plaintext data key that protects your data and encrypts it. It can also have additional CMKS that encrypt the same plaintext...
data key. When encrypting, the AWS KMS keyring that you use must have a generator key. Generator keys are not required for decrypting. When decrypting, any key in the AWS KMS keyring can be used to decrypt an encrypted data key.

Like all keyrings, AWS KMS keyrings can be used independently or in a multi-keyring (p. 22) with other keyrings of the same or a different type.

Topics

- Required permissions for AWS KMS keyrings (p. 15)
- Identifying CMKs in an AWS KMS keyring (p. 15)
- Encrypting with an AWS KMS keyring (p. 15)
- Decrypting with an AWS KMS keyring (p. 17)
- Using an AWS KMS discovery keyring (p. 18)
- Using an AWS KMS regional discovery keyring (p. 20)

Required permissions for AWS KMS keyrings

The AWS Encryption SDK doesn't require an AWS account and it doesn't depend on any AWS service. However, to use an AWS Key Management Service (AWS KMS) customer master key (CMK) in an AWS KMS keyring, you need an AWS account and the following minimum permissions on the CMKs in your keyring.

- To encrypt with an AWS KMS keyring, you need `kms:GenerateDataKey` permission on the generator key. You need `kms:Encrypt` permission on all additional keys in the AWS KMS keyring.
- To decrypt with an AWS KMS keyring, you need `kms:Decrypt` permission on at least one key in the AWS KMS keyring.
- To encrypt with a multi-keyring comprised of AWS KMS keyrings, you need `kms:GenerateDataKey` permission on the generator key in the generator keyring. You need `kms:Encrypt` permission on all other keys in all other AWS KMS keyrings.

For detailed information about permissions for AWS KMS customer master keys, see Authentication and Access Control for AWS KMS in the AWS Key Management Service Developer Guide.

Identifying CMKs in an AWS KMS keyring

An AWS KMS keyring can include one or more AWS KMS customer master keys (CMKs). To specify a CMK in an AWS KMS keyring, use a supported AWS KMS key identifier. The key identifiers you can use to identify a CMK in a keyring vary with the operation and the language implementation. For details about the key identifiers for a AWS KMS CMK, see Key Identifiers in the AWS Key Management Service Developer Guide.

- In an encryption keyring, you can use a key ARN or alias ARN to identify CMKs. Some language implementations allow other formats.
- In a decryption keyring, you must use a key ARN to identify CMKs. This requirement applies to all language implementations of the AWS Encryption SDK.
- In a keyring used for encryption and decryption, you must use a key ARN to identify CMKs. This requirement applies to all language implementations of the AWS Encryption SDK.

Encrypting with an AWS KMS keyring

You can configure each AWS KMS keyring with a single CMK or multiple CMKs in the same or different AWS accounts and AWS Regions. You can also configure an AWS KMS discovery keyring (p. 18) with no CMKs. As with all keyrings, you can use one or more AWS KMS keyrings in a multi-keyring (p. 22).
When you create an AWS KMS keyring to encrypt data, you must specify a *generator key*, which is a CMK that generates a plaintext data key and encrypts it. Then, if you choose, you can specify additional CMKs that encrypt the same plaintext data key. The *encrypted message* (p. 10) that the AWS Encryption SDK returns includes the ciphertext and all of the encrypted data keys. The caller must have `kms:GenerateDataKey` permission for the generator CMK, and `kms:Encrypt` permission for all additional CMKs.

For example, the following AWS KMS keyring specifies a generator key and one additional key. When you use this keyring to encrypt data, it returns one plaintext data key generated by the generator key and two encrypted data keys, one encrypted under the generator key and one encrypted under the additional key. To decrypt the data protected by this keyring, the keyring that you use must include at least one of the CMKs that encrypted the data key, or no CMKs. (An AWS KMS keyring with no CMKs is known as an *AWS KMS discovery keyring* (p. 18).)

**Note**
If you plan to use the same keyring for encrypting and decrypting data, use a key ARN to identify each CMK. Key ARNs are required when decrypting.

**C**

To identify an AWS KMS CMK in an encryption keyring, specify a *key ARN* or *alias ARN*. In a decryption keyring, you must use a key ARN. For details, see *Identifying CMKs in an AWS KMS keyring* (p. 15).

For a complete example, see *string.cpp*.

```c
const char * generator_key = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
const char * additional_key = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"
struct aws_cryptosdk_keyring *kms_encrypt_keyring =
  Aws::Cryptosdk::KmsKeyring::Builder().Build(generator_key,{additional_key});
```

**JavaScript Browser**

When you specify an AWS KMS CMK for an encryption keyring in the AWS Encryption SDK for JavaScript, you can use any valid CMK identifier: a *key ID*, *key ARN*, *alias name*, or *alias ARN*. For help identifying CMKs in an AWS KMS keyring, see *Identifying CMKs in an AWS KMS keyring* (p. 15).

For a complete example, see *kms_simple.ts*.

```javascript
const clientProvider = getClient(KMS, { credentials })
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
const additionalKey = 'alias/exampleAlias'

const keyring = new KmsKeyringBrowser({
  clientProvider,
  generatorKeyId,
  keyIds: [additionalKey]
});
```

**JavaScript Node.js**

When you specify an AWS KMS CMK for an encryption keyring in the AWS Encryption SDK for JavaScript, you can use any valid CMK identifier: a *key ID*, *key ARN*, *alias name*, or *alias ARN*. For help identifying CMKs in an AWS KMS keyring, see *Identifying CMKs in an AWS KMS keyring* (p. 15).

For a complete example, see *kms_simple.ts*.
Deciphering with an AWS KMS keyring

You also specify an AWS KMS keyring when decrypting the encrypted message (p. 10) that the AWS Encryption SDK returns. You can use a decryption keyring to determine which encrypted data keys can be decrypted. If the decryption keyring includes CMKs, the AWS Encryption SDK will only decrypt encrypted data keys that were encrypted by the CMKs in the keyring. (You can also use an AWS KMS discovery keyring (p. 18), which doesn’t specify any CMKs.)

When decrypting, the AWS Encryption SDK searches the AWS KMS keyring for a CMK that can decrypt one of the encrypted data keys. Specifically, the AWS Encryption SDK uses the following pattern for each encrypted data key in an encrypted message.

- The AWS Encryption SDK parses the metadata of the encrypted data key. It gets the key ARN of the CMK that encrypted the data key.
- The AWS Encryption SDK searches the decryption keyring for a CMK with a matching key ARN.
- If it finds a CMK with a matching key ARN in the keyring, the AWS Encryption SDK asks AWS KMS to decrypt the encrypted data key.
- Otherwise, it skips to the next encrypted data key, if any.

The AWS Encryption SDK never attempts to decrypt an encrypted data key unless the key ARN of the CMK that encrypted that data key is included in the decryption keyring. If the decryption keyring doesn’t include the ARNs of any of the CMKs that encrypted any of the data keys, the AWS Encryption SDK fails the decrypt call without ever calling AWS KMS.

A decrypt call with an AWS KMS keyring succeeds when at least one CMK in the decryption keyring can decrypt one of the encrypted data keys in the encrypted message. Also, the caller must have kms:Decrypt permission on that CMK. This behavior enables you to encrypt data under multiple CMKs in different AWS Regions and accounts, but provide a more limited decryption keyring tailored to a particular account, Region, user, group, or role.

When you specify a CMK in a decryption keyring, you must use its key ARN. Otherwise, the CMK is not recognized. For help finding the key ARN, see Finding the Key ID and ARN in the AWS Key Management Service Developer Guide.

Note
If you reuse an encryption keyring for decrypting, be sure that the CMKs in the keyring are identified by their key ARNs.

For example, the following AWS KMS keyring includes only the additional key that was used in the encryption keyring. You can use this keyring to decrypt a message that was encrypted under both the generator key and the additional key, provided that you have permission to use the additional key to decrypt data.

```
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
const additionalKeyId = 'alias/exampleAlias'

const keyring = new KmsKeyringNode({
    generatorKeyId,
    keyIds: [additionalKeyId]
})
```
Using an AWS KMS discovery keyring

You can also use an AWS KMS keyring that specifies a generator key for decrypting, such as the following one. When decrypting, the AWS Encryption SDK ignores the distinction between generator keys and additional keys. It can use any of the specified CMKs to decrypt an encrypted data key. The call to AWS KMS succeeds only when the caller has permission to use that CMK to decrypt data.

```
struct aws_cryptosdk_keyring *kms_decrypt_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(generator_key, {additional_key, other_cmk});
```

```
const clientProvider = getClient(KMS, { credentials })
const keyring = new KmsKeyringNode({
    generatorKeyId,
    keyIds: [additionalKey, otherCmk]
})
```

Unlike an encryption keyring that uses all of the specified CMKs, you can decrypt an encrypted message using a decryption keyring that includes CMKs that are unrelated to the encrypted message, and CMKs that the caller doesn't have permission to use. If a decrypt call to AWS KMS fails, such as when the caller doesn't have the required permission, the AWS Encryption SDK just skips to the next encrypted data key.

**Using an AWS KMS discovery keyring**

Typically, when decrypting, you provide an AWS KMS keyring that limits the CMKs that the AWS Encryption SDK can use to those that you specify. However, you can also create an *AWS KMS discovery*
keyring, that is, an AWS KMS keyring that doesn't specify any CMKs. It allows the AWS Encryption SDK to ask AWS KMS to decrypt any encrypted data key in an encrypted message by using the CMK encrypted it, regardless of who owns or has access to that CMK. The call succeeds only if the caller has `kms:Decrypt` permission on the CMK.

The following code instantiates an AWS KMS discovery keyring.

```
C

For a complete example, see: kms_discovery.cpp.

```struct kms_discovery_keyring = Aws::Cryptosdk::KmsKeyring::Builder().BuildDiscovery();
```

JavaScript Browser

In JavaScript, you must explicitly specify the discovery property.

```
const clientProvider = getClient(KMS, { credentials })
const keyring = new KmsKeyringBrowser({discovery: true})
```

JavaScript Node.js

In JavaScript, you must explicitly specify the discovery property.

```
const keyring = new KmsKeyringNode({discovery: true})
```

When encrypting, an AWS KMS discovery keyring has no effect. It doesn't return any encrypted data keys. However, you might want to include an AWS KMS discovery keyring in a multi-keyring that will be used for encrypting and decrypting.

When decrypting with an AWS KMS discovery keyring, the AWS Encryption SDK calls AWS KMS to decrypt each of the encrypted data keys in the encrypted message in the order that they appear until one succeeds. To succeed, the caller must have `kms:Decrypt` permission on at least one of the CMKs that encrypted one of the encrypted data keys.

The AWS Encryption SDK provides an AWS KMS discovery keyring for convenience. However, we recommend that you use a more limited keyring whenever possible for the following reasons.

- **Authenticity** – An AWS KMS discovery keyring can use any CMK that was used to encrypt a data key in the encrypted message, just so the caller has permission to use that CMK to decrypt. This might not be the CMK that the caller intends to use. For example, one of the encrypted data keys might have been encrypted under a less secure CMK that anyone can use.

- **Latency and performance** – An AWS KMS discovery keyring might be perceptibly slower than other keyrings because the AWS Encryption SDK tries to decrypt all of the encrypted data keys, including those encrypted by CMKs in other AWS accounts and Regions, and CMKs that the caller doesn't have permission to use for decryption.

When you use an AWS KMS discovery keyring in a multi-keyring, it has no effect on encryption. When decrypting, by allowing the use of any CMK, an AWS KMS discovery keyring overrides any CMK limits that other AWS KMS keyrings in the multi-keyring might impose. For example, if you combine an AWS KMS keyring that uses a particular CMK with an AWS KMS discovery keyring, the resulting multi-keyring behaves just like the KMS Discovery keyring. It lets the AWS Encryption SDK ask AWS KMS to decrypt any encrypted data key, even if it was encrypted by a different CMK.
Using an AWS KMS regional discovery keyring

Instead of using an AWS KMS keyring that specifies particular CMKs, or an AWS KMS discovery keyring that can use any CMK, you might want to use an AWS KMS regional discovery keyring that includes or excludes the AWS Encryption SDK to CMKs in a particular AWS Region.

For example, the following code creates an AWS KMS regional discovery keyring that uses only CMKs in the US West (Oregon) Region (us-west-2).

C

To view this keyring, and the create_kms_client method, in a working example, see kms_discovery.cpp.

```c
struct aws_cryptosdk_keyring *kms_regional_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder()
        .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery();
```

JavaScript Browser

```javascript
const clientProvider = getClient(KMS, { credentials })
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser({ clientProvider, discovery })
```

JavaScript Node.js

To view this keyring, and the limitRegions and function, in a working example, see kms_regional_discovery.ts.

```javascript
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({ clientProvider, discovery })
```

The AWS Encryption SDK for JavaScript also exports an excludeRegions function for Node.js and the browser. This function creates an AWS KMS regional discovery keyring that omits CMKs in particular regions. The following example creates an AWS KMS regional discovery keyring that can use CMKs in every AWS Region except for US East (N. Virginia) (us-east-1).

The AWS Encryption SDK for C does not have an analogous method, but you can implement one by creating a custom ClientSupplier.

This example shows the code for Node.js.

```javascript
const discovery = true
const clientProvider = excludeRegions(['us-east-1'], getKmsClient)
const keyring = new KmsKeyringNode({ clientProvider, discovery })
```

When encrypting, an AWS KMS regional discovery keyring has no effect. Because it doesn’t specify any CMKs, it can’t generate or encrypt data keys. However, you can include an AWS KMS regional discovery keyring in a multi-keyring that will be used for encrypting and decrypting.

When decrypting with an AWS KMS regional discovery keyring, the AWS Encryption SDK can ask AWS KMS to decrypt any encrypted data key that was encrypted under a CMK in the specified AWS Region. To succeed, the caller must have kms:Decrypt permission on at least one of the CMKs in the specified AWS Region that encrypted one of the data keys in the encrypted message.
In a multi-keyring, because it allows the use of any CMK in the AWS Region, an AWS KMS regional discovery keyring can override CMK limits that other AWS KMS keyrings in the multi-keyring might impose. For example, if you combine an AWS KMS keyring that lets you use a particular CMK in the Europe (London) Region and a KMS Regional Discovery keyring for the Europe (London) Region, the resulting multi-keyring behaves just like the AWS KMS regional discovery keyring alone. It lets the AWS Encryption SDK ask AWS KMS to decrypt any encrypted data key that was encrypted by any CMK in the Europe (London) Region.

Raw AES keyrings

The Raw AES keyring uses the AES-GCM algorithm and a wrapping key that you specify as a byte array to encrypt data keys. You can specify only one wrapping key in each Raw AES keyring, but you can include multiple Raw AES keyrings in each multi-keyring (p. 22).

The Raw AES keyring is equivalent to and interoperates with the JceMasterKey in the AWS Encryption SDK for Java and the RawMasterKey in the AWS Encryption SDK for Python when they are used with symmetric encryption keys. You can encrypt data with one implementation and decrypt the data with any other implementation using the same wrapping key.

Use a Raw AES keyring when you need to provide the wrapping key and encrypt the data keys locally, or you need to write an application that is compatible with the AWS Encryption SDK for Java or the AWS Encryption SDK for Python. Whenever possible, we recommend using a hardware security module (HSM) or a service, such as AWS KMS, that doesn’t expose wrapping keys and encrypts data keys within a secure boundary.

To identify the wrapping key, the Raw AES keyring uses a namespace and name that you provide. These are equivalent to the Provider ID and Key ID fields in the AWS Encryption SDK for Java and the AWS Encryption SDK for Python. These values are not secret. They appear in plaintext in the header of the encrypted message (p. 10) that the AWS Encryption SDK returns. However, they are critical. You must use the same namespace and name for a key in your decryption keyring that you used for that key in your encryption keyring. If the namespace and name are not an exact, case-sensitive match, the AWS Encryption SDK will not recognize that the wrapping keys are the same, even if the bytes are identical, and it will not be able to decrypt the encrypted data keys.

For an example of how to use a Raw AES keyring, see:

- C: raw_aes_keyring.c
- JavaScript Node.js: aes_simple.ts
- JavaScript Browser: aes_simple.ts

Raw RSA keyrings

The Raw RSA keyring performs asymmetric encryption and decryption of data keys in local memory with an RSA public and private keys that you specify. The encryption function encrypts the data key under the RSA public key. The decryption function decrypts the data key using the private key. You can select from among the several RSA padding modes.

A Raw RSA keyring that encrypts and decrypts must include an asymmetric public key and private key pair. But you can encrypt data with a Raw RSA keyring that has only a public key, and you can decrypt data with a Raw RSA keyring that has only a private key. And you can include any Raw RSA keyring in a multi-keyring (p. 22). The Raw RSA keyring is equivalent to and interoperates with the JceMasterKey in the AWS Encryption SDK for Java and the RawMasterKey in the AWS Encryption SDK for Python when they are used with asymmetric encryption keys. You can encrypt data with one implementation and decrypt the data with any other implementation using the same wrapping key.
Use a Raw RSA keyring when you want to use an asymmetric key pair and provide the wrapping key and unwrapping keys, or you need to be compatible with the AWS Encryption SDK for another programming language.

To identify the key pair, the Raw RSA keyring uses a namespace and name that you provide. These values are not secret. They appear in plaintext in the header of the encrypted message (p. 10) that the AWS Encryption SDK returns. However, they are critical. If you use the same key pair in an encryption keyring and a decryption keyring, be sure to use the same namespace and name for the key pair in both keyrings. If the namespace and name are not an exact, case-sensitive match, the AWS Encryption SDK will not recognize that the asymmetric keys are a pair, and it will not be able to decrypt the encrypted data keys.

When constructing a Raw RSA keyring in the AWS Encryption SDK for C, be sure to provide the contents of the PEM file that includes each key as a null-terminated C-string, not as a path or file name. When constructing a Raw RSA keyring in JavaScript, be aware of potential incompatibility (p. 52) with other language implementations.

For an example of how to use a Raw RSA keyring, see:

- C: raw_rsa_keyring.c
- JavaScript Node.js: rsa_simple.ts
- JavaScript Browser: rsa_simple.ts

Multi-keyrings

You can combine keyrings into a multi-keyring. A multi-keyring is a keyring that consists of one or more individual keyrings of the same or a different type. The effect is like using several keyrings in a series. When you use a multi-keyring to encrypt data, any of the wrapping keys in any of its keyrings can decrypt that data.

When you create a multi-keyring to encrypt data, you designate one of the keyrings as the generator keyring. All other keyrings are known as child keyrings. The generator keyring generates and encrypts the plaintext data key. Then, all of the wrapping keys in all of the child keyrings encrypt the same plaintext data key. The multi-keyring returns the plaintext key and one encrypted data key for each wrapping key in the multi-keyring. If you create a multi-keyring with no generator keyring, you can use it to decrypt data, but not to encrypt. If the generator keyring is a KMS keyring (p. 14), the generator key in the AWS KMS keyring generates and encrypts the plaintext key. Then, all additional CMKs in the AWS KMS keyring, and all wrapping keys in all child keyrings in the multi-keyring, encrypt the same plaintext key.

When decrypting, the AWS Encryption SDK uses the keyrings to try to decrypt one of the encrypted data keys. The keyrings are called in the order that they are specified in the multi-keyring. Processing stops as soon as any key in any keyring can decrypt an encrypted data key.

To see a working example of a multi-keyring, see:

- C: multi_keyring.cpp
- JavaScript Node.js: multi_keyring.ts
- JavaScript Browser: multi_keyring.ts

To create a multi-keyring, first instantiate the child keyrings. In this example, we use an AWS KMS keyring and a Raw AES keyring, but you can combine any supported keyrings in a multi-keyring.

C

```c
// Define an AWS KMS keyring. For details, see string.cpp.
struct aws_cryptosdk_keyring *kms_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(example_CMK);
```
// Define a Raw AES keyring. For details, see raw_aes_keyring.c.
struct aws_cryptosdk_keyring *aes_keyring = aws_cryptosdk_raw_aes_keyring_new(
    alloc, wrapping_key_namespace, wrapping_key_name, wrapping_key,
    AWS_CRYPTOSDK_AES256);

JavaScript Browser

const clientProvider = getClient(KMS, { credentials })

// Define an AWS KMS keyring. For details, see kms_simple.ts.
const kmsKeyring = new KmsKeyringBrowser({ generatorKeyId: exampleCmk })

// Define a Raw AES keyring. For details, see aes_simple.ts.
const aesKeyring = new RawAesKeyringWebCrypto({ keyName, keyNamespace, wrappingSuite, masterKey })

JavaScript Node.js

// Define an AWS KMS keyring. For details, see kms_simple.ts.
const kmsKeyring = new KmsKeyringNode({ generatorKeyId: exampleCmk })

// Define a Raw AES keyring. For details, see raw_aes_keyring_node.ts.
const aesKeyring = new RawAesKeyringNode({ keyName, keyNamespace, wrappingSuite, unencryptedMasterKey })

Next, create the multi-keyring and specify its generator keyring, if any. In this example, we create a multi-keyring in which the AWS KMS keyring is the generator keyring and the AES keyring is the child keyring.

C

In the multi-keyring constructor in C, you specify only its generator keyring.

struct aws_cryptosdk_keyring *multi_keyring = aws_cryptosdk_multi_keyring_new(alloc, kms_keyring);

To add a child keyring to your multi-keyring, use the aws_cryptosdk_multi_keyring_add_child method. You need to call the method once for each child keyring that you add.

// Add the Raw AES keyring (C only)
aws_cryptosdk_multi_keyring_add_child(multi_keyring, aes_keyring);

JavaScript Browser

JavaScript multi-keyrings are immutable. The JavaScript multi-keyring constructor lets you specify the generator keyring and multiple child keyrings.

const clientProvider = getClient(KMS, { credentials })

const multiKeyring = new MultiKeyringWebCrypto(generator: kmsKeyring, children: [aesKeyring]);

JavaScript Node.js

JavaScript multi-keyrings are immutable. The JavaScript multi-keyring constructor lets you specify the generator keyring and multiple child keyrings.
```javascript
const multiKeyring = new MultiKeyringNode(generator: kmsKeyring, children: [aesKeyring]);
```

Now, you can use the multi-keyring to encrypt and decrypt data.

For a complete example of how to build and use a multi-keyring in C, see `multi_keyring.cpp`.
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 implementation 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 supports several different implementations of AES-GCM. 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

<table>
<thead>
<tr>
<th>Algorithm name</th>
<th>Data encryption key length (in bits)</th>
<th>Algorithm mode</th>
<th>Key derivation algorithm</th>
<th>Signature algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>256</td>
<td>GCM</td>
<td>HKDF with SHA-384</td>
<td>ECDSA with P-384 and SHA-384</td>
</tr>
<tr>
<td>AES</td>
<td>192</td>
<td>GCM</td>
<td>HKDF with SHA-384</td>
<td>ECDSA with P-384 and SHA-384</td>
</tr>
<tr>
<td>AES</td>
<td>128</td>
<td>GCM</td>
<td>HKDF with SHA-256</td>
<td>ECDSA with P-256 and SHA-256</td>
</tr>
</tbody>
</table>
Other supported algorithm suites

The AWS Encryption SDK supports the following alternate algorithm suites for backward compatibility. In general, we do not recommend these algorithm suites. However, we recognize that using an algorithm suite with key derivation, but without signing, is appropriate in some cases. We discourage the use of any algorithm suite that lacks both key derivation and signing.

**AES-GCM with key derivation only**

This algorithm suite uses a key derivation function, but lacks the ECDSA signature that provides authenticity and non-repudiation. 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. 147).
AWS Encryption SDK programming languages

The AWS Encryption SDK is available for the following programming languages. All language implementations are interoperable. You can encrypt with one language implementation and decrypt with another. Interoperability might be subject to language constraints. If so, these constraints are described in the topic about the language implementation. Also, when encrypting and decrypting, you must use compatible keyrings, or master keys and master key providers. For details, see the section called “Keyring compatibility” (p. 14).

Topics
- AWS Encryption SDK for C (p. 27)
- AWS Encryption SDK for Java (p. 43)
- AWS Encryption SDK for JavaScript (p. 51)
- AWS Encryption SDK for Python (p. 60)
- AWS Encryption SDK command line interface (p. 67)

AWS Encryption SDK for C

The AWS Encryption SDK for C provides a client-side encryption library for developers who are writing applications in C. It also serves as a foundation for implementations of the AWS Encryption SDK in higher-level programming languages.

Like all implementations of the AWS Encryption SDK, the AWS Encryption SDK for C offers advanced data protection features. These include envelope encryption, additional authenticated data (AAD), and secure, authenticated, symmetric key algorithm suites (p. 8), such as 256-bit AES-GCM with key derivation and signing.

All language-specific implementations of the AWS Encryption SDK are fully interoperable. For example, you can encrypt data with the AWS Encryption SDK for C and decrypt it with any supported language implementation (p. 27), including the AWS Encryption CLI (p. 67).

The AWS Encryption SDK for C uses the AWS SDK for C++ to interact with AWS Key Management Service (AWS KMS) so it can support the optional AWS KMS keyring (p. 14). However, the AWS Encryption SDK doesn’t require AWS KMS or any other AWS service.

Learn More
- For details about programming with the AWS Encryption SDK for C, see the C examples (p. 38), the examples in the aws-encryption-sdk-c repository on GitHub, and the AWS Encryption SDK for C API documentation.
- For a discussion about how to use the AWS Encryption SDK for C to encrypt data so that you can decrypt it in multiple AWS Regions, see How to decrypt ciphertexts in multiple regions with the AWS Encryption SDK in C in the AWS Security Blog.

Topics
- Installing the AWS Encryption SDK for C (p. 28)
- Using the AWS Encryption SDK for C (p. 35)
Installing the AWS Encryption SDK for C

This topic explains how to install the AWS Encryption SDK for C on all supported platforms. If you're having trouble with your installation, file an issue on the aws-encryption-sdk-c repository or use the Feedback link in the lower-right corner of this page.

Before you begin, decide whether you want to use AWS Key Management Service (AWS KMS) to generate and protect the encryption keys that protect your data. Otherwise, you need to provide master keys that you generate and protect. If you use AWS KMS, you need to install the AWS SDK for C++.

If you don’t use hardware security modules (HSMs) or another encryption key management service, we recommend AWS KMS. You need to create at least one AWS KMS customer master key. For instructions, see Creating Keys.

Topics
• Required libraries (p. 28)
• Install the AWS Encryption SDK for C (p. 28)
• Compile and build (p. 35)

Required libraries

The AWS Encryption SDK for C requires the following libraries:

• OpenSSL 1.0.2 or greater, including 1.1.0 or later – The AWS Encryption SDK uses the encryption algorithms and other cryptographic primitives in the OpenSSL cryptography library. The SDK doesn't define or create any cryptographic primitives.

• aws-c-common 0.3.0 or later – This library defines general-purpose tools and structures that the AWS Encryption SDK for C uses. It's installed automatically when you install the AWS SDK for C++.

  Note
  The preferred version of the aws-c-common library is installed automatically when you install the AWS SDK for C++. If you're installing the SDK for C++, don't install the aws-c-common library separately.

To build the SDK, you need:

• A C compiler for your platform.
• CMake 3.9 or later.

To use the AWS Encryption SDK for C with AWS KMS, you need the following:

• AWS SDK for C++ 1.7.36 or later – The AWS Encryption SDK for C uses the AWS SDK for C++ to interact with AWS KMS. To use AWS KMS to protect your encryption keys, and to run many of the examples in this guide and in the repository, install the SDK for C++. This C++ SDK requires a C++ compiler and the curl tool.

Install the AWS Encryption SDK for C

Use the following procedure to install the required components and build the AWS Encryption SDK for C on your preferred platform.
These instructions install the AWS Encryption SDK for C on an Amazon Elastic Compute Cloud (Amazon EC2) instance with Amazon Linux 1.

1. **Install CMake 3.9 or later.**

   Download and install CMake 3.9 or later. Then, confirm that CMake is in your path. Don't use the earlier version of CMake that `yum` installs.

2. **Install the dependent libraries.**

   These instructions install the tools that the AWS Encryption SDK requires, including the `aws-c-common` library, which includes many of the basic functions that the AWS Encryption SDK for C uses.

Run option A or B, but not both.

If you plan to use a KMS keyring (p. 14), or run the examples (p. 38) in the AWS Encryption SDK for C that use the AWS KMS keyring, use installation option A. This option installs the AWS SDK for C++, which is required for the AWS Encryption SDK to interact with AWS Key Management Service (AWS KMS). Installing the SDK for C++ automatically installs the `aws-c-common` library for you.

**Option A: Install with AWS SDK for C++ (for AWS KMS).**

The AWS SDK for C++ is required to use the AWS Encryption SDK for C with AWS KMS. It is also required for many of the examples (p. 38) in this guide and the examples in the `aws-encryption-sdk-c` repository.

Run the following commands to download and install the dependent libraries for the AWS Encryption SDK and SDK for C++.

```
sudo yum update -y
sudo yum install -y openssl-devel git libcurl-devel gcc-c++
```

Next, to download and install the SDK for C++, change to your preferred build directory and run the following commands. When you install the SDK for C++, the `aws-c-common` is installed automatically. **Do not install it again.**

These commands install only the AWS KMS modules of the SDK for C++. If you are using other libraries in the SDK for C++, you can omit the `-DBUILD_ONLY="kms"` parameter, but it might take an extended amount of time to install.

```
git clone https://github.com/aws/aws-sdk-cpp.git
mkdir build-aws-sdk-cpp && cd build-aws-sdk-cpp
make -DBUILD_SHARED_LIBS=ON -DBUILD_ONLY="kms" -DENABLE_UNITY_BUILD=ON ../aws-sdk-cpp
make && sudo make install ; cd ..
```
Option B: Install without SDK for C++.

Use these installation instructions only if you are not using the AWS Encryption SDK for C with AWS KMS or running the examples that use AWS KMS. These components do not allow you to use the AWS Encryption SDK for C with AWS KMS.

Run the following commands to download and install the dependent libraries for the AWS Encryption SDK.

```
sudo yum update -y
sudo yum install -y openssl-devel git gcc
```

Next, to download and install the `aws-c-common` library, change to your preferred build directory, and run the following commands.

```
git clone https://github.com/awslabs/aws-c-common.git
mkdir build-aws-c-common && cd build-aws-c-common
make -DBUILD_SHARED_LIBS=ON ../aws-c-common
make && sudo make install ; cd ..
```

3. **Install the AWS Encryption SDK.**

Change to your preferred directory. Then, run the following commands to install and build the AWS Encryption SDK for C.

```
git clone https://github.com/aws/aws-encryption-sdk-c.git
mkdir build-aws-encryption-sdk-c && cd build-aws-encryption-sdk-c
make -DBUILD_SHARED_LIBS=ON ../aws-encryption-sdk-c
make && sudo make install ; cd ..
```

**macOS**

These instructions install the AWS Encryption SDK in standard directories in `/usr/local` using the Homebrew package management system. If you don't have it yet, install Homebrew, then follow these instructions.

1. **Install the dependent libraries.**

The following command installs the required versions of OpenSSL and CMake.

```
brew install openssl@1.1 cmake
```

2. **Install the `aws-c-common` library and (optionally) the AWS SDK for C++.**

These instructions install the tools that the AWS Encryption SDK requires, including the `aws-c-common` library, which includes many of the basic functions that the AWS Encryption SDK for C uses.

Run option **A** or **B**, but not both.
If you plan to use a KMS keyring (p. 14), or run the examples (p. 38) in the AWS Encryption SDK for C that use the AWS KMS keyring, use installation option A. This option installs the AWS SDK for C++, which is required for the AWS Encryption SDK to interact with AWS Key Management Service (AWS KMS). Installing the SDK for C++ automatically installs the aws-c-common library for you.

**Option A: Install with AWS SDK for C++ (for AWS KMS).**

The AWS SDK for C++ is required to use the AWS Encryption SDK for C with AWS KMS. It is also required for many of the examples (p. 38) in this guide and the examples in the aws-encryption-sdk-c repository.

Run the following commands to download and install the dependent libraries for the AWS Encryption SDK and SDK for C++.

When you install the SDK for C++, the aws-c-common is installed automatically. Do not install it again.

```
git clone https://github.com/aws/aws-sdk-cpp.git
mkdir build-aws-sdk-cpp && cd build-aws-sdk-cpp
cmake -DBUILD_SHARED_LIBS=ON -DBUILD_ONLY="kms" -DENABLE_UNITY_BUILD=ON ../aws-sdk-cpp
make && sudo make install ; cd ..
```

**Option B: Install without SDK for C++.**

Use these installation instructions only if you aren't using the AWS Encryption SDK for C with AWS KMS or running the examples that use AWS KMS. These components don't allow you to use the AWS Encryption SDK for C with AWS KMS.

Run the following commands to install the aws-c-common library that the AWS Encryption SDK for C requires.

```
git clone https://github.com/awslabs/aws-c-common.git
mkdir build-aws-c-common && cd build-aws-c-common
cmake -DBUILD_SHARED_LIBS=ON ../aws-c-common
make && sudo make install ; cd ..
```

### 3. Install the AWS Encryption SDK.

Change to your preferred directory. Then, run the following commands to install and build the AWS Encryption SDK for C. These commands specify the location of the OpenSSL library, because HomeBrew doesn't install it in the default location.

```
git clone https://github.com/aws/aws-encryption-sdk-c-c.git
mkdir build-awas-encryption-sdk-c-c && cd build-aws-encryption-sdk-c-c
cmake -DBUILD_SHARED_LIBS=ON -DOPENSSL_ROOT_DIR=/usr/local/opt/openssl@1.1 ../aws-encryption-sdk-c-c
make && sudo make install ; cd ..
```
Ubuntu

These instructions install the AWS Encryption SDK for C on Ubuntu.

1. **Install dependent libraries.**

These instructions install the tools that the AWS Encryption SDK requires, including the `aws-c-common` library, which includes many of the basic functions that the AWS Encryption SDK for C uses.

Run option A or B, but not both.

If you plan to use a KMS keyring (p. 14), or run the examples (p. 38) in the AWS Encryption SDK for C that use the AWS KMS keyring, use installation option A. This option installs the AWS SDK for C++, which is required for the AWS Encryption SDK to interact with AWS Key Management Service (AWS KMS). Installing the SDK for C++ automatically installs the `aws-c-common` library for you.

**Option A: Install with AWS SDK for C++ (for AWS KMS).**

The AWS SDK for C++ is required to use the AWS Encryption SDK for C with AWS KMS. It is also required for many of the examples (p. 38) in this guide and the examples in the `aws-encryption-sdk-c` repository.

Run the following commands to download and install the dependent libraries for the AWS Encryption SDK and SDK for C++.

```bash
sudo apt-get update
sudo apt-get install -y libssl-dev cmake g++ libcurl4-openssl-dev zlib1g-dev
```

Next, to download and install the SDK for C++, change to your preferred build directory and run the following commands. When you install the SDK for C++, the `aws-c-common` is installed automatically. *Do not install it again.*

These commands install only the AWS KMS modules of the SDK for C++. If you are using other libraries in the SDK for C++, you can omit the `-DBUILD_ONLY="kms"` parameter, but it might take an extended amount of time to install.

```bash
git clone https://github.com/aws/aws-sdk-cpp.git
mkdir build-aws-sdk-cpp & cd build-aws-sdk-cpp
cmake -DBUILD_SHARED_LIBS=ON -DBUILD_ONLY="kms" -DENABLE_UNITY_BUILD=ON ../aws-sdk-cpp
make & sudo make install ; cd ..
```

**Option B: Install without SDK for C++.**

Use these installation instructions only if you are not using the AWS Encryption SDK for C with AWS KMS or running the examples that use AWS KMS. These components do not allow you to use the AWS Encryption SDK for C with AWS KMS.
Run the following commands to download and install the dependent libraries for the AWS Encryption SDK.

```
sudo apt-get update
sudo apt-get install -y libssl-dev cmake gcc
```

Next, to download and install the `aws-c-common` library, change to your preferred build directory, and run the following commands.

```
git clone https://github.com/awslabs/aws-c-common.git
mkdir build-aws-c-common && cd build-aws-c-common
cmake -DBUILD_SHARED_LIBS=ON ../aws-c-common
make && sudo make install ; cd ..
```

2. **Install the AWS Encryption SDK.**

Run the following commands to install and build the AWS Encryption SDK for C.

```
git clone https://github.com/aws/aws-encryption-sdk-c.git
mkdir build-aws-encryption-sdk-c && cd build-aws-encryption-sdk-c
cmake -DBUILD_SHARED_LIBS=ON ../aws-encryption-sdk-c
make && sudo make install ; cd ..
```

Windows

The following instructions install the AWS Encryption SDK and its dependencies. They require Visual Studio 15 or later, Git for Windows, and the Universal C Runtime Library. Run the following commands in the **x64 Native Tools Command Prompt for Visual Studio** in your preferred build and installation directory.

1. **Install the dependent libraries.**

These commands install the `vcpkg` library manager. Then, use `vcpkg` to install `curl` and OpenSSL.

```
mkdir install && mkdir build && cd build
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg && .\bootstrap-vcpkg.bat
\vcpkg install curl:x64-windows openssl:x64-windows && cd ..
```

2. **Install the aws-c-common library and (optionally) the AWS SDK for C++.**

These instructions install the tools that the AWS Encryption SDK requires, including the `aws-c-common` library, which includes many of the basic functions that the AWS Encryption SDK for C uses.

Run option **A** or **B**, but not both.

If you plan to use a **KMS keyring** (p. 14), or run the **examples** (p. 38) in the AWS Encryption SDK for C that use the AWS KMS keyring, use installation option **A**. This option installs the AWS SDK for C++, which is required for the AWS Encryption SDK to interact with AWS Key
Management Service (AWS KMS). Installing the SDK for C++ automatically installs the aws-c-common library for you.

**Option A: With AWS SDK for C++ for AWS KMS**

The AWS SDK for C++ is required to use the AWS Encryption SDK for C with AWS KMS. It is also required for many of the examples (p. 38) in this guide and the examples in the aws-encryption-sdk-c repository.

The following commands install only the AWS KMS modules of the SDK for C++. If you are using other libraries in the SDK for C++, you can omit the `-DBUILD_ONLY="kms"` parameter, but it might take an extended amount of time to install.

Run the following commands in your preferred build directory. When you install the SDK for C++, the aws-c-common is installed automatically. Do not install it again.

```
git clone https://github.com/aws/aws-sdk-cpp.git
mkdir build-aws-sdk-cpp && cd build-aws-sdk-cpp
cmake -DCMAKE_INSTALL_PREFIX=%cd%../../../../install -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=ON -DBUILD_ONLY=kms -DCMAKE_TOOLCHAIN_FILE=%cd%../../../../vcpkg/scripts/buildsystems/vcpkg.cmake -G Ninja ..
aws-sdk-cpp
cmake --build . & & cmake --build . --target install & & cd ..
```

**Option B: Without SDK for C++**

Use these installation instructions only if you aren't using the AWS Encryption SDK for C with AWS KMS or running the examples that use AWS KMS.

Run the following commands in your preferred build directory. These commands install and build the aws-c-common library. These components don't allow you to use the AWS Encryption SDK for C with AWS KMS.

```
git clone https://github.com/awslabs/aws-c-common.git
mkdir build-aws-c-common & & cd build-aws-c-common
cmake -DCMAKE_INSTALL_PREFIX=%cd%../../../../install -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=ON -DCMAKE_TOOLCHAIN_FILE=%cd%../../../../vcpkg/scripts/buildsystems/vcpkg.cmake -G Ninja ..
aws-c-common
cmake --build . & & cmake --build . --target install & & cd ..
```

3. **Install the AWS Encryption SDK.**

Change to your preferred directory. Then, run the following commands to install and build the AWS Encryption SDK for C.

```
git clone https://github.com/aws/aws-encryption-sdk-c.git
mkdir build-aws-encryption-sdk-c & & cd build-aws-encryption-sdk-c
cmake -DCMAKE_INSTALL_PREFIX=%cd%../../../../install -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=ON -DCMAKE_TOOLCHAIN_FILE=%cd%../../../../vcpkg/scripts/buildsystems/vcpkg.cmake -G Ninja ..
aws-encryption-sdk-c
cmake --build . & & cmake --build . --target install & & cd ..
```
Compile and build

After you install the SDK, you can start building and using it. This guide includes topics that help you to understand keyrings (p. 12), choose a keyring (p. 12), learn the basic programming patterns (p. 35), and work through a simple example (p. 38).

For help compiling and building your applications, see the README file in the aws-encryption-sdk-c repository.

Using the AWS Encryption SDK for C

This topic explains some of the features of the AWS Encryption SDK for C that are not supported in other programming language implementations.

For details about programming with the AWS Encryption SDK for C, see the C examples (p. 38), the examples in the aws-encryption-sdk-c repository on GitHub, and the AWS Encryption SDK for C API documentation.

See also: Using keyrings (p. 12)

Topics

• Patterns for encrypting and decrypting data (p. 35)
• Reference counting (p. 37)

Patterns for encrypting and decrypting data

When you use the AWS Encryption SDK for C, you follow a pattern similar to this: create a keyring (p. 8), create a CMM (p. 7) that uses the keyring, create a session that uses the CMM (and keyring), and then process the session.

1. Create a keyring.

Configure your keyring (p. 8) with the wrapping keys that you want to use to encrypt your data keys. This example uses an AWS KMS keyring (p. 14) with one AWS KMS customer master key (CMK), but you can use any type of keyring in its place.

To identify an AWS KMS CMK in an encryption keyring, specify a key ARN or alias ARN. In a decryption keyring, you must use a key ARN. For details, see Identifying CMKs in an AWS KMS keyring (p. 15).

```c
const char * KEY_ARN = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
struct aws_cryptosdk_keyring *kms_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(KEY_ARN);
```

2. Create a session.

In the AWS Encryption SDK for C, you use a session to encrypt a single plaintext message or decrypt a single ciphertext message, regardless of its size. The session maintains the state of the message throughout its processing.

Configure your session with an allocator, a keyring, and a mode: AWS_CRYPTOSDK_ENCRYPT or AWS_CRYPTOSDK_DECRYPT. If you need to change the mode of the session, use the aws_cryptosdk_session_reset method.

When you create a session with a keyring, the AWS Encryption SDK for C automatically creates a default cryptographic materials manager (CMM) for you. You don't need to create, maintain, or destroy this object.
For example, the following session uses the allocator and the keyring that was defined in step 1. When you encrypt data, the mode is `AWS_CRYPTOSDK_ENCRYPT`.

```c
struct aws_cryptosdk_session * session =
    aws_cryptosdk_session_new_from_keyring(allocator, AWS_CRYPTOSDK_ENCRYPT, kms_keyring);
```

3. Set the message size.

When encrypting data, you need to call the `aws_cryptosdk_session_set_message_size` method, which tells the AWS Encryption SDK the length of the plaintext data. You can call this method before or during the session processing, but you must call it once. Otherwise, the session processing method doesn't know where the input data ends.

Don't call this method when decrypting. The AWS Encryption SDK uses the footer in the encrypted message (p. 133) to determine where the ciphertext ends.

```c
const char *plaintext = "Hello world!";
const size_t plaintext_len = strlen(plaintext);
aws_cryptosdk_session_set_message_size(session, plaintext_length);
```

4. Encrypt or decrypt the data.

To process the data in the session, use the `aws_cryptosdk_session_process` method. If the input buffer is large enough to hold the entire plaintext, and the output buffer is large enough to hold the entire ciphertext, you need to call the method only once. However, if you need to handle streaming data, you can call the method in a loop. For an example, see the `file_streaming.cpp` example.

When the session is configured to encrypt data, the plaintext fields describe the input and the ciphertext fields describe the output. The plaintext field holds the message that you want to encrypt and the ciphertext field gets the encrypted message (p. 133) that the encrypt method returns.

```c
// Encrypting data
aws_cryptosdk_session_process(session,
    ciphertext, ciphertext_buffer_size, ciphertext_length,
    plaintext, plaintext_length, &plaintext_consumed)
```

When the session is configured to decrypt data, the ciphertext fields describe the input and the plaintext fields describe the output. The ciphertext field holds the encrypted message (p. 133) that the encrypt method returned, and the plaintext field gets the plaintext message that the decrypt method returns.

To decrypt the data, call the `aws_cryptosdk_session_process` method.

```c
// Decrypting data
aws_cryptosdk_session_process(session,
    plaintext, plaintext_buffer_size, plaintext_length,
    ciphertext, ciphertext_length, ciphertext_length,
    &ciphertext_consumed)
```
Reference counting

To prevent memory leaks, be sure to release your references to all objects that you create when you are finished with them. Otherwise, you end up with memory leaks. The SDK provides methods to make this task easier.

Whenever you create a parent object with one of the following child objects, the parent object gets and maintains a reference to the child object, as follows:

- A keyring (p. 8), such as creating a session with a keyring
- A default cryptographic materials manager (p. 7) (CMM), such as creating a session or custom CMM with a default CMM
- A data key cache (p. 97), such as creating a caching CMM with a keyring and cache

Unless you need an independent reference to the child objects that you create explicitly. You are not responsible for managing references to any objects that the SDK creates for you. If the SDK creates an object, such as the default CMM that the `aws_cryptosdk_caching_cmm_new_from_keyring` method adds to a session, the SDK manages the creation and destruction of the object and its references.

In the following example, when you create a session with a keyring (p. 8), the session gets a reference to the keyring, and maintains that reference until the session is destroyed. If you do not need to maintain an additional reference to the keyring, you can use the `aws_cryptosdk_keyring_release` method to release the keyring object as soon as the session is created. This method decrements the reference count for the keyring. The session's reference to the keyring is released when you call `aws_cryptosdk_session_destroy` to destroy the session.

```c
// The session gets a reference to the keyring.
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring(alloc, AWS_CRYPTOSDK_ENCRYPT, keyring);

// After you create a session with a keyring, release the reference to the keyring object.
aws_cryptosdk_keyring_release(keyring);
```

For more complex tasks, such as reusing a keyring for multiple sessions or specifying an algorithm suite in a CMM, you might need to maintain an independent reference to the object. If so, don't call the release methods immediately. Instead, release your references when you are no longer using the objects, in addition to destroying the session.

This reference counting technique also works when you are using alternate CMMs, such as the caching CMM for data key caching (p. 97). When you create a caching CMM from a cache and a keyring, the caching CMM gets a reference to both objects. Unless you need them for another task, you can release your independent references to the cache and keyring as soon as the caching CMM is created. Then, when you create a session with the caching CMM, you can release your reference to the caching CMM.

Notice that you are only responsible for releasing references to objects that you create explicitly. Objects that the methods create for you, such as the default CMM that underlies the caching CMM, are managed by the method.

```c
// Create the caching CMM from a cache and a keyring.
struct aws_cryptosdk_cmm *caching_cmm =
    aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL, 60,
    AWS_TIMESTAMP_SECS);
```
// Release your references to the cache and the keyring.
aws_cryptosdk_materials_cache_release(cache);
aws_cryptosdk_keyring_release(kms_keyring);

// Create a session with the caching CMM.
struct aws_cryptosdk_session *session = aws_cryptosdk_session_new_from_cmm(allocator,
AWS_CRYPTOSDK_ENCRYPT, caching_cmm);

// Release your references to the caching CMM.
aws_cryptosdk_cmm_release(caching_cmm);

// ...
aws_cryptosdk_session_destroy(session);

AWS Encryption SDK for C examples

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

When you install and build the AWS Encryption SDK for C, the source code for these and other examples are included in the examples subdirectory, and they are compiled and built into the build directory. You can also find them in the examples subdirectory of the aws-encryption-sdk-c repository on GitHub.

Topics
- Encrypting and decrypting strings (p. 38)

Encrypting and decrypting strings

The following example shows you how to use the AWS Encryption SDK for C to encrypt and decrypt a string.

This example features the KMS keyring (p. 8), a type of keyring that uses an AWS Key Management Service (AWS KMS) customer master key (CMK) to generate and encrypt data keys. The example includes some code written in C++ because the AWS Encryption SDK for C uses the AWS SDK for C++ to call AWS KMS.

For help creating a CMK, see Creating Keys in the AWS Key Management Service Developer Guide. For help identifying CMKs in an AWS KMS keyring, see Identifying CMKs in an AWS KMS keyring (p. 15)

See the complete code sample: string.cpp

Topics
- Encrypt a string (p. 38)
- Decrypt a string (p. 41)

Encrypt a string

The first part of this example uses an AWS KMS keyring with one CMK to encrypt a plaintext string.

Step 1: Construct the keyring.

Create an AWS KMS keyring for encryption. The keyring in this example is configured with one CMK, but you can configure an AWS KMS keyring with multiple CMKs, including CMKs in different AWS Regions and different accounts.
To identify an AWS KMS CMK in an encryption keyring, specify a key ARN or alias ARN. In a decryption keyring, you must use a key ARN. For details, see Identifying CMKs in an AWS KMS keyring (p. 15).

Identifying CMKs in an AWS KMS keyring (p. 15)

When you create a keyring with multiple CMKs, you specify the CMK that is used to generate and encrypt the plaintext data key, and an optional array of additional CMKs that encrypt the same plaintext data key. In this case, we specify only the generator CMK.

Before running this code, replace the example key ARN with a valid one.

```c
const char * key_arn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
struct aws_cryptosdk_keyring *kms_keyring =
  Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

Step 2: Create a session.

Create a session using the allocator, a mode enumerator, and the keyring.

Every session requires a mode: either AWS_CRYPTOSDK_ENCRYPT to encrypt or AWS_CRYPTOSDK_DECRYPT to decrypt. To change the mode of an existing session, use the aws_cryptosdk_session_reset method.

After you create a session with the keyring, you can release your reference to the keyring using the method that the SDK provides. The session retains a reference to the keyring object during its lifetime. References to the keyring and session objects are released when you destroy the session. This reference counting (p. 37) technique helps to prevent memory leaks and to prevent the objects from being released while they are in use.

```c
struct aws_cryptosdk_session *session =
  aws_cryptosdk_session_new_from_keyring(alloc, AWS_CRYPTOSDK_ENCRYPT,
    kms_keyring);

  // When you add the keyring to the session, release the keyring object
  aws_cryptosdk_keyring_release(kms_keyring);
```

Step 3: Set the size of the plaintext data.

Use the aws_cryptosdk_session_set_message_size method to tell the SDK the exact size of the plaintext string. This method call isn't necessary when decrypting, because the SDK gets the ciphertext length from the encrypted message (p. 10).

```c
const char *plaintext_input = "Hello world!";
const size_t plaintext_len_input = strlen(plaintext_input);
aws_cryptosdk_session_set_message_size(session, plaintext_len_input)
```

Step 4: Set the encryption context.

An encryption context (p. 9) is arbitrary, non-secret additional authenticated data. When you provide an encryption context on encrypt, the AWS Encryption SDK cryptographically binds the encryption context to the ciphertext so that the same encryption context is required to decrypt the data. Using an encryption context is optional, but we recommend it as a best practice.

First, create a hash table that includes the encryption context strings.

```c
// Allocate a hash table for the encryption context.
```
int set_up_enc_ctx(struct aws_allocator *alloc, struct aws_hash_table *my_enc_ctx)

    // Create encryption context strings
    AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key1, "Example");
    AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value1, "String");
    AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key2, "Company");
    AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value2, "MyCryptoCorp");

    // Put the key-value pairs in the hash table
    aws_hash_table_put(my_enc_ctx, enc_ctx_key1, (void *)enc_ctx_value1, &was_created)
    aws_hash_table_put(my_enc_ctx, enc_ctx_key2, (void *)enc_ctx_value2, &was_created)

Get a mutable pointer to the encryption context in the session. Then, use the
aws_cryptosdk_enc_ctx_clone function to copy the encryption context into the session. Keep
the copy in my_enc_ctx so you can validate the value after decrypting the data.

    The encryption context is part of the session, not a parameter passed to the session process
    function. This guarantees that the same encryption context is used for every segment of a message,
    even if the session process function is called multiple times to encrypt the entire message.

    struct awa_hash_table *session_enc_ctx =
        aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);
    aws_cryptosdk_enc_ctx_clone(alloc, session_enc_ctx, my_enc_ctx)

Step 5: Encrypt the string.

To encrypt the plaintext string, use the aws_cryptosdk_session_process method with the
session in encryption mode.

    When encrypting, the plaintext fields are input fields; the ciphertext fields are output fields. When
    the processing is complete, the ciphertext_output field contains the encrypted message (p. 10),
    including the actual ciphertext, encrypted data keys, and the encryption context. You can decrypt
    this encrypted message by using the AWS Encryption SDK for any supported programming
    language.

    // Gets the length of the plaintext that the session processed
    size_t plaintext_consumed_output;
    aws_cryptosdk_session_process(session,
        ciphertext_output,
        ciphertext_buf_sz_output,
        ciphertext_len_output,
        plaintext_input,
        plaintext_len_input,
        &plaintext_consumed_output)

Step 6: Clean up the session.

The final steps verify that the session processing is complete and that all of the plaintext is
processed. Then, they destroy the session, including the references to the CMM and the keyring.

    If you prefer, instead of destroying the session, you can reuse the session with the same keyring
    and CMM to decrypt the string, or to encrypt or decrypt other messages. To use the session
    for decrypting, use the aws_cryptosdk_session_reset method to change the mode to
    AWS_CRYPTOSDK_DECRYPT.

    if (!aws_cryptosdk_session_is_done(session)) {
        aws_cryptosdk_session_destroy(session);
        return 8;
    }
if (plaintext_consumed != plaintext_len) abort();
aws_cryptosdk_session_destroy(session);

**Decrypt a string**

The second part of this example decrypts an encrypted message that contains the ciphertext of the original string.

**Step 1: Construct the keyring.**

When you decrypt data in AWS KMS, you pass in the **encrypted message** (p. 10) that the encrypt API returned. The **Decrypt API** doesn't take a CMK as input. Instead, AWS KMS uses the same CMK to decrypt the ciphertext that it used to encrypt it. However, the AWS Encryption SDK lets you specify an AWS KMS keyring with CMKs on encrypt and decrypt.

On decrypt, you can configure a keyring with only the CMKs that you want to use to decrypt the encrypted message. For example, you might want to create a keyring with only the CMK that is used by a particular role in your organization. The AWS Encryption SDK will never use a CMK unless it appears in the decryption keyring. If the SDK can't decrypt the encrypted data keys by using the CMKs in the keyring that you provide, either because none of CMKs in the keyring were used to encrypt any of the data keys, or because the caller doesn't have permission to use the CMKs in the keyring to decrypt, the decrypt call fails.

When you specify an AWS KMS CMK for a decryption keyring, you must use its key ARN. **Alias ARNs** are permitted only in encryption keyrings. For help identifying CMKs in an AWS KMS keyring, see **Identifying CMKs in an AWS KMS keyring** (p. 15).

In this example, we specify a keyring that is configured with the same CMK that was used to encrypt the string. Before running this code, replace the example key ARN with a valid one.

```c
const char * key_arn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
struct aws_cryptosdk_keyring *kms_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

**Step 2: Create a session.**

Create a session using the allocator and the keyring. To configure the session for decryption, configure the session with the **AWS_CRYPTOSDK_DECRYPT** mode.

After you create a session with a keyring, you can release your reference to the keyring using the method that the SDK provides. The session retains a reference to the keyring object during its lifetime, and both the session and keyring are released when you destroy the session. This reference counting technique helps to prevent memory leaks and to prevent the objects from being released while they are in use.

```c
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring(alloc, AWS_CRYPTOSDK_DECRYPT, kms_keyring);

// When you add the keyring to the session, release the keyring object
aws_cryptosdk_keyring_release(kms_keyring);
```

**Step 3: Decrypt the string.**

To decrypt the string, use the **aws_cryptosdk_session_process** method with the session that is configured for decryption.
When decrypting, the ciphertext fields are input fields and the plaintext fields are output fields. The ciphertext_input field holds the encrypted message (p. 133) that the encrypt method returned. When the processing is complete, the plaintext_output field contains the plaintext (decrypted) string.

```c
size_t ciphertext_consumed_output;
aws_cryptosdk_session_process(session,
    plaintext_output,
    plaintext_buf_sz_output,
    plaintext_len_output,
    ciphertext_input,
    ciphertext_len_input,
    &ciphertext_consumed_output)
```

Step 4: Confirm that the process is complete.

After decrypting, the code runs aws_cryptosdk_session_is_done, which confirms that the session processing is complete and that the signature has been verified. If it returns FALSE, the code destroys the session to prevent memory leaks. Also, if any ciphertext remains in the buffer, the example stops.

Be sure that the session is done, especially on decrypt. The entire plaintext might be returned before the signature is verified.

```c
if (!aws_cryptosdk_session_is_done(session)) {
    aws_cryptosdk_session_destroy(session);
    return 13;
}
if (ciphertext_consumed != ciphertext_len) abort();
```

Step 5: Verify the encryption context.

Be sure that the actual encryption context — the one that was used to decrypt the message — contains the encryption context that you provided when encrypting the message. The actual encryption context might include extra pairs, because the cryptographic materials manager (p. 7) (CMM) can add pairs to the provided encryption context before encrypting the message.

In the AWS Encryption SDK for C, you are not required to provide an encryption context when decrypting because the encryption context is included in the encrypted message that the SDK returns. But, before it returns the plaintext message, your decrypt function should verify that all pairs in the provided encryption context appear in the encryption context that was used to decrypt the message.

First, get a read-only pointer to the hash table in the session. This hash table contains the encryption context that was used to decrypt the message.

```c
const struct aws_hash_table *session_enc_ctx =
    aws_cryptosdk_session_get_enc_ctx_ptr(session);
```

Then, loop through the encryption context in the my_enc_ctx hash table that you copied when encrypting. Verify that each pair in the my_enc_ctx hash table that was used to encrypt appears in the session_enc_ctx hash table that was used to decrypt. If any key is missing, or that key has a different value, stop processing and write an error message.

```c
for (struct aws_hash_iter iter = aws_hash_iter_begin(my_enc_ctx); !
    aws_hash_iter_done(&iter);
```
aws_hash_iter_next(&iter)) {
    struct aws_hash_element *session_enc_ctx_kv_pair;
    aws_hash_table_find(session_enc_ctx, iter.element.key, &session_enc_ctx_kv_pair)

    if (!session_enc_ctx_kv_pair ||
        !aws_string_eq((struct aws_string *)iter.element.value, (struct aws_string *)session_enc_ctx_kv_pair->value)) {
        fprintf(stderr, "Wrong encryption context!\n");
        abort();
    }
}

Clean up the session.

After you verify the encryption context, you can destroy the session. As usual, you can reuse the session. If you need to reconfigure it, use the aws_cryptosdk_session_reset method.

aws_cryptosdk_session_destroy(session);

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 AWS Encryption SDK for Java, see the aws-encryption-sdk-java repository on GitHub. For API documentation, see the Javadoc for the AWS Encryption SDK for Java.

Topics
- Prerequisites (p. 43)
- Installation (p. 44)
- AWS Encryption SDK for Java example code (p. 44)

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

The AWS Encryption SDK for Java requires Bouncy Castle.

- AWS Encryption SDK for Java versions 1.6.1 and later use Bouncy Castle to serialize and deserialize cryptographic objects. You can use Bouncy Castle or Bouncy Castle FIPS to satisfy this requirement. For help installing and configuring Bouncy Castle FIPS, see BC FIPS Documentation, especially the User Guides and Security Policy PDFs.

- Earlier versions of the AWS Encryption SDK for Java use Bouncy Castle's cryptography API for Java. This requirement is satisfied only by non-FIPS Bouncy Castle.
If you don't have Bouncy Castle, go to Bouncy Castle latest releases to download the provider file that corresponds to your JDK. You can also use Apache Maven to get the artifact for the standard Bouncy Castle provider (bcprov-ext-jdk15on) or the artifact for Bouncy Castle FIPS (bc-fips).

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. 44) 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.

```xml
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>1.6.1</version>
</dependency>
```

After you install the SDK, get started by looking at the example Java code (p. 44) 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. 44)
- Encrypting and decrypting byte streams (p. 46)
- Encrypting and decrypting byte streams with multiple master key providers (p. 48)

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. For help identifying CMKs in an AWS KMS keyring, see Identifying CMKs in an AWS KMS keyring (p. 15)

When you call encryptString, the AWS Encryption SDK returns the encrypted message (p. 10). This includes the ciphertext, the encrypted data keys, and the encryption context, if you use it. When you call
getResult on the returned object, the AWS Encryption SDK returns a base-64-encoded string version of the encrypted message (p. 133).

Similarly, when you call decryptString in this example, the decryptResult object contains the encrypted message. Before returning the plaintext, verify that the CMK ID and the encryption context in the encrypted message are the ones that you expect.

```java
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;

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 = KmsMasterKeyProvider.builder().withKeysForEncryption(keyArn).build();

        // 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;

/**
 * Encrypts and then decrypts a file under a random key
 * 
 * Arguments:
 * <ol>
 * <li>Name of file containing plaintext data to encrypt
 * </li>
 * 
 * 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 too 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.
        FileInputStream in2 = new FileInputStream(srcFile + ".encrypted");
        CryptoInputStream<JceMasterKey> decryptingStream = crypto.createDecryptingStream(masterKey, in);
        // Does it contain the expected encryption context?
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.
import com.amazonaws.encryptionsdk.multi.MultipleProviderFactory;
import com.amazonaws.util.IOUtils;

/**
 * Encrypts a file using both KMS and an asymmetric key pair.
 * Arguments:
 * <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
 */

* 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 = KmsMasterKeyProvider.builder().withKeysForEncryption(kmsArn).build();
    }
Example code

```java
// 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 = KmsMasterKeyProvider.builder().withKeysForEncryption(kmsArn).build();

// 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 Developer Guide
JavaScript

AWS Encryption SDK for JavaScript

The AWS Encryption SDK for JavaScript is designed to provide a client-side encryption library for developers who are writing web browser applications in JavaScript or web server applications in Node.js.

Like all implementations of the AWS Encryption SDK, the AWS Encryption SDK for JavaScript offers advanced data protection features. These include envelope encryption, additional authenticated data (AAD), and secure, authenticated, symmetric key algorithm suites (p. 8), such as 256-bit AES-GCM with key derivation and signing.

All language-specific implementations of the AWS Encryption SDK are designed to be interoperable, subject to the constraints of the language. For details about language constraints for JavaScript, see the section called “Compatibility” (p. 52).

Learn More
• For details about programming with the AWS Encryption SDK for JavaScript, see the aws-encryption-sdk-javascript repository on GitHub.
• For programming examples, see the section called “Examples” (p. 55) and the example-browser and example-node modules in the aws-encryption-sdk-javascript repository.
• For a real-world example of using the AWS Encryption SDK for JavaScript to encrypt data in a web application, see How to enable encryption in a browser with the AWS Encryption SDK for JavaScript and Node.js in the AWS Security Blog.

Topics
Compatibility of the AWS Encryption SDK for JavaScript

The AWS Encryption SDK for JavaScript is designed to be interoperable with other language implementations of the AWS Encryption SDK. In most cases, you can encrypt data with the AWS Encryption SDK for JavaScript and decrypt it with any other language implementation, including the AWS Encryption SDK Command Line Interface (p. 67). And you can use the AWS Encryption SDK for JavaScript to decrypt encrypted messages (p. 10) produced by other language implementations of the AWS Encryption SDK.

However, when you use the AWS Encryption SDK for JavaScript, you need to be aware of some compatibility issues in the JavaScript language implementation and in web browsers.

Also, when using different language implementations, be sure to configure compatible master key providers, master keys, and keyrings. For details, see Keyring compatibility (p. 14).

AWS Encryption SDK for JavaScript compatibility

The JavaScript implementation of the AWS Encryption SDK differs from other language implementations in the following ways:

- The encrypt operation of the AWS Encryption SDK for JavaScript doesn't return nonframed ciphertext. However, the AWS Encryption SDK for JavaScript will decrypt framed and nonframed ciphertext returned by other language implementations of the AWS Encryption SDK.
- Beginning in Node.js version 12.9.0, Node.js supports the following RSA key wrapping options:
  - OAEP with SHA1, SHA256, SHA384, or SHA512
  - OAEP with SHA1 and MGF1 with SHA1
  - PKCS1v15
- Before version 12.9.0, Node.js supports only the following RSA key wrapping options:
  - OAEP with SHA1 and MGF1 with SHA1
  - PKCS1v15

Browser compatibility

Some web browsers don't support basic cryptographic operations that the AWS Encryption SDK for JavaScript requires. You can compensate for some of the missing operations by configuring a fallback for the WebCrypto API that the browser implements.

Web browser limitations

The following limitations are common to all web browsers:

- The WebCrypto API doesn't support PKCS1v15 key wrapping.
- Browsers don't support 192-bit keys.

Required cryptographic operations
The AWS Encryption SDK for JavaScript requires the following operations in web browsers. If a browser doesn't support these operations, it's incompatible with the AWS Encryption SDK for JavaScript.

- The browser must include `crypto.getRandomValues()`, which is a method for generating cryptographically random values. For information about the web browser versions that support `crypto.getRandomValues()`, see Can I Use `crypto.getRandomValues()`?

**Required fallback**

The AWS Encryption SDK for JavaScript requires the following libraries and operations in web browsers. If you support a web browser that doesn't fulfill these requirements, you must configure a fallback. Otherwise, attempts to use the AWS Encryption SDK for JavaScript with the browser will fail.

- The WebCrypto API, which performs basic cryptographic operations in web applications, isn't available for all browsers. For information about the web browser versions that support web cryptography, see Can I Use Web Cryptography?.
- Modern versions of the Safari web browser don't support AES-GCM encryption of zero bytes, which the AWS Encryption SDK requires. If the browser implements the WebCrypto API, but can't use AES-GCM to encrypt zero bytes, the AWS Encryption SDK for JavaScript uses the fallback library only for zero-byte encryption. It uses the WebCrypto API for all other operations.

To configure a fallback for either limitation, add the following statements to your code. In the `configureFallback` function, specify a library that supports the missing features. The following example uses the Microsoft Research JavaScript Cryptography Library (`msrcrypto`), but you can replace it with a compatible library.

```javascript
import { configureFallback } from '@aws-crypto/client-browser
configureFallback(msrcrypto)
```

**Installing the AWS Encryption SDK for JavaScript**

The AWS Encryption SDK for JavaScript consists of a collection of interdependent modules. Several of the modules are just collections of modules that are designed to work together. Some modules are designed to work independently. A few modules are required for all implementations; a few others are required only for special cases. For information about the modules in the AWS Encryption SDK for JavaScript, see Modules in the AWS Encryption SDK for JavaScript (p. 54) and the `Readme.md` file in each of the modules in the `aws-encryption-sdk-javascript` repository on GitHub.

To install the modules, use the `npm` package manager.

For example, to install the `client-node` module, which includes all of the modules you need to program with the AWS Encryption SDK for JavaScript in Node.js, use the following command.

```bash
npm install @aws-crypto/client-node
```

To install the `client-browser` module, which includes all of the modules you need to program with the AWS Encryption SDK for JavaScript in the browser, use the following command.

```bash
npm install @aws-crypto/client-browser
```

For working examples of how to use the AWS Encryption SDK for JavaScript, see the examples in the `example-node` and `example-browser` modules in the `aws-encryption-sdk-javascript` repository on GitHub.
Modules in the AWS Encryption SDK for JavaScript

The modules in the AWS Encryption SDK for JavaScript make it easy to install the code that you need for your projects.

Modules for JavaScript Node.js

**client-node**

Includes all of the modules you need to program with the AWS Encryption SDK for JavaScript in Node.js.

**caching-materials-manager-node**

Exports functions that support the data key caching (p. 97) feature in the AWS Encryption SDK for JavaScript in Node.js.

**decrypt-node**

Exports functions that decrypt and verify encrypted messages representing data and data streams. Included in the `client-node` module.

**encrypt-node**

Exports functions that encrypt and sign different types of data. Included in the `client-node` module.

**example-node**

Exports working examples of programming with the AWS Encryption SDK for JavaScript in Node.js. Includes example of different types of keyrings and different types of data.

**hkdf-node**

Exports an HMAC-based Key Derivation Function (HKDF) that the AWS Encryption SDK for JavaScript in Node.js uses in particular algorithm suites. The AWS Encryption SDK for JavaScript in the browser uses the native HKDF function in the WebCrypto API.

**integration-node**

Defines tests that verify that the AWS Encryption SDK for JavaScript in Node.js is compatible with other language implementations of the AWS Encryption SDK.

**kms-keyring-node**

Exports functions that support AWS KMS keyrings in Node.js.

**raw-aes-keyring-node**

Exports functions that support Raw AES keyrings (p. 21) in Node.js.

**raw-rsa-keyring-node**

Exports functions that support Raw RSA keyrings (p. 21) in Node.js.

Modules for JavaScript Browser

**client-browser**

Includes all of the modules you need to program with the AWS Encryption SDK for JavaScript in the browser.

**caching-materials-manager-browser**

Exports functions that support the data key caching (p. 97) feature for JavaScript in the browser.
**decrypt-browser**
Exports functions that decrypt and verify encrypted messages representing data and data streams.

**encrypt-browser**
Exports functions that encrypt and sign different types of data.

**example-browser**
Working examples of programming with the AWS Encryption SDK for JavaScript in the browser. Includes examples of different types of keyrings and different types of data.

**integration-browser**
Defines tests that verify that the AWS Encryption SDK for JavaScript in the browser is compatible with other language implementations of the AWS Encryption SDK.

**kms-keyring-browser**
Exports functions that support AWS KMS keyrings (p. 14) in the browser.

**raw-aes-keyring-browser**
Exports functions that support Raw AES keyrings (p. 21) in the browser.

**raw-rsa-keyring-browser**
Exports functions that support Raw RSA keyrings (p. 21) in the browser.

**Modules for all implementations**

**cache-material**
Supports the data key caching (p. 97) feature. Provides code for assembling the cryptographic materials that are cached with each data key.

**kms-keyring**
Exports functions that support KMS keyrings (p. 14).

**material-management**
Implements the cryptographic materials manager (p. 7) (CMM).

**raw-keyring**
Exports functions required for raw AES and RSA keyrings.

**serialize**
Exports functions that the SDK uses to serialize its output.

**web-crypto-backend**
Exports functions that use the WebCrypto API in the AWS Encryption SDK for JavaScript in the browser.

**AWS Encryption SDK for JavaScript examples**

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

You can find more examples of using the AWS Encryption SDK for JavaScript in the example-node and example-browser modules in the aws-encryption-sdk-javascript repository on GitHub. These example modules are not installed when you install the client-browser or client-node modules.
Encrypting data with an AWS KMS keyring

The following example shows you how to use the AWS Encryption SDK for JavaScript to encrypt and decrypt a short string or byte array.

This example features an AWS KMS keyring (p. 14), a type of keyring that uses an AWS Key Management Service (AWS KMS) customer master key (CMK) to generate and encrypt data keys. For help creating a CMK, see Creating Keys in the AWS Key Management Service Developer Guide. For help identifying CMKs in an AWS KMS keyring, see Identifying CMKs in an AWS KMS keyring (p. 15)

Step 1: Construct the keyring.

Create an AWS KMS keyring for encryption.

When encrypting with an AWS KMS keyring, you must specify a generator key, that is, an AWS KMS CMK that is used to generate the plaintext data key and encrypt it. You can also specify zero or more additional keys that encrypt the same plaintext data key. The keyring returns the plaintext data key and one encrypted copy of that data key for each CMK in the keyring, including the generator key. To decrypt the data, you need to decrypt any one of the encrypted data keys.

To specify the CMKs for an encryption keyring in the AWS Encryption SDK for JavaScript, you can use any supported AWS KMS key identifier (p. 15). This example uses a generator key, which is identified by its alias ARN, and one additional key, which is identified by a key ARN.

Note
If you plan to reuse your AWS KMS keyring for decrypting, you must use key ARNs to identify the CMKs in the keyring.

Before running this code, replace the example CMK identifiers with valid identifiers. You must have the permissions required to use the CMKs (p. 15) in the keyring.

JavaScript Browser

Begin by providing your credentials to the browser. The AWS Encryption SDK for JavaScript examples use the webpack.DefinePlugin, which replaces the credential constants with your actual credentials. But you can use any method to provide your credentials. Then, use the credentials to create an AWS KMS client.

```javascript
declare const credentials: {accessKeyId: string, secretAccessKey:string, sessionToken:string }

const clientProvider = getClient(KMS, { 
 credentials: { 
 accessKeyId, 
 secretAccessKey, 
 sessionToken
 }
 })
```

Next, specify the AWS KMS CMKs for the generator key and additional key. Then, create an AWS KMS keyring using the AWS KMS client and the CMKs.

```javascript
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'
```
Step 2: Set the encryption context.

An encryption context (p. 9) is arbitrary, non-secret additional authenticated data. When you provide an encryption context on encrypt, the AWS Encryption SDK cryptographically binds the encryption context to the ciphertext so that the same encryption context is required to decrypt the data. Using an encryption context is optional, but we recommend it as a best practice.

Create a simple object that includes the encryption context pairs. The key and value in each pair must be a string.

JavaScript Browser

```javascript
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

JavaScript Node.js

```javascript
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

Step 3: Encrypt the data.

To encrypt the plaintext data, call the encrypt function. Pass in the AWS KMS keyring, the plaintext data, and the encryption context.

The encrypt function returns an encrypted message (p. 10) (result) that contains the encrypted data, the encrypted data keys, and important metadata, including the encryption context and signature.

You can decrypt this encrypted message (p. 58) by using the AWS Encryption SDK for any supported programming language.

JavaScript Browser

```javascript
const plaintext = new Uint8Array([1, 2, 3, 4, 5])
const { result } = await encrypt(keyring, plaintext, { encryptionContext: context })
```

JavaScript Node.js

```javascript
const plaintext = 'asdf'
```
Decryption data with an AWS KMS keyring

You can use the AWS Encryption SDK for JavaScript to decrypt the encrypted message and recover the original data.

In this example, we decrypt the data that we encrypted in the section called “Encrypting data with an AWS KMS keyring” (p. 56) example.

Step 1: Construct the keyring.

To decrypt the data, pass in the encrypted message (p. 10) (result) that the encrypt function returned. The encrypted message includes the encrypted data, the encrypted data keys, and important metadata, including the encryption context and signature.

You must also specify an AWS KMS keyring (p. 14) when decrypting. You can use the same keyring that was used to encrypt the data or a different keyring. To succeed, at least one CMK in the decryption keyring must be able to decrypt one of the encrypted data keys in the encrypted message. Because no data keys are generated, you do not need to specify a generator key in a decryption keyring. If you do, the generator key and additional keys are treated the same way.

To specify a CMK for a decryption keyring in the AWS Encryption SDK for JavaScript, you must use the key ARN. Otherwise, the CMK is not recognized. For help identifying CMKs in an AWS KMS keyring, see Identifying CMKs in an AWS KMS keyring (p. 15)

Note
If you use the same keyring for encrypting and decrypting, use key ARNs to identify the CMKs in the keyring.

In this example, we create a keyring that includes only one of the CMKs in the encryption keyring. Before running this code, replace the example key ARN with a valid one. You must have kms:Decrypt permission on the CMK.

JavaScript Browser

Begin by providing your credentials to the browser. The AWS Encryption SDK for JavaScript examples use the webpack.DefinePlugin, which replaces the credential constants with your actual credentials. But you can use any method to provide your credentials. Then, use the credentials to create an AWS KMS client.

```javascript
const credentials = { accessKeyId: string, secretAccessKey: string, sessionToken: string }

const clientProvider = getClient(KMS, {
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken
  }
})
```

Next, create an AWS KMS keyring using the AWS KMS client. This example uses just one of the CMKs from the encryption keyring.

```javascript
const keyIds = ['arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
```
const keyring = new KmsKeyringBrowser({ clientProvider, keyIds })

JavaScript Node.js

const keyIds = ['arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
const keyring = new KmsKeyringNode({ keyIds })

Step 2: Decrypt the data.

Next, call the `decrypt` function. Pass in the decryption keyring that you just created (`keyring`) and the encrypted message (p. 10) that the `encrypt` function returned (`result`). The AWS Encryption SDK uses the keyring to decrypt one of the encrypted data keys. Then it uses the plaintext data key to decrypt the data.

If the call succeeds, the `plaintext` field contains the plaintext (decrypted) data. The `messageHeader` field contains metadata about the decryption process, including the encryption context that was used to decrypt the data.

JavaScript Browser

```js
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

JavaScript Node.js

```js
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

Step 3: Verify the encryption context.

The encryption context (p. 9) that was used to decrypt the data is included in the message header (`messageHeader`) that the `decrypt` function returns. Before your application returns the plaintext data, verify that the encryption context that you provided when encrypting is included in the encryption context that was used when decrypting. A mismatch might indicate that the data was tampered with, or that you didn’t decrypt the right ciphertext.

When verifying the encryption context, do not require an exact match. When you use an encryption algorithm with signing, the cryptographic materials manager (p. 7) (CMM) adds the public signing key to the encryption context before encrypting the message. But all of the encryption context pairs that you submitted should be included in the encryption context that was returned.

First, get the encryption context from the message header. Then, verify that each key-value pair in the original encryption context (`context`) matches a key-value pair in the returned encryption context (`encryptionContext`).

JavaScript Browser

```js
const { encryptionContext } = messageHeader
Object.entries(context).forEach(([key, value]) => {
  if (encryptionContext[key] !== value) throw new Error('Encryption Context does not match expected values')
})
```

JavaScript Node.js

```js
const { encryptionContext } = messageHeader
```
Object
  .entries(context)
  .forEach(([key, value]) => {
    if (encryptionContext[key] !== value) throw new Error('Encryption Context does not match expected values')
  })

If the encryption context check succeeds, you can return the plaintext data.

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 AWS Encryption SDK for Python, see the aws-encryption-sdk-python repository on GitHub. For API documentation, see Read the Docs.

Topics
- Prerequisites (p. 60)
- Installation (p. 60)
- AWS Encryption SDK for Python example code (p. 61)

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. 61) 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. 61)
- Encrypting and decrypting byte streams (p. 62)
- Encrypting and decrypting byte streams with multiple master key providers (p. 64)
- Using data key caching to encrypt messages (p. 66)

### 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 Amazon Key Management Service (AWS KMS) as the master key.

```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
# https://aws.amazon.com/apache-2-0/
#
from __future__ import print_function
import aws_encryption_sdk

def cycle_string(key_arn, source_plaintext, botocore_session=None):
    
    # 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,
```

----

61
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):
    
    provider_id = 'static-random'

    def __init__(self, **kwargs):
        

62
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).

```python
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 an AWS KMS customer master key (CMK) and an RSA key pair as the master keys.

```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 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.
...```

```python
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,
            ..."""
```python
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.

    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)
    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)
            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=kms_master_key_provider)
            for chunk in decryptor:
                plaintext.write(chunk)
```
Using data key caching to encrypt messages

The following example shows how to use data key caching (p. 97) in the AWS Encryption SDK for Python. It is designed to show you how to configure an instance of the local cache (p. 115) (LocalCryptoMaterialsCache) with the required capacity value and an instance of the caching cryptographic materials manager (p. 115) (caching CMM) with cache security thresholds (p. 111).

This very basic example creates a function that encrypts a fixed string. It lets you specify an AWS KMS customer master key, the required cache size (capacity), and a maximum age value. For a more complex, real-world example of data key caching, see the section called “Python example” (p. 123).

Although it is optional, this example also uses an encryption context (p. 9) as additional authenticated data. When you decrypt data that was encrypted with an encryption context, be sure that your application verifies that the encryption context is the one that you expect before returning the plaintext data to your caller. An encryption context is a best practice element of any encryption or decryption operation, but it plays a special role in data key caching. For details, see Encryption Context: How to Select Cache Entries (p. 116).

# 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...
```
# Example of encryption with 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 to retain in cache at once
    
    # Data to be encrypted
    my_data = "My plaintext data"
    
    # Security thresholds
    # Max messages (or 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 customer master key (CMK)
    key_provider = aws_encryption_sdk.KMSMasterKeyProvider(key_ids=[kms_cmk_arn])
    
    # Create a local 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 call to encrypt data specifies a caching CMM,
    # the encryption operation uses the data key cache specified
    # in the caching CMM
    encrypted_message, _header = aws_encryption_sdk.encrypt(
        source=my_data,
        materials_manager=caching_cmm,
        encryption_context=encryption_context
    )

    return encrypted_message
```

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 (p. 60) 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. 43) 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.

Performance

The AWS Encryption CLI is built on the AWS Encryption SDK for Python. Each time you run the CLI, you start a new instance of the Python runtime. To improve performance, whenever possible, use a single command instead of a series of independent commands. For example, run one command that processes the files in a directory recursively instead of running separate commands for each file.

Topics

- Installing the AWS Encryption SDK command line interface (p. 68)
- How to use the AWS Encryption SDK command line interface (p. 70)
- Examples of the AWS Encryption SDK command line interface (p. 77)
- AWS Encryption SDK CLI syntax and parameter reference (p. 91)

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. 68)
- Installing the AWS Encryption CLI (p. 69)

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 the following.
AWS Encryption SDK Developer Guide

Installing the CLI

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. 8), 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
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. 77). For complete documentation, see Read the Docs.

Topics
- How to encrypt and decrypt data (p. 70)
- How to specify a master key (p. 71)
- How to provide input (p. 73)
- How to specify the output location (p. 73)
- How to use an encryption context (p. 74)
- How to store parameters in a configuration file (p. 76)

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. 7), 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. 10) and for metadata about the encryption operation. An encryption context (p. 9) 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. 10) 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 \\
```
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. 7). You can use an AWS KMS customer master key (CMK) or a master key from a custom master key provider (p. 8). 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. 71) 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. 72) 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. 72) 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 key ID, its key ARN, an alias name, or alias ARN. For example, this encrypt command uses an alias ARN in the value of the `key` attribute. For details about the key identifiers for a AWS KMS CMK, see Key Identifiers in the AWS Key Management Service Developer Guide.

```
```
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. 8). 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. 10) 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.
How to use the CLI

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

--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. 10). 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 output the 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.

  ```
  --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 `--output` 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 pairs that the CMM (p. 7) adds, is cryptographically bound to the encrypted data. It is also included (in plaintext) in the encrypted message (p. 9) 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 an 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 as 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
```

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.
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. 70). For a quick reference, see AWS Encryption SDK CLI syntax and parameter reference (p. 91).

Topics
- Encrypting a file (p. 77)
- Decrypting a file (p. 79)
- Encrypting all files in a directory (p. 80)
- Decrypting all files in a directory (p. 81)
- Encrypting and decrypting on the command line (p. 83)
- Using multiple master keys (p. 84)
- Encrypting and decrypting in scripts (p. 86)
- Using data key caching (p. 89)

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. 7), encrypts the file contents under the data key, and then writes the encrypted message (p. 10) 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. For details about the key identifiers for an AWS KMS customer master key, see Key Identifiers in the AWS Key Management Service Developer Guide.

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. 71), and its required key attribute, tell the command to use the master key represented by the key 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. 74).

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

```bash
\ To run this example, replace the fictitious key 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

```powershell
# To run this example, replace the fictitious key ARN with a valid value.

$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. 10) 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
```
Decrypted 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. 74) 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. 7) 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. 71), and its required `key` attribute, specify the master key. The encrypt command includes an encryption context (p. 74), `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

```bash
# To run this example, replace the fictitious key 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

```powershell
# To run this example, replace the fictitious key 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

```bash
$ ls testenc
```

81
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 "\n  --input C:\TestEnc --recursive " \
  --encryption-context dept=IT " \
  --metadata-output $home\Metadata.txt " \
  --output C:\TestDec --interactive

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

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

  **PowerShell**

  ```powershell
  $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**

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

  **PowerShell**

  ```powershell
  $encrypted = 'Hello World' | aws-encryption-cli --encrypt --master-keys key=$cmkArn -S
  ```
• 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

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

### PowerShell

```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

```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

```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 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**

```bash
$ cmk1=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$ cmk3=arn:aws:kms:ap-southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c64-5e6f1a2b3c4d

$ 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**

```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.

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)]

)
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
                --encryption-context $EncryptionContext --metadata-output $MetadataDirectory --v) 2>&1
            # Check error status
            if ($? -eq $false) {
                ...
            }
        }
    }
}

[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({Get-S3Bucket -BucketName $_})]
[String]
$S3Bucket,

[Parameter()]
[String]
$S3BucketFolder
```bash
# Continue running even if an operation fails.
set +e

dir=$1
dirname=$(dirname $1)
encryptionContext=$2
s3bucket=$3
s3folder=$4
masterKeyProvider="aws-kms"
metadataOutput="/tmp/metadata-$(date +%s)"

clean(){
    gzip -qf $1
}

encrypt(){
    # -e encrypt
    # -i input
    # -o output
    # --metadata-output unique file for metadata
    # -m masterKey read from environment variable
    # -o encryption context read from the second argument.
    # -v be verbose
    aws-encryption-cli -e -i $1 -o $(dirname $1) --metadata-output
    metadataOutput -m key="$(masterKey)" provider="$(masterKeyProvider)" -c
    "$(encryptionContext)" -v
}

s3put (){  
    # copy file argument 1 to s3 location passed into the script.
    aws s3 cp $(dirname $1) s3bucket/$s3folder
}

# Validate all required arguments are present.
if [ "*(dir)*" ] && [ "*(encryptionContext)*" ] && [ "*(s3bucket)*" ] && [ "*(s3folder)*" ]; then
    # Is *dir* a valid directory?
    test -d *dir* || { echo "Error: *dir* is not a directory." >&2; exit 1; }
    # Is *dir* a valid directory?
    if [ $# -ne 0 ]; then
        # set *dir* and encryptionContext* to passed in arguments
        dir=$1
e
        # set encryptionContext and s3bucket* to passed in arguments
        encryptionContext=$2
        s3bucket=$3
        s3folder=$4
        masterKeyProvider="aws-kms"
        metadataOutput="/tmp/metadata-$(date +%s)"
        clean()
        encrypt()
        s3put ()
    fi
fi
```

AWS Encryption SDK Developer Guide
Examples
Using data key caching

This example uses data key caching (p. 97) 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

```bash
# cmkArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$ aws-encryption-cli --encrypt
  --input /var/log/httpd --recursive
  --master-keys key=$cmkArn
  --encryption-context class=log
  --suppress-metadata
  --caching capacity=1 max_age=10 max_messages_encrypted=10
```
PowerShell

```powershell
PS C:\> $cmkArn = arn:aws:kms:us-west-2:11112223333: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        : 0.00029459629861111
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}

max_messages_encrypted=10)

Days             : 0
Hours            : 0
Minutes          : 0
Seconds          : 11
Milliseconds     : 813
Ticks            : 118132640
TotalDays        : 0.000136727592592593
TotalHours       : 0.003281462222222222
TotalMinutes     : 0.196887733333333
TotalSeconds     : 11.813264
```
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}
```

---

**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](#). For examples, see [Examples of the AWS Encryption SDK command line interface](#). For complete documentation, see [Read the Docs](#).

**Topics**

- [AWS Encryption CLI syntax](#)
- [AWS Encryption CLI command line parameters](#)
- [Advanced parameters](#)

**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.
Encrypt data

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

```
aws-encryption-cli --encrypt
   --input <input> [--recursive] [--decode]
   --master-keys [...] [key=<keyID> [provider=<provider-name>]] [region=<aws-region>]
   [profile=<aws-profile>]
   --metadata-output <location> [--overwrite-metadata] [--suppress-metadata]
   [--encryption-context <encryption_context> [...]]
   [--algorithm <algorithm_suite>]
   [--caching <attributes>]
   [--frame-length <length>]
   [-v | -vv | -vvv | --quiet]
```

Decrypt data

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

```
aws-encryption-cli --decrypt
   --input <input> [--recursive] [--decode]
   --metadata-output <location> [--overwrite-metadata] [--suppress-metadata]
   [key=<keyID> [provider=<provider-name>]] [region=<aws-region>]
   [profile=<aws-profile>]
   [--encryption-context <encryption_context> [...]]
   [--algorithm <algorithm_suite>]
   [--caching <attributes>]
   [--max-length <length>]
   [-v | -vv | -vvv | --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. 76).

```
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. 7) 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 key ID, key ARN, an alias name, or an alias ARN. For details about the

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. 8). 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. 74) 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. 8). 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 nonframed 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. 97) 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. 97) 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. 7) and related cryptographic material (p. 116) 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 local cache (p. 115) and a caching cryptographic materials manager (p. 115) (caching CMM) that interacts with the cache and enforces security thresholds (p. 111) 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. 111) 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 use data key caching (p. 97)
- Setting cache security thresholds (p. 111)
- Data key caching details (p. 112)
- Data key caching example (p. 117)

How to use data key caching

This topic shows you how to use 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.

Note

The AWS Encryption SDK for JavaScript is a beta release. The code and the documentation are subject to change.

For complete and tested examples of using data key caching in the AWS Encryption SDK, see:

- C/C++: caching_cmm.cpp
Using 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. 115).** In these examples, we use the local cache that the AWS Encryption SDK provides. We limit the cache to 10 data keys.

```c
// Cache capacity (maximum number of entries) is required
size_t cache_capacity = 10;
struct aws_allocator *allocator = aws_default_allocator();

struct aws_cryptosdk_materials_cache *cache =
    aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);
```

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

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

```javascript Browser
const capacity = 10

const cache = getLocalCryptographicMaterialsCache(capacity)
```

```javascript Node.js
const capacity = 10

const cache = getLocalCryptographicMaterialsCache(capacity)
```

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

cache = LocalCryptoMaterialsCache(MAX_CACHE_SIZE)
```
Create a master key provider (p. 8) (Java and Python) or a keyring (p. 8) (C and JavaScript). These examples use an AWS Key Management Service (AWS KMS) master key provider or a compatible AWS KMS keyring (p. 14).

C

```c
// Create an AWS KMS keyring
//   The input is the Amazon Resource Name (ARN)
//   of an AWS KMS customer master key (CMK)

struct aws_cryptosdk_keyring *kms_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_cmk_arn);
```

Java

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

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

JavaScript Browser

In the browser, you must inject your credentials securely. This example defines credentials in a webpack (kms.webpack.config) that resolves credentials at runtime. It creates an AWS KMS client provider instance from an AWS KMS client and the credentials. Then, when it creates the keyring, it passes the client provider to the constructor along with the AWS KMS customer master key (generatorKeyId).

```javascript
const { accessKeyId, secretAccessKey, sessionToken } = credentials

const clientProvider = getClient(KMS, {
    credentials: {
        accessKeyId,
        secretAccessKey,
        sessionToken
    }
})

/* Create an AWS KMS keyring
 * The input is the Amazon Resource Name (ARN)
 */

const keyring = new KmsKeyringBrowser({ clientProvider, generatorKeyId })
```

JavaScript Node.js

```javascript
/* Create an AWS KMS keyring
 * The input is the Amazon Resource Name (ARN)
 */

const keyring = new KmsKeyringNode({ generatorKeyId })
```

Python

```python
# Create an AWS KMS master key provider
# The input is the Amazon Resource Name (ARN)
# of an AWS KMS customer master key (CMK)
```
Create a caching cryptographic materials manager (p. 115) (caching CMM).

Associate your caching CMM with your cache and your master key provider or keyring. Then, set cache security thresholds (p. 111) on the caching CMM.

In the AWS Encryption SDK for C, you can create a caching CMM from an underlying CMM, such as the default CMM, or from a keyring. This example creates the caching CMM from a keyring.

After you create the caching CMM, you can release your references to the keyring and the cache. For details, see the section called “Reference counting” (p. 37).

```
// Create the caching CMM
//   Set the partition ID to NULL.
//   Set the required maximum age value to 60 seconds.
struct aws_cryptosdk_cmm *caching_cmm =
    aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL, 60, AWS_TIMESTAMP_SECS);

// Add an optional message threshold
//   The cached data key will not be used for more than 10 messages.
aws_status = aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, 10);

// Release your references to the cache and the keyring.
aws_cryptosdk_materials_cache_release(cache);
aws_cryptosdk_keyring_release(kms_keyring);
```

Java

```
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();
```

JavaScript Browser

```
/*
 * Security thresholds
 * Max age (in milliseconds) is required.
 * Max messages (and max bytes) per entry are optional.
 */
```
const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring */
const cachingCmm = new WebCryptoCachingMaterialsManager(
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
)

JavaScript Node.js

const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring */
const cachingCmm = new NodeCachingMaterialsManager(
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
)

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.

C

In the AWS Encryption SDK for C, you create a session with the caching CMM and then process the session.
By default, when the message size is unknown and unbounded, the AWS Encryption SDK does not cache data keys. To allow caching when you don't know the exact data size, use the `aws_cryptosdk_session_set_message_bound` method to set a maximum size for the message. Set the bound larger than the estimated message size. If the actual message size exceeds the bound, the encryption operation fails.

```c
// Create a session with the caching CMM. Set the session mode to encrypt.
struct aws_cryptosdk_session *session = aws_cryptosdk_session_new_from_cmm(allocator,
AWS_CRYPTOSDK_ENCRYPT, caching_cmm);

// Set a message bound of 1000 bytes
aws_status = aws_cryptosdk_session_set_message_bound(session, 1000);

// Encrypt the message using the session with the caching CMM
aws_status = aws_cryptosdk_session_process(
    session, output_buffer, output_capacity, &output_produced, input_buffer,
    input_len, &input_consumed);

// Release your references to the caching CMM and the session.
aws_cryptosdk_cmm_release(caching_cmm);
aws_cryptosdk_session_destroy(session);
```

Java

```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();
```

JavaScript Browser

```javascript
const { result } = await encrypt(cachingCmm, plaintext)
```

JavaScript Node.js

When you use the caching CMM in the AWS Encryption SDK for JavaScript for Node.js, the `encrypt` method requires the length of the plaintext. If you don't provide it, the data key is not cached. If you provide a length, but the plaintext data that you supply exceeds that length, the encrypt operation fails. If you don't know the exact length of the plaintext, such as when you're streaming data, provide the largest expected value.

```javascript
const { result } = await encrypt(cachingCmm, plaintext, { plaintextLength: plaintext.length })
```

Python

```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. 98) into test code that you can run.

The example creates a local cache (p. 115) and a master key provider (p. 8) or keyring (p. 8) for an AWS KMS customer master key (CMK). Then, it uses the local cache and master key provider or keyring to create a caching CMM with appropriate security thresholds (p. 111). In Java and Python, the encryption request specifies the caching CMM, the plaintext data to encrypt, and an encryption context (p. 116). In C, the caching CMM is specified in the session, and the session is provided to the encryption request.

To run these examples, you need to supply the Amazon Resource Name (ARN) of an AWS 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. 119) for Java, Data key caching example in Python (p. 123) for Python, and caching_cmm.cpp for C/C++.

C

```c
#include <aws/cryptosdk/cache.h>
#include <aws/cryptosdk/cpp/kms_keyring.h>
#include <aws/cryptosdk/session.h>

void encrypt_with_caching(
    uint8_t *ciphertext, // output will go here (assumes ciphertext_capacity bytes already allocated)
    size_t *ciphertext_len, // length of output will go here
    size_t ciphertext_capacity,
    const char *kms_cmk_arn,
    int max_entry_age,
    int cache_capacity) {
    const uint64_t MAX_ENTRY_MSGS = 100;
    struct aws_allocator *allocator = aws_default_allocator();

    // Create a keyring
    struct aws_cryptosdk_keyring *kms_keyring =
        Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_cmk_arn);

    // Create a cache
    struct aws_cryptosdk_materials_cache *cache =
        aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);

    // Create a caching CMM
    struct aws_cryptosdk_cmm *caching_cmm =
        aws_cryptosdk_caching_cmm_new_from_keyring(kms_keyring, cache, max_entry_age,
```

103
allocator, cache, kms_keyring, NULL, max_entry_age, AWS_TIMESTAMP_SECS);
if (!caching_cmm) abort();

if (aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, MAX_ENTRY_MSGS)) abort();

// Create a session
struct aws_cryptosdk_session *session =
aws_cryptosdk_session_new_from_cmm(allocator, AWS_CRYPTOSDK_ENCRYPT,
caching_cmm);
if (!session) abort();

// Encryption context
struct aws_hash_table *enc_ctx =
aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);
if (!enc_ctx) abort();
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key, "purpose");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value, "test");
if (aws_hash_table_put(enc_ctx, enc_ctx_key, (void *)enc_ctx_value, NULL)) abort();

// Plaintext data to be encrypted
const char *my_data = "My plaintext data";
size_t my_data_len = strlen(my_data);
if (aws_cryptosdk_session_set_message_size(session, my_data_len)) abort();

// When the session uses a caching CMM, the encryption operation uses the data key cache
// specified in the caching CMM.
size_t bytes_read;
if (aws_cryptosdk_session_process(
    session,
ciphertext,
ciphertext_capacity,
ciphertext_len,
    (const uint8_t *)my_data,
    my_data_len,
    &bytes_read)) abort();
if (!aws_cryptosdk_session_is_done(session) || bytes_read != my_data_len) abort();

aws_cryptosdk_session_destroy(session);
aws_cryptosdk_cmm_release(caching_cmm);
aws_cryptosdk_materials_cache_release(cache);
aws_cryptosdk_keyring_release(kms_keyring);
}

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;

/**
 * Encrypts a string using an AWS KMS customer master key (CMK) and data key caching
 * 
 * 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 {
    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();

JavaScript Browser

/*
 * Copyright 2019 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.
 */

/* This example shows how to use a data key cache when encrypting data
 * with JavaScript in the browser. It uses a KMS keyring, but you can use
 * any valid keyring.
 */

import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  encrypt,
  decrypt,
  WebCryptoCachingMaterialsManager,
  getLocalCryptographicMaterialsCache
} from '@aws-crypto/client-browser'
import { toBase64 } from '@aws-sdk/util-base64-browser'

/* Begin by providing your credentials to the browser in a secure manner. The AWS
Encryption
* SDK for JavaScript examples use the webpack.DefinePlugin, which replaces the
* credential
* constants with your actual credentials.
*/
declare const credentials: {accessKeyId: string, secretAccessKey:string,
  sessionToken:string }

/* Create a client provider that will inject the correct credentials.
* The credentials here are injected by webpack from your environment bundle.
* The credential values are pulled using @aws-sdk/credential-provider-node.
* See kms.webpack.config
*/
const { accessKeyId, secretAccessKey, sessionToken } = credentials

/* getClient takes a KMS client constructor and optional configuration values.
*/
const clientProvider = getClient(KMS, {
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken
  }
}
/* This example uses a KMS keyring. First, it defines an AWS KMS customer master key (CMK) as the generator key. To use this CMK as a generator in an encrypt function, you need kms:GenerateDataKey permission on the CMK.
Before running this code, replace the example CMK ID with a valid one from your AWS account.
*/
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'

/* This example also specifies an additional key, which is optional. You must have kms:Encrypt permission on these CMKs.
The data key that encrypts your data is encrypted by the generator key and by each of the additional keys. The encrypted message that the encrypt function returns contains all of the encrypted data keys. To decrypt the message, you must decrypt any one of the encrypted data keys.
Before running this code, replace the example CMK ID with a valid one from your AWS account.
*/
const keyIds = ['arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']

/* Create the KMS keyring. Pass in the client provider, the required generator key, and any (optional) additional keys.*/
const keyring = new KmsKeyringBrowser({ clientProvider, generatorKeyId, keyIds })

/* Create a cache to hold the data keys and related cryptographic material. In this example, we use the local cache provided by the Encryption SDK.
`capacity` represents the maximum number of data keys in the cache.
When the cache is full, the oldest entry is evicted to make room for a newer one. This value is required.
*/
const capacity = 10

/* There is also a second optional parameter, "proactiveFrequency". By default, every 60 seconds (60,000 milliseconds) all data keys in the cache are checked to verify that they conform to all data key caching thresholds. Data keys that exceed any threshold are evicted.
To change how often this check runs, pass in a second value for the "proactiveFrequency" parameter. Its value is in milliseconds.
This example changes the frequency to 30000 (milliseconds).
*/
const proactiveFrequency = 30000
const cache = getLocalCryptographicMaterialsCache(capacity, proactiveFrequency)

/* Set the security thresholds on the caching CMM: maxAge, maxBytesEncrypted, maxMessagesEncrypted.
Only maxAge is required.*/

/* maxAge is the time in milliseconds that an entry is cached. The cache actively removes entries that have exceeded the thresholds.
*/
const maxAge = 1000 * 60

/* The maximum amount of bytes encrypted under a single data key.*/
This value is optional, but you should configure the lowest practical value.

```javascript
const maxBytesEncrypted = 100
```

The maximum number of messages encrypted under a single data key.

```javascript
const maxMessagesEncrypted = 100
```

Use the keyring, local cache, and security thresholds to create a caching CMM.

```javascript
const cachingCmm = new WebCryptoCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  maxAge,
  maxBytesEncrypted,
  maxMessagesEncrypted
})
```

The encryption context is non-secret data that is cryptographically bound to the ciphertext when it is encrypted. To decrypt the data, the Encryption SDK uses the same encryption context. (It gets the encryption context from the header of the encrypted message; you don't have to supply it.)

In addition, cached data keys are reused only when their encryption contexts match. Therefore, you can use the encryption context to create subgroups of data keys in your cache.


```javascript
const encryptionContext = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

Define the data to encrypt.

```javascript
const plainText = new Uint8Array([1, 2, 3, 4, 5])
```

Encrypt the data.

```javascript
const { result } = await encrypt(cachingCmm, plainText, { encryptionContext })
```
import { KmsKeyringNode, encrypt, decrypt, NodeCachingMaterialsManager, getLocalCryptographicMaterialsCache } from '@aws-crypto/client-node'

export async function cachingMaterialsManagerNodeSimpleTest (plaintext) {

    /* A KMS CMK is required to generate the data key. 
     * You need kms:GenerateDataKey permission on the CMK in generatorKeyId. 
     */
    const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'

    /* Configure the KMS keyring with the desired CMKs. 
     * A generator key is required when encrypting data. You can also add an array of 
     * additional keys. 
     */
    const keyring = new KmsKeyringNode({ generatorKeyId })

    /* Create a cache to hold the data keys and related cryptographic material. 
     * In this example, we use the local cache provided by the Encryption SDK. 
     * 
     * ‘capacity’ represents the maximum number of data keys in the cache. 
     * When the cache is full, the oldest entry is evicted to make room for a 
     * newer one. This value is required. 
     */
    const capacity = 10

    /* There is also a second optional parameter, "proactiveFrequency". 
     * By default, every 60 seconds (60,000 milliseconds) all data keys in the 
     * cache are checked to verify that they conform to all data key caching 
     * thresholds. Data keys that exceed any threshold are evicted. 
     * 
     * To change how often this check runs, pass in a second value for the 
     * "proactiveFrequency" parameter. Its value is in milliseconds. 
     * This example changes the frequency to 30000 (milliseconds). 
     */
    const proactiveFrequency = 30000

    const cache = getLocalCryptographicMaterialsCache(capacity, proactiveFrequency)

    /* Set the security thresholds on the caching CMM: 
     * maxAge, maxBytesEncrypted, maxMessagesEncrypted. 
     * Only maxAge is required. 
     */
    /* maxAge is the time in milliseconds that an entry is cached. 
     * The cache actively removes elements that exceed its thresholds. 
     */
    const maxAge = 1000 * 60

    /* The maximum amount of bytes that will be encrypted under a single data key. 
     * The default value is 2^53 - 1 bytes. This value is optional, but you should 
     * set it to the lowest practical value. 
     */
    const maxBytesEncrypted = 100

    /* The maximum number of messages encrypted under a single data key. 
     * The default value is 2^32 bytes. This value is optional, but you should set 
     * it to the lowest practical value. 
     */
    const maxMessagesEncrypted = 100

    /* Use the keyring, local cache, and security thresholds to create 
     * a caching CMM. 
     */

const cachingCmm = new NodeCachingMaterialsManager(
  backingMaterials: keyring,
  cache,
  maxAge,
  maxBytesEncrypted,
  maxMessagesEncrypted
))

/* The encryption context is non-secret data that is cryptographically bound
 * to the ciphertext when it is encrypted. To decrypt the data, the Encryption SDK
 * uses the same encryption context. (It gets the encryption context from the header
 * of the encrypted message; you don't have to supply it.)
 * In addition, cached data keys are reused only when their encryption contexts
 * match. Therefore, you can use the encryption context to create subgroups of data keys in
 * your cache.
 * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-
 * caching-details.html#caching-encryption-context
 */
const encryptionContext = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}

/* Encrypt the data.
 * When the call to the encrypt function specifies a caching CMM,
 * the encryption operation uses the data key cache.
 * When you use the caching CMM, the encrypt method requires the length
 * of the plaintext. If you don't provide it, the encrypt operation fails.
 * If you don't know the exact length of the plaintext, such as when you're
 * streaming data, provide the largest expected value.
 */
const { result } = await encrypt(cachingCmm, plaintext, { encryptionContext,
  plaintextLength: plaintext.length })

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.

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
Setting cache security thresholds

When you implement data key caching, you need to configure the security thresholds that the caching CMM (p. 115) 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 soon as possible. The first use of each data key (before caching) is exempt from these thresholds.

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 some of 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, the Java and Python implementations of the AWS Encryption SDK provide a null cryptographic materials cache (null cache). The null cache returns a miss for every GET request and does not respond to PUT requests. We recommend that you use the null cache instead of setting the cache capacity (p. 115) or security thresholds to 0. For more information, see the null cache in Java and Python.

**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. The AWS Encryption SDK does not limit the maximum age value.
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 use data key caching only when you are encrypting empty message strings.
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. 113)
- Creating a cryptographic materials cache (p. 115)
- Creating a caching cryptographic materials manager (p. 115)
- What is in a data key cache entry? (p. 116)
- Encryption context: How to select cache entries (p. 116)
- Is my application using cached data keys? (p. 117)
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.

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. 111). To avoid this behavior, add the message size to the encryption request.

In addition to a cache, data key caching uses a caching cryptographic materials manager (p. 115) (caching CMM). The caching CMM is a specialized cryptographic materials manager (CMM) (p. 7) that interacts with a cache (p. 115) and an underlying CMM (p. 7). (When you specify a master key provider (p. 8) or keyring, the AWS Encryption SDK creates a default CMM for you.) The caching CMM caches the data keys that its underlying CMM returns. The caching CMM also enforces cache security thresholds that you set.

To prevent the wrong data key from being selected from the cache, all compatible caching CMMs require that the following properties of the cached cryptographic materials match the materials request.

- Algorithm suite (p. 8)
- Encryption context (p. 116) (even when empty)
- Partition name (a string that identifies the caching CMM)
- (Decryption only) Encrypted data keys

**Note**
The AWS Encryption SDK caches data keys only when the algorithm suite (p. 8) uses a key derivation function.

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 get encryption materials without caching:

1. An application asks the AWS Encryption SDK to encrypt data.
   
   The request specifies a master key provider or keyring. The AWS Encryption SDK creates a default CMM that interacts with your master key provider or keyring.

2. The AWS Encryption SDK asks the CMM for encryption materials (get cryptographic materials).

3. The CMM asks its keyring (p. 8) (C and JavaScript) or master key provider (p. 8) (Java and Python) for cryptographic materials. This might involve a call to a cryptographic service, such as AWS Key Management Service (AWS KMS). The CMM returns the encryption materials to the AWS Encryption SDK.

4. The AWS Encryption SDK uses the plaintext data key to encrypt the data. It stores the encrypted data and encrypted data keys in an encrypted message (p. 10), which it returns to the user.
Encrypt data with caching

To get encryption materials 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. 115) that is associated with a underlying cryptographic materials manager (CMM). When you specify a master key provider or keyring, the AWS Encryption SDK creates a default CMM for you.

2. The SDK asks the specified caching CMM for encryption materials.

3. The caching CMM requests encryption materials 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 encryption materials to the caching CMM.

      If the cache entry conforms to its security thresholds (p. 111), 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 underlying CMM gets the cryptographic materials from its keyring (C and JavaScript) or master key provider (Java and Python). This might involve a call to a service, such as AWS Key Management Service. The underlying CMM returns the plaintext and encrypted copies of the data key to the caching CMM.

      The caching CMM saves the new encryption materials in the cache.

4. The caching CMM returns the encryption materials to the AWS Encryption SDK.

5. The AWS Encryption SDK uses the plaintext data key to encrypt the data. It stores the encrypted data and encrypted data keys in an encrypted message (p. 10), which it returns 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 a local cache, which is a configurable, in-memory, least recently used (LRU) cache. To create an instance of the local cache, use the `LocalCryptoMaterialsCache` constructor in Java and Python, the `getLocalCryptographicMaterialsCache` function in JavaScript, or the `aws_cryptosdk_materials_cache_local_new` constructor in C.

The local cache 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 the local cache as is, customize it, or substitute any compatible cache.

When you create a local cache, 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 for Java and the AWS Encryption SDK for Python also provide 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. 115) (caching CMM). The caching CMM gets data keys from the cache, puts data keys in the cache, and enforces security thresholds (p. 111) 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. 115) 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. 8) or keyring (p. 8).

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 keyring (C or JavaScript) or a master key provider (Java and Python). When the SDK asks the CMM for encryption or decryption materials, the CMM gets the materials from its keyring or master key provider. In Java and Python, the CMM uses the master keys to generate, encrypt, or decrypt the data keys. In C and JavaScript, the keyring generates, encrypts, and returns the cryptographic materials.

• A caching CMM is associated with one cache, such as a local cache (p. 115), and an underlying CMM. When the SDK asks the caching CMM for cryptographic materials, the caching CMM tries to get them from the cache. If it cannot find a match, the caching CMM asks its underlying CMM for the materials. Then, it caches the new cryptographic materials before returning them to the caller.

The caching CMM also enforces security thresholds (p. 111) 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.

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 an encryption operation include the following elements:

• Plaintext data key
• Encrypted data keys (one or more)
• Encryption context (p. 116)
• Message signing key (if one is used)
• Algorithm suite (p. 8)
• 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. 9) 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. 10) 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.

Is my application using cached data keys?

Data key caching is an optimization strategy that is very effective for certain applications and workloads. However, because it entails some risk, it's important to determine how effective it is likely to be for your situation, and then decide whether the benefits outweigh the risks.

Because data key caching reuses data keys, the most obvious effect is reducing the number of calls to generate new data keys. When data key caching is implemented, the AWS Encryption SDK calls the AWS KMS `GenerateDataKey` operation only to create the initial data key and when the cache misses. But, caching improves performance perceptibly only in applications that generate numerous data keys with the same characteristics, including the same encryption context and algorithm suite.

To determine whether your implementation of the AWS Encryption SDK is actually using data keys from the cache, try the following techniques.

- In the logs of your master key infrastructure, check the frequency of calls to create new data keys. When data key caching is effective, the number of calls to create new keys should drop perceptibly. For example, if you are using a AWS KMS master key provider or keyring, search the CloudTrail logs for `GenerateDataKey` calls.
- Compare the encrypted messages (p. 10) that the AWS Encryption SDK returns in response to different encrypt requests. For example, if you are using the AWS Encryption SDK for Java, compare the `ParsedCiphertext` object from different encrypt calls. In the AWS Encryption SDK for JavaScript, compare the contents of the `encryptedDataKeys` property of the `MessageHeader`. When data keys are reused, the encrypted data keys in the encrypted message are identical.

Data key caching example

This example uses data key caching (p. 97) with a local cache (p. 115) 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 an AWS KMS master key provider (p. 8). To reduce calls to KMS, each producer and consumer has their own local cache.

You can find the source code for these examples in Java (p. 119) and Python (p. 123). The sample also includes a AWS CloudFormation template that defines the resources for the samples.
### Local cache results

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

#### Producer requests

<table>
<thead>
<tr>
<th></th>
<th>Requests per second per client</th>
<th>Clients per region</th>
<th>Average requests per second per region</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generate data key (us-west-2)</td>
<td>Encrypt data key (eu-central-1)</td>
<td>Total (per region)</td>
</tr>
<tr>
<td>No cache</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Local cache</td>
<td>1 rps / 100 uses</td>
<td>1 rps / 100 uses</td>
<td>1 rps / 100 uses</td>
</tr>
</tbody>
</table>
**Consumer requests**

<table>
<thead>
<tr>
<th></th>
<th>Decrypt data key</th>
<th>Producers</th>
<th>Total</th>
<th>Client per region</th>
<th>Average requests per second per region</th>
</tr>
</thead>
<tbody>
<tr>
<td>No cache</td>
<td>1 rps per producer</td>
<td>500</td>
<td>500</td>
<td>2</td>
<td>1,000</td>
</tr>
<tr>
<td>Local cache</td>
<td>1 rps per producer / 100 uses</td>
<td>500</td>
<td>5</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

**Data key caching example in Java**

This code sample creates a basic implementation of data key caching with a local cache ([p. 115](#)) in Java. For details about the Java implementation of the AWS Encryption SDK, see [AWS Encryption SDK for Java](#) ([p. 43](#)).

The code creates two instances of a local cache: one for data producers that are encrypting data and another for data consumers (AWS 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 AWS Region.

The code defines a caching cryptographic materials manager ([p. 115](#)) (caching CMM) and associates it with a local cache ([p. 115](#)) and an underlying AWS KMS master key provider ([p. 8](#)). The caching CMM caches the data keys (and related cryptographic materials ([p. 116](#))) 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.

```java
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

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

/*
 * 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.
 */
```
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 an Amazon 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.
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.caching.KmsMasterKey;
import com.amazonaws.encryptionsdk.caching.KmsMasterKeyProvider;
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.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_;

    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);
    }

    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 example creates a basic implementation of data key caching with a local cache (p. 115) in Python. For details about the Python implementation of the AWS Encryption SDK, see AWS Encryption SDK for Python (p. 60).

The code creates two instances of a local cache; 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.

For a very simple example that focuses on the basic elements of data key caching, see Using data key caching to encrypt messages (p. 66).

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 AWS Region.

The code defines a caching cryptographic materials manager (p. 115) (caching CMM) and associates it with a local cache (p. 115) and an underlying AWS KMS master key provider (p. 8). The caching CMM caches the data keys (and related cryptographic materials (p. 116)) 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."

        # 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.

```python
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 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)

Local cache 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. 130)
• How is the AWS Encryption SDK different from the Amazon S3 encryption client? (p. 130)
• Which cryptographic algorithms are supported by the AWS Encryption SDK, and which one is the default? (p. 130)
• How is the initialization vector (IV) generated and where is it stored? (p. 131)
• How is each data key generated, encrypted, and decrypted? (p. 131)
• How do I keep track of the data keys that were used to encrypt my data? (p. 131)
• How does the AWS Encryption SDK store encrypted data keys with their encrypted data? (p. 131)
• How much overhead does the AWS Encryption SDK message format add to my encrypted data? (p. 131)
• Can I use my own master key provider? (p. 131)
• Can I encrypt data under more than one master key? (p. 131)
• Which data types can I encrypt with the AWS Encryption SDK? (p. 132)
• How does the AWS Encryption SDK encrypt and decrypt input/output (I/O) streams? (p. 132)

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 SDKs 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. 25).
For implementation details about the supported algorithms, see Algorithms reference (p. 147).

**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. 133).

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

The method depends on the master key provider or keyring and its implementation of its master keys or wrapping keys. When AWS KMS is the master key provider, the AWS Encryption 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 or keyring.

**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. 10) 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. 10), 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. 133).

**How much overhead does the AWS Encryption SDK 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. 139) 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. 133).

**Can I use my own master key provider?**

Yes. The implementation details vary depending on which of the supported programming languages (p. 27) you use. However, all supported languages allow you to define custom cryptographic materials managers (CMMs) (p. 7), master key providers, keyrings, master keys, and wrapping keys.

**Can I encrypt data under more than one master key?**

Yes. You can encrypt the data key with additional master keys or wrapping 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 or a keyring with multiple wrapping keys. You can see examples of this pattern in the example code for Java (p. 48) and Python (p. 64). When working with keyrings, you can create a single keyring with multiple wrapping keys (p. 15) or a multi-keyring (p. 22).

When you encrypt data by using a master key provider that returns multiple master keys or a keyring with multiple wrapping keys, the AWS Encryption SDK uses one master key to generate a plaintext data key. It encrypts the data that you pass to the encryption methods with the data key and encrypts that data key with the same master key. Then, it encrypts the data key with the other master keys. The resulting encrypted message (p. 10) 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. 27).

**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. 27).
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. 27).

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in the aws-encryption-sdk-specification repository in GitHub.

The AWS Encryption SDK uses the supported algorithms (p. 25) 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. 133)
- Body additional authenticated data (AAD) reference for the AWS Encryption SDK (p. 141)
- AWS Encryption SDK message format examples (p. 142)
- AWS Encryption SDK algorithms reference (p. 147)
- AWS Encryption SDK initialization vector reference (p. 149)

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. 27).

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in the aws-encryption-sdk-specification repository in GitHub.

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

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**

<table>
<thead>
<tr>
<th>Field</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version (p. 134)</td>
<td>1</td>
<td>The version of this message format. The current version is 1.0, encoded as the byte 01 in hexadecimal notation.</td>
</tr>
<tr>
<td>Type (p. 134)</td>
<td>1</td>
<td>The type of this message format. The type indicates the kind of structure. The only supported type is described as <em>customer authenticated encrypted data</em>. Its type value is 128, encoded as byte 80 in hexadecimal notation.</td>
</tr>
<tr>
<td>Algorithm ID (p. 135)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Message ID (p. 135)</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>AAD Length (p. 135)</td>
<td>2</td>
<td>When the encryption context (p. 9) is empty, the AAD Length is 0. AAD Length is variable. It is equal to the value specified in the previous 2 bytes. When the encryption context (p. 9) is empty, there is no AAD field in the header.</td>
</tr>
<tr>
<td>AAD (p. 135)</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (AAD Length).</td>
<td></td>
</tr>
<tr>
<td>Encrypted Data Key Count (p. 136)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Encrypted Data Key(s) (p. 136)</td>
<td>Variable. Determined by the number of encrypted data keys and the length of each.</td>
<td></td>
</tr>
<tr>
<td>Content Type (p. 137)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Reserved (p. 137)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IV Length (p. 137)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Frame Length (p. 137)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Header Authentication (p. 137)</td>
<td>Variable. Determined by the algorithm (p. 147) that generated the message.</td>
<td></td>
</tr>
</tbody>
</table>

**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. 147).

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. 9) is empty, the AAD Length is 0.

AAD

The additional authenticated data. The AAD is an encoding of the encryption context (p. 9), 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. 147) 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Value Pair Count (p. 135)</td>
<td>2</td>
</tr>
<tr>
<td>Key Length (p. 136)</td>
<td>2</td>
</tr>
<tr>
<td>Key (p. 136)</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Key Length).</td>
</tr>
<tr>
<td>Value Length (p. 136)</td>
<td>2</td>
</tr>
<tr>
<td>Value (p. 136)</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Value Length).</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Provider ID Length (p. 136)</td>
<td>2</td>
</tr>
<tr>
<td>Key Provider ID (p. 136)</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Key Provider ID Length).</td>
</tr>
<tr>
<td>Key Provider Information Length (p. 136)</td>
<td>2</td>
</tr>
<tr>
<td>Key Provider Information (p. 137)</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Key Provider Information Length).</td>
</tr>
<tr>
<td>Encrypted Data Key Length (p. 137)</td>
<td>2</td>
</tr>
<tr>
<td>Encrypted Data Key (p. 137)</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Encrypted Data Key Length).</td>
</tr>
</tbody>
</table>

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 master key provider or you are using an AWS KMS keyring, this value contains the Amazon Resource Name (ARN) of the AWS KMS customer master key (CMK).

Encrypted Data Key Length

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

Encrypted Data Key

The encrypted data key. It is the data encryption key encrypted by the key provider.

Content Type

The type of encrypted content, either nonframed 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. 147) 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 nonframed—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. 147) 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV (p. 137)</td>
<td>Variable. Determined by the IV bytes value of the algorithm (p. 147) that generated the message.</td>
</tr>
<tr>
<td>Authentication Tag (p. 138)</td>
<td>Variable. Determined by the authentication tag bytes value of the algorithm (p. 147) that generated the message.</td>
</tr>
</tbody>
</table>

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 (nonframed or framed). The following sections describe the format of the message body for each content type.

Topics
- Non-framed data (p. 138)
- Framed data (p. 139)

Non-framed data

Non-framed data is encrypted in a single blob with a unique IV and body AAD (p. 141). The following table describes the fields that form nonframed data. The bytes are appended in the order shown.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV (p. 138)</td>
<td>Variable. Equal to the value specified in the IV</td>
</tr>
<tr>
<td></td>
<td>Length (p. 137) byte of the header.</td>
</tr>
<tr>
<td>Encrypted Content Length</td>
<td>8</td>
</tr>
<tr>
<td>Encrypted Content (p. 138)</td>
<td>Variable. Equal to the value specified in the</td>
</tr>
<tr>
<td></td>
<td>previous 8 bytes (Encrypted Content Length).</td>
</tr>
<tr>
<td>Authentication Tag (p. 138)</td>
<td>Variable. Determined by the algorithm</td>
</tr>
<tr>
<td></td>
<td>implementation (p. 147) used.</td>
</tr>
</tbody>
</table>

IV

The initialization vector (IV) to use with the encryption algorithm (p. 147).

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. 147).

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. 147).

Authentication Tag

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

In framed data, the plaintext data is divided into equal-length parts called frames. The AWS Encryption SDK encrypts each frame separately with a unique IV and body AAD (p. 141).

The frame length (p. 137), which is the length of the encrypted content (p. 140) in the frame, can be different for each message. The maximum number of bytes in a frame is $2^{32} - 1$. The maximum number of frames in a message is $2^{32} - 1$.

There are two types of frames: regular and final. Every message must consist of or include a final frame. All regular frames in a message have the same frame length. The final frame can have a different frame length.

The composition of frames in framed data varies with the length of the encrypted content.

- **Equal to the frame length** — When the encrypted content length is the same as the frame length of the regular frames, the message can consist of a regular frame that contains the data, followed by a final frame of zero (0) length. Or, the message can consist only of a final frame that contains the data. In this case, the final frame has the same frame length as the regular frames.

- **Multiple of the frame length** — When the encrypted content length is an exact multiple of the frame length of the regular frames, the message can end in a regular frame that contains the data, followed by a final frame of zero (0) length. Or, the message can end in a final frame that contains the data. In this case, the final frame has the same frame length as the regular frames.

- **Not a multiple of the frame length** — When the encrypted content length is not an exact multiple of the frame length of the regular frames, the final frame contains the remaining data. The frame length of the final frame is less than the frame length of the regular frames.

- **Less than the frame length** — When the encrypted content length is less than the frame length of the regular frames, the message consists of a final frame that contains all of the data. The frame length of the final frame is less than the frame length of the regular frames.

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

### Framed Body Structure, Regular Frame

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number (p. 139)</td>
<td>4</td>
</tr>
<tr>
<td>IV (p. 140)</td>
<td>Variable. Equal to the value specified in the IV Length (p. 137) byte of the header.</td>
</tr>
<tr>
<td>Encrypted Content (p. 140)</td>
<td>Variable. Equal to the value specified in the Frame Length (p. 137) of the header.</td>
</tr>
<tr>
<td>Authentication Tag (p. 140)</td>
<td>Variable. Determined by the algorithm used, as specified in the Algorithm ID (p. 135) of the header.</td>
</tr>
</tbody>
</table>

**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. 147) used.

Encrypted Content

The encrypted content (ciphertext) for the frame, as returned by the encryption algorithm (p. 147).

Authentication Tag

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

Framed Body Structure, Final Frame

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number End (p. 140)</td>
<td>4</td>
</tr>
<tr>
<td>Sequence Number (p. 140)</td>
<td>4</td>
</tr>
<tr>
<td>IV (p. 140)</td>
<td>Variable. Equal to the value specified in the IV Length (p. 137) byte of the header.</td>
</tr>
<tr>
<td>Encrypted Content Length (p. 140)</td>
<td>4</td>
</tr>
<tr>
<td>Encrypted Content (p. 140)</td>
<td>Variable. Equal to the value specified in the previous 4 bytes (Encrypted Content Length).</td>
</tr>
<tr>
<td>Authentication Tag (p. 141)</td>
<td>Variable. Determined by the algorithm used, as specified in the Algorithm ID (p. 135) of the header.</td>
</tr>
</tbody>
</table>

Sequence Number End

An indicator for the final frame. The value is encoded as the 4 bytes \texttt{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. 147).

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. 147).
Authentication Tag

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

Footer structure

When the algorithms with signing (p. 147) 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature Length (p. 141)</td>
<td>2</td>
</tr>
<tr>
<td>Signature (p. 141)</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Signature Length).</td>
</tr>
</tbody>
</table>

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. 27).

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in the aws-encryption-sdk-specification repository in GitHub.

You must provide additional authenticated data (AAD) to the AES-GCM algorithm (p. 147) for each cryptographic operation. This is true for both framed and nonframed body data (p. 138). For more information about AAD and how it is used in Galois/Counter Mode (GCM), see Recommendations for Block Cipher Modes of Operations: Galois/Counter Mode (GCM) and GMAC.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message ID (p. 142)</td>
<td>16</td>
</tr>
</tbody>
</table>
### 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. 27).

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in the aws-encryption-sdk-specification repository in GitHub.

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. 143)
Non-framed data

The following example shows the message format for nonframed data.

```
+--------+
| Header |
+--------+
 01                                         Version (1.0)
 80                                         Type (128, customer authenticated encrypted data)
 0378                                        Algorithm ID (see Algorithms reference)
| 80 | 0004 | 0005 | 30746869 73 | 0002 | 6973 |
| 0003 | 31616E | 000A | 65EB6372 79774690 6F6E | 0008 | 326566F6 76457874 |
| 0007 | 6578616D 706C65 | 0015 | 6177732D 63727970 746F2D70 75626C69 736E73656475726E616E6365 |
| 0044 | 47657374 323A3131 31313232 3233333A6B 65792F37 31356308318D352D613735352D313338613664396131316536 |

| 0002 | 0007 | 004B |
| 6177732D 6B6D73 | 6177732D 6B6D73 | 6177732D 6B6D73 |

Length (75)
6177732D 6B6D737A 75737277


Encrypted Data Key 1, Key Provider ID Length (7)
Encrypted Data Key 1, Key Provider ID ("aws-kms")
```

---

Framed data (p. 145)
Encrypted Data Key 2, Key Provider ID Length (7)
0007
Encrypted Data Key 2, Key Provider ID ("aws-kms")
6177732D 6B6D73
Encrypted Data Key 2, Key Provider Information Length (78)
6172DE3A 6177733A 6B6D733A 63612D63
("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-be3435b2423f")
564E74F2 61CD31 3A313131 31323232
32323333 33346B65 72F93962 33363631
3462DE61 66363E3D 34663138 2D616134
372D6265 33343335 62343233 6666
Encrypted Data Key 2, Encrypted Data Key Length (167)
01010200 78FAFFFB D6DE06AF AC72F9B
EE75BD87 3F60F4E6 FD161444 5A002C94
AF787150 69000000 7E307C06 092A6648
8E70D01 0706A06F 306D2021 00306806
092A6648 86F70D01 0703101E 06096806
48016503 04012E30 11040CB2 A820DC0C
76616EF2 A6B3DD02 0110803B 8073DD0F
FDD010BD 90B9890D 099FDBFC F7B13548
3CC686D7 F3CF7C7A CCC52639 122A1495
71F1BAA6 80E2C43F A34C0E58 11D05114
2A3632C2 A11397
01
00000000
00
00000000
7341C1BE 032F7025 84CDA9D0
IV
2C82BB23 4CBF4AAAB 8F5C6002 622E886C
Authentication Tag
| Body |
| * * * * * * |
D39DD3E5 915E0201 77A4AB11
IV
Encrypted Content Length (654)
00000000 0000028E
08E69F55 8552F2EE4 FD892244 4E1D5L55
5871BAAC 93F78436 1085E4F8 D66ECE28
59455BD8 76479DFC 2802D0B8 0B23D3D3
E4159DFE C8A944B6 685643FC EA24122B
6766ECD5 E35F4653 DF205D30 001DD2D8
55FCD85B 9F5318BC F4265B06 2FE7C741
C7D75BCC 10F05E6A1 0E2F2F40 47A60344
ECB10AA7 55AF633 9DE2C218 12AC8807
95FE9C5C8 C65329D1 377C4C7D EA103EC1
3184F848 A81CBC07 E5A07179 704211E5
B48A2068 B060DF60 B492A737 21B0B2B2
C9B21A10 371B6179 78FAPFB0B BAAEC3F4
9D68E334 701E1442 EA5D288 6448S077
54C0C231 AD43571A B9071925 609A4E59
B8178478 EEB734A4 FAE462B6 F5B374B8
12B0000C 4829F504 936B2492 AAF47E94
A5BA804F 7F190927 5D2DF651 B59D42CF
A15D5551 DAEBA4AF 2060DD0D CB1DA4E6
5E2034DB 4D19B7CD EEA6CF7E 549C86AC
46B2C979 ABB4EE12 202FD6DF E7E3C09F
C2394012 AF20A97E 369BCBDA 62459D3E
C6FFB914 FEFD4DE5 8FF5AEF1 98488557
1BABB5DA 855352EF 4FB7E60C 81C04E8E
F3C8B6B6 71666C06 6BF74E1B 0F881F31
B731B39B CF7116FA 84CA95F5 953D3B44
E3B62D66 338E025B C345C3F8 A31D54F3
6920A7A6 0B8E9E03 552C5A04 917CDD11
D4B5DF5C 491EEB66 20C33FBE 5D21F0AD
6932E6C7 C64B3A26 B98982B5 CFF33328
63490741 3AB79D60 D8AEFFB9 2F48E25A
Framed data

The following example shows the message format for framed data.

```
#--------+
| Header  |
#--------+
01
80
data)
0378
687C0FBD 4DF4A999 717C22A2 DDFE1A27
008E
0004
0005
30746869 73
6973
0003
31616E
000A
656E6372 79774690 6F6E
0008
32636F6E 74657874
0007
6578616D 706C65
0015
6177732D 63727970 746F2D70 75626C69
632D6B54 79
0044
4164A173 7569326F 7430364C 4877715A
("A\aAsui20t0Lk3Q7J8u/AqC2+90kP021cc8T7g2q7rs5aLTg7lvfUEW/86+/5w==")
58444A6E 552F4171 63327644 2B304F6B
704F5A31 63633854 67327164 37727335
61455467 376C7666 5545572F 3836282F
35773DD3
0002
0007
```

0067
Signature Length (103)
30650230 7229DDF5 B86A5B64 54E4D27
CBE194F1 1CCOF8CF D27B7F8B F50658C0
BB4B355 3CED1721 A0BE8A1B 8E1F449E
1BE8B281 023100B2 0CB323EF 584AC2E3
1559963B 89972C3 B55D1700 5FB26E61
331F3E14 BC407CCE B86A66FA CBF74D9E
34CB7E4B 363A3E8

**Authentication Tag**

```
+--------+
| Footer  |
+--------+
```

**Signature**

```
0067
978A019C FE49EE0A 0E96BF0D D6074DDB
66DF313 0E10226F 0A1B219C BE54E4C2
2C15100C 6A2A3F1 8251874 FDC94F6B
9247EF61 3E7B7E0D 29F3ADB9 FA14A29C
76E08EB9 9ADCD98C C8864FDB A696F6CB
E24FDE26 3044C856 BF08F051 1ADD32D9
C4A6A1E 5B572FE 09604F1 F3F3571B
2EAFD9CB B9EB8B3 AE05885A 8FP2D2793
1E3305D9 0C9E2294 88AD7E3B B5E4DCE9
6276C5F1 A3B7E51E 422D365D E4C259C
50775406 822D1682 80B0F2E5 SC94
65B2E942 24BBEEA6 E513F918 CCEC1DE3
```
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
  6573742D 323A3131 31331232 32323333
  33333A6B 65792F37 31356330 3831332D
  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 973E3C3C
  02A4EF29 7F000000 7E307C06 092A8648
  86F70D01 0706A06F 306D0201 00306B06
  092A8648 86F70D01 0701301E 06096066
  48016503 04012E30 11040C3F F02C897B
  7A12EB19 8BF2D082 0110803B 24003D1F
  A5474FBC 392360B5 CB9997E0 6A17DE4C
  A6BD7332 6BF86DAB 60D8CCB8 8295DBE9
  4707E356 ADA3735A 7C52D778 B3135A47
  9F224BF9 E67E87
  0007
  6177732D 6B6D73
  Encrypted Data Key 2, Key Provider ID ("aws-kms")
  004B
  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 614CD3D1 3A111311 31323232
  32333333 333A6B65 792F3962 31333631
  34622D61 6663632D 34366138 2D611343
  372D6265 33343335 62343233 6666
  00A7
  Encrypted Data Key 2, Encrypted Data Key Length
  (167)
  01010200 785F2FDB D6DE06AF AC72F79B
  Encrypted Data Key 2, Encrypted Data Key
  0E57BD87 3F604E6E FD196144 5A002C94
  AF787150 69000000 7E307C06 092A8648
  86F70D01 0706A06F 306D0201 00306B06
  092A8648 86F70D01 0701301E 06096066
  48016503 04012E30 11040C36 CD985E12
  D218B674 5BCC6102 0110803B 0320E3CD
  E470AA27 DEAB660B 3E0CE8E0 8B1A89E4
  57DCC69B AAB1294F 2120C101 9A50D323
  72BEAFDB E24EE3ED 71186E0FA DB40508F
  556FBD58 9E621C
  02
  00000000
  00
  00000100
  4ECBD5C0 9899CA65 923D2347
  0B896144 OA279795 CA571201 4DA58029
  ++-----+
  | Body |
  ++-----+
  00000001
  6B33FE9C AD8BCB23 5B89BE8F
  1F6471B0 A51AF310 10F99F66 F0C76EDF
  F5AFA33C 7D286C6C 9C5D5175 A212AF8E
  FBDA9A0C3 C63FB59 C125DBF2 89AC7939
  BDE4E43A8 0F0049E ACBBDD2B 1C758089
  A90DB923 699A1495 C3B13B50 0A48A830
  201E3AD9 1EA6DA14 7F0496DB 68C104A4
  DE87F372 375ECB28 9BF84B6D 2863898F
  CB80167 9C361C4B SEC07438 7A4822B4
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. 27).

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in the aws-encryption-sdk-specification repository in GitHub.

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

<table>
<thead>
<tr>
<th>Algorithm ID (in 2-byte hex)</th>
<th>Algorithm name</th>
<th>Data key length (in bits)</th>
<th>Algorithm mode</th>
<th>IV length (in bytes)</th>
<th>Authenticator tag length (in bytes)</th>
<th>Key derivation algorithm</th>
<th>Signature algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>03 78</td>
<td>AES</td>
<td>256</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>HKDF with SHA-384</td>
<td>ECDSA with P-384 and SHA-384</td>
</tr>
<tr>
<td>03 46</td>
<td>AES</td>
<td>192</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>HKDF with SHA-384</td>
<td>ECDSA with P-384 and SHA-384</td>
</tr>
<tr>
<td>02 14</td>
<td>AES</td>
<td>128</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>HKDF with SHA-256</td>
<td>ECDSA with P-256 and SHA-256</td>
</tr>
<tr>
<td>01 78</td>
<td>AES</td>
<td>256</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>HKDF with SHA-256</td>
<td>Not applicable</td>
</tr>
<tr>
<td>01 46</td>
<td>AES</td>
<td>192</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>HKDF with SHA-256</td>
<td>Not applicable</td>
</tr>
<tr>
<td>01 14</td>
<td>AES</td>
<td>128</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>HKDF with SHA-256</td>
<td>Not applicable</td>
</tr>
<tr>
<td>00 78</td>
<td>AES</td>
<td>256</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>00 46</td>
<td>AES</td>
<td>192</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>00 14</td>
<td>AES</td>
<td>128</td>
<td>GCM</td>
<td>12</td>
<td>16</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Algorithm ID

A 2-byte value that uniquely identifies an algorithm's implementation. This value is stored in the ciphertext's message header (p. 134).
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. 149).

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 keyring or 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).
To use the AWS Encryption SDK in one of the supported programming languages, see Programming languages (p. 27).

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in the aws-encryption-sdk-specification repository in GitHub.

The AWS Encryption SDK supplies the initialization vectors (IVs) that are required by all supported algorithm suites (p. 147). 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. 97), 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. 151)
• Earlier updates (p. 151)

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: February 5, 2019

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td>General availability</td>
<td>Added and updated documentation for the general availability release of the AWS Encryption SDK for JavaScript.</td>
<td>October 1, 2019</td>
</tr>
<tr>
<td>Preview release</td>
<td>Added and updated documentation of the public beta release of the AWS Encryption SDK for JavaScript.</td>
<td>June 21, 2019</td>
</tr>
<tr>
<td>General availability</td>
<td>Added and updated documentation for the general availability release of the AWS Encryption SDK for C.</td>
<td>May 16, 2019</td>
</tr>
<tr>
<td>Preview release</td>
<td>Added documentation of the preview release of the AWS Encryption SDK for C.</td>
<td>February 5, 2019</td>
</tr>
<tr>
<td>New release</td>
<td>Added documentation of the command line interface for the AWS Encryption SDK.</td>
<td>November 20, 2017</td>
</tr>
</tbody>
</table>

Earlier updates

The following table describes significant changes to the AWS Encryption SDK Developer Guide before November 2017.
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>New release</td>
<td>Added the <em>Data key caching</em> (p. 97) chapter for the new feature.</td>
<td>July 31, 2017</td>
</tr>
<tr>
<td></td>
<td>Added the the section called “Initialization vector reference” (p. 149) topic that explains that the SDK changed from generating random IVs to constructing deterministic IVs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added the the section called “Concepts” (p. 6) topic to explain concepts, including the new cryptographic materials manager.</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Expanded the <em>Message format reference</em> (p. 133) documentation into a new <em>AWS Encryption SDK reference</em> (p. 133) section.</td>
<td>March 21, 2017</td>
</tr>
<tr>
<td></td>
<td>Added a section about the AWS Encryption SDK <em>Supported algorithm suites</em> (p. 25).</td>
<td></td>
</tr>
<tr>
<td>New release</td>
<td>The AWS Encryption SDK now supports the <em>Python</em> (p. 60) programming language, in addition to <em>Java</em> (p. 43).</td>
<td>March 21, 2017</td>
</tr>
<tr>
<td>Initial release</td>
<td>Initial release of the AWS Encryption SDK and this documentation.</td>
<td>March 22, 2016</td>
</tr>
</tbody>
</table>