Developer Guide

AWS Encryption SDK

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.
AWS Encryption SDK: Developer Guide

Copyright © 2024 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
  Developed in open-source repositories ...................................................... 2
  Compatibility with encryption libraries and services .................................... 3
  Support and maintenance ............................................................................. 4
Learning more ............................................................................................... 4
Sending feedback ............................................................................................ 5
Concepts .......................................................................................................... 6
  Envelope encryption ..................................................................................... 7
  Data key ....................................................................................................... 8
  Wrapping key ............................................................................................. 9
  Keyrings and master key providers ............................................................. 10
Encryption context .......................................................................................... 11
Encrypted message ......................................................................................... 12
Algorithm suite ............................................................................................... 12
Cryptographic materials manager ............................................................... 13
Symmetric and asymmetric encryption ......................................................... 13
Key commitment ............................................................................................. 14
Commitment policy ....................................................................................... 15
Digital signatures ........................................................................................... 17
How the SDK works ...................................................................................... 17
  How the AWS Encryption SDK encrypts data .............................................. 18
  How the AWS Encryption SDK decrypts an encrypted message ............... 18
Supported algorithm suites .......................................................................... 19
  Recommended: AES-GCM with key derivation, signing, and key commitment 19
  Other supported algorithm suites ............................................................... 20
Interacting with AWS KMS ............................................................................ 22
Best practices .................................................................................................. 24
Configuring the SDK ...................................................................................... 28
  Selecting a programming language ............................................................ 28
  Selecting wrapping keys .............................................................................. 28
  Using multi-Region AWS KMS keys .......................................................... 30
  Choosing an algorithm suite ...................................................................... 51
  Limiting encrypted data keys ...................................................................... 60
  Creating a discovery filter ......................................................................... 64
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting a commitment policy</td>
<td>66</td>
</tr>
<tr>
<td>Working with streaming data</td>
<td>67</td>
</tr>
<tr>
<td>Caching data keys</td>
<td>67</td>
</tr>
<tr>
<td>Using keyrings</td>
<td>68</td>
</tr>
<tr>
<td>How keyrings work</td>
<td>69</td>
</tr>
<tr>
<td>Keyring compatibility</td>
<td>70</td>
</tr>
<tr>
<td>Varying requirements for encryption keyrings</td>
<td>71</td>
</tr>
<tr>
<td>Compatible Keyrings and Master Key Providers</td>
<td>71</td>
</tr>
<tr>
<td>Choosing a keyring</td>
<td>72</td>
</tr>
<tr>
<td>AWS KMS keyrings</td>
<td>73</td>
</tr>
<tr>
<td>AWS KMS Hierarchical keyrings</td>
<td>92</td>
</tr>
<tr>
<td>Raw AES keyrings</td>
<td>116</td>
</tr>
<tr>
<td>Raw RSA keyrings</td>
<td>119</td>
</tr>
<tr>
<td>Multi-keyrings</td>
<td>125</td>
</tr>
<tr>
<td>Programming languages</td>
<td>130</td>
</tr>
<tr>
<td>C</td>
<td>130</td>
</tr>
<tr>
<td>Installing</td>
<td>131</td>
</tr>
<tr>
<td>Using the C SDK</td>
<td>132</td>
</tr>
<tr>
<td>Examples</td>
<td>137</td>
</tr>
<tr>
<td>.NET</td>
<td>144</td>
</tr>
<tr>
<td>Install and build</td>
<td>146</td>
</tr>
<tr>
<td>Debugging</td>
<td>146</td>
</tr>
<tr>
<td>AWS KMS keyrings</td>
<td>147</td>
</tr>
<tr>
<td>Required encryption context CMM</td>
<td>150</td>
</tr>
<tr>
<td>Examples</td>
<td>152</td>
</tr>
<tr>
<td>Java</td>
<td>160</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>160</td>
</tr>
<tr>
<td>Installation</td>
<td>162</td>
</tr>
<tr>
<td>AWS KMS keyrings</td>
<td>163</td>
</tr>
<tr>
<td>Required encryption context CMM</td>
<td>166</td>
</tr>
<tr>
<td>Examples</td>
<td>168</td>
</tr>
<tr>
<td>JavaScript</td>
<td>181</td>
</tr>
<tr>
<td>Compatibility</td>
<td>181</td>
</tr>
<tr>
<td>Installation</td>
<td>183</td>
</tr>
<tr>
<td>Modules</td>
<td>184</td>
</tr>
<tr>
<td>Examples</td>
<td>187</td>
</tr>
</tbody>
</table>
Migrating your AWS Encryption SDK ................................................................. 345
  How to migrate and deploy ................................................................. 347
    Stage 1: Update your application to the latest 1.x version .................. 347
    Stage 2: Update your application to the latest version ..................... 348
  Updating AWS KMS master key providers ............................................. 349
    Migrating to strict mode ................................................................ 350
    Migrating to discovery mode .......................................................... 354
  Updating AWS KMS keyrings ............................................................... 357
  Setting your commitment policy ......................................................... 359
    How to set your commitment policy ................................................ 361
  Troubleshooting migration to the latest versions .................................... 368
    Deprecated or removed objects ....................................................... 369
    Configuration conflict: Commitment policy and algorithm suite ........ 369
    Configuration conflict: Commitment policy and ciphertext ............... 370
    Key commitment validation failed ................................................... 371
    Other encryption failures .............................................................. 371
    Other decryption failures .............................................................. 371
    Rollback considerations ............................................................... 372

Frequently asked questions ..................................................................... 373

Reference .................................................................................................. 378
  Message format reference ................................................................. 378
    Header structure ............................................................................ 379
    Body structure ............................................................................. 387
    Footer structure ............................................................................ 392
  Message format examples ................................................................. 393
    Framed data (message format version 1) ......................................... 393
    Framed data (message format version 2) ......................................... 397
    Non-framed data (message format version 1) ................................... 399
  Body AAD reference .......................................................................... 403
  Algorithms reference ......................................................................... 404
  Initialization vector reference .......................................................... 409
  AWS KMS Hierarchical keyring technical details ................................. 410

Document history .................................................................................... 411
Recent updates .......................................................................................................................... 411
Earlier updates .......................................................................................................................... 414
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?
- How do I use the data keys that AWS KMS returns?

With the AWS Encryption SDK, you define a master key provider (Java and Python) or a keyring (C, C#/.NET, and JavaScript) that determines which wrapping 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 the section called “Supported algorithm suites”.

---
A framework for protecting data keys with wrapping keys

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

For example, encrypt data under an AWS KMS key in AWS KMS and a key from your on-premises HSM. You can use either of the wrapping keys to decrypt the data, in case one is unavailable or the caller doesn't have permission to use both keys.

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 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 KMS keys to protect your data.

Developed in open-source repositories

The AWS Encryption SDK is developed in open-source repositories on GitHub. You can use these repositories to view the code, read and submit issues, and find information that is specific to your language implementation.

- AWS Encryption SDK for C — aws-encryption-sdk-c
- AWS Encryption SDK for .NET — aws-encryption-sdk-net directory of the aws-encryption-sdk-dafny repository.
- AWS Encryption CLI — aws-encryption-sdk-cli
- AWS Encryption SDK for Java — aws-encryption-sdk-java
- AWS Encryption SDK for JavaScript — aws-encryption-sdk-javascript
- AWS Encryption SDK for Python — aws-encryption-sdk-python
Compatibility with encryption libraries and services

The AWS Encryption SDK is supported in several 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”.

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. These libraries cannot decrypt the encrypted message the AWS Encryption SDK returns.

**AWS Key Management Service (AWS KMS)**

The AWS Encryption SDK can use AWS KMS keys and data keys to protect your data, including multi-Region KMS keys. For example, you can configure the AWS Encryption SDK to encrypt your data under one or more AWS KMS keys 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 the AWS Encryption SDK returns.

The AWS Encryption SDK supports only symmetric encryption KMS keys. You cannot use an asymmetric KMS key for encryption or signing in the AWS Encryption SDK. The AWS Encryption SDK generates its own ECDSA signing keys for algorithm suites 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.
Support and maintenance

The AWS Encryption SDK uses the same maintenance policy that the AWS SDK and Tools use, including its versioning and life-cycle phases. As a best practice, we recommend that you use the latest available version of the AWS Encryption SDK for your programming language, and upgrade as new versions are released. When a version requires significant changes, such as the upgrade from AWS Encryption SDK versions earlier than 1.7.x to versions 2.0.x and later, we provide detailed instructions to help you.

Each programming language implementation of the AWS Encryption SDK is developed in a separate open-source GitHub repository. The life-cycle and support phase of each version is likely to vary among repositories. For example, a given version of the AWS Encryption SDK might be in the general availability (full support) phase in one programming language, but the end-of-support phase in a different programming language. We recommend that you use a fully supported version whenever possible and avoid versions that are no longer supported.

To find the life-cycle phase of AWS Encryption SDK versions for your programming language, see the SUPPORT_POLICY.rst file in each AWS Encryption SDK repository.

- AWS Encryption SDK for C — SUPPORT_POLICY.rst
- AWS Encryption SDK for .NET — SUPPORT_POLICY.rst
- AWS Encryption CLI — SUPPORT_POLICY.rst
- AWS Encryption SDK for Java — SUPPORT_POLICY.rst
- AWS Encryption SDK for JavaScript — SUPPORT_POLICY.rst
- AWS Encryption SDK for Python — SUPPORT_POLICY.rst

For more information, see Versions of the AWS Encryption SDK and AWS SDKs and Tools maintenance policy in the AWS SDKs and Tools Reference Guide.

Learning more

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

- For help with the terms and concepts used in this SDK, see Concepts in the AWS Encryption SDK.
- For best practice guidelines, see Best practices for the AWS Encryption SDK.
For information about how this SDK works, see [How the SDK works](#).  
For examples that show how to configure options in the AWS Encryption SDK, see [Configuring the AWS Encryption SDK](#).  
For detailed technical information, see the [Reference](#).  
For the technical specification for the AWS Encryption SDK, see the [AWS Encryption SDK Specification](#) 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](#), the AWS Encryption SDK [C documentation](#), and the [aws-encryption-sdk-c](#) repository on GitHub.
- **C#/.NET**: See [AWS Encryption SDK for .NET](#) and the [aws-encryption-sdk-net](#) directory of the aws-encryption-sdk-dafny repository on GitHub.
- **Command Line Interface**: See [AWS Encryption SDK command line interface](#), [Read the Docs](#) for the AWS Encryption CLI, and the [aws-encryption-sdk-cli](#) repository on GitHub.
- **Java**: See [AWS Encryption SDK for Java](#), the AWS Encryption SDK [Javadoc](#), and the [aws-encryption-sdk-java](#) repository on GitHub.
- **JavaScript**: See the section called “JavaScript” and the [aws-encryption-sdk-javascript](#) repository on GitHub.
- **Python**: See [AWS Encryption SDK for Python](#), the AWS Encryption SDK [Python documentation](#), and the [aws-encryption-sdk-python](#) 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 links 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.

Concepts in the AWS Encryption SDK

This section introduces the concepts used in the AWS Encryption SDK, and provides a glossary and reference. It's designed to help you understand how the AWS Encryption SDK works and the terms we use to describe it.

Need help?

• Learn how the AWS Encryption SDK uses envelope encryption to protect your data.
• Learn about the elements of envelope encryption: the data keys that protect your data and the wrapping keys that protect your data keys.
• Learn about the keyrings and master key providers that determine which wrapping keys you use.
• Learn about the encryption context that adds integrity to your encryption process. It's optional, but it's a best practice that we recommend.
• Learn about the encrypted message that the encryption methods return.
• Then you're ready to use the AWS Encryption SDK in your preferred programming language.

Topics

• Envelope encryption
• Data key
• Wrapping key
• Keyrings and master key providers
• Encryption context
• Encrypted message
• Algorithm suite
• Cryptographic materials manager
• Symmetric and asymmetric encryption
• Key commitment
• Commitment policy
Digital signatures

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 key-encryption key or wrapping key. The practice of using a wrapping key to encrypt data keys is known as envelope encryption.

Protecting data keys

The AWS Encryption SDK encrypts each message with a unique data key. Then it encrypts the data key under the wrapping key you specify. It stores the encrypted data key with the encrypted data in the encrypted message that it returns.

To specify your wrapping key, you use a keyring or master key provider.

Encrypting the same data under multiple wrapping keys

You can encrypt the data key under multiple wrapping keys. You might want to provide different wrapping keys for different users, or wrapping keys of different types, or in different locations. Each of the wrapping keys encrypts the same data key. The AWS Encryption SDK stores all of the encrypted data keys with the encrypted data in the encrypted message.

To decrypt the data, you need to provide a wrapping key that can decrypt one of the encrypted data keys.
Combining the strengths of multiple algorithms

To encrypt your data, by default, the AWS Encryption SDK uses a sophisticated algorithm suite with AES-GCM symmetric encryption, a key derivation function (HKDF), and signing. To encrypt the data key, you can specify a symmetric or asymmetric encryption algorithm appropriate to your wrapping key.

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. To combine the strengths of each, you can encrypt your data with symmetric key encryption, and then encrypt the data key with public key encryption.

Data key

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, the AWS Encryption SDK uses a unique data key to encrypt each message.

You don't need to specify, generate, implement, extend, protect or use data keys. The AWS Encryption SDK does that work for you when you call the encrypt and decrypt operations.

To protect your data keys, the AWS Encryption SDK encrypts them under one or more key-encryption keys known as wrapping keys or master 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 that the encrypt operations return. For details, see the section called “How the SDK works”.

Tip

In the AWS Encryption SDK, we distinguish data keys from data encryption keys. Several of the supported algorithm suites, 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.

Each encrypted data key includes metadata, including the identifier of the wrapping key that encrypted it. This metadata makes it easier for the AWS Encryption SDK to identify valid wrapping keys when decrypting.

Wrapping key

A wrapping key is a key-encryption key that the AWS Encryption SDK uses to encrypt the data key that encrypts your data. Each plaintext data key can be encrypted under one or more wrapping keys. You determine which wrapping keys are used to protect your data when you configure a keyring or master key provider.

Note

Wrapping key refers to the keys in a keyring or master key provider. Master key is typically associated with the MasterKey class that you instantiate when you use a master key provider.

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

When you use envelope encryption, you need to protect your wrapping keys from unauthorized access. You can do this in any 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 other key management tools and services.

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 wrapping keys. However, the AWS Encryption SDK does not require AWS or any AWS service.

**Keyrings and master key providers**

To specify the wrapping keys you use for encryption and decryption, you use a keyring (C, C# / .NET, and JavaScript) or a master key provider (Java, Python, CLI). You can use the keyrings and master key providers that the AWS Encryption SDK provides or design your own implementations. The AWS Encryption SDK provides keyrings and master key providers that are compatible with each other subject to language constraints. For details, see Keyring compatibility.

A keyring generates, encrypts, and decrypts data keys. When you define a keyring, you can specify the wrapping keys that encrypt your data keys. Most keyrings specify at least one wrapping key or a service that provides and protects wrapping keys. You can also define a keyring with no wrapping keys or a more complex keyring with additional configuration options. For help choosing and using the keyrings that the AWS Encryption SDK defines, see Using keyrings. Keyrings are supported in C, C# / .NET, JavaScript, and version 3.x of the AWS Encryption SDK for Java.

A master key provider is an alternative to a keyring. The master key provider returns the wrapping keys (or master keys) you specify. Each master key is associated with one master key provider, but a master key provider typically provides multiple master keys. Master key providers are supported in Java, Python, and the AWS Encryption CLI.

You must specify a keyring (or master key provider) for encryption. You can specify the same keyring (or master key provider), or a different one, for decryption. When encrypting, the AWS Encryption SDK uses all of the wrapping keys you specify to encrypt the data key. When decrypting, the AWS Encryption SDK uses only the wrapping keys you specify to decrypt an encrypted data key. Specifying wrapping keys for decryption is optional, but it's a AWS Encryption SDK best practice.

For details about specifying wrapping keys, see Selecting wrapping keys.
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 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 (CMM) adds. Specifically, whenever you use an encryption algorithm with signing, 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 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 (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 programming language implementations, 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.

Note

With version 4.x of the AWS Encryption SDK for .NET and version 3.x of the AWS Encryption SDK for Java, you can require an encryption context in all encrypt requests with the required encryption context CMM.

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 that the AWS Encryption 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 submitting and verifying an encryption context in your code, see the examples for your preferred programming language.

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 that includes the encrypted data along with encrypted copies of the data keys, the algorithm ID, and, optionally, an encryption context and a digital 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.

Algorithm suite

The AWS Encryption SDK uses an algorithm suite to encrypt and sign the data in the encrypted message that the encrypt and decrypt operations return. The AWS Encryption SDK supports several algorithm suites. 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 manager (CMM), but unless an alternate is required for your situation, it is best to use the default. The current default is AES-GCM with an HMAC-based extract-and-expand key derivation function (HKDF), key commitment, an Elliptic Curve Digital Signature Algorithm (ECDSA) signature, and a 256-bit encryption key.

If your application requires high performance and the users who are encrypting data and those who are decrypting data are equally trusted, you might consider specifying an algorithm suite without a digital signature. However, we strongly recommend an algorithm suite that includes key commitment and a key derivation function. Algorithm suites without these features are supported only for backward compatibility.

**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 never interact with the CMM directly. The encryption and decryption methods handle it for you.

You can use the default CMM or the caching CMM that the AWS Encryption SDK provides, or write a custom CMM. And you can specify a CMM, but it's not required. When you specify a keyring or master key provider, the AWS Encryption SDK creates a default CMM for you. The default CMM gets the encryption or decryption materials from the keyring or master key provider that you specify. This might involve a call to a cryptographic service, such as AWS Key Management Service (AWS KMS).

Because the CMM acts as a liaison between the AWS Encryption 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.

**Symmetric and asymmetric encryption**

*Symmetric encryption* uses the same key to encrypt and decrypt data.

*Asymmetric encryption* uses a mathematically related data key pair. One key in the pair encrypts the data; only the other key in the pair can decrypt the data. For details, see Cryptographic algorithms in the AWS Cryptographic Services and Tools Guide.
The AWS Encryption SDK uses **envelope encryption**. It encrypts your data with a symmetric data key. It encrypts the symmetric data key with one or more symmetric or asymmetric wrapping keys. It returns an **encrypted message** that includes the encrypted data and at least one encrypted copy of the data key.

**Encrypting your data (symmetric encryption)**

To encrypt your data, the AWS Encryption SDK uses a symmetric **data key** and an **algorithm suite** that includes a symmetric encryption algorithm. To decrypt the data, the AWS Encryption SDK uses the same data key and the same algorithm suite.

**Encrypting your data key (symmetric or asymmetric encryption)**

The **keyring** or **master key provider** that you supply to an encrypt and decrypt operation determines how the symmetric data key is encrypted and decrypted. You can choose a keyring or master key provider that uses symmetric encryption, such as an AWS KMS keyring, or one that uses asymmetric encryption, such as a raw RSA keyring or JceMasterKey.

**Key commitment**

The AWS Encryption SDK supports **key commitment** (sometimes known as **robustness**), a security property that guarantees that each ciphertext can be decrypted only to a single plaintext. To do this, key commitment guarantees that only the data key that encrypted your message will be used to decrypt it. Encrypting and decrypting with key commitment is an **AWS Encryption SDK best practice**.

Most modern symmetric ciphers (including AES) encrypt a plaintext under a single secret key, such as the **unique data key** that the AWS Encryption SDK uses to encrypt each plaintext message. Decrypting this data with the same data key returns a plaintext that is identical to the original. Decrypting with a different key will usually fail. However, it's possible to decrypt a ciphertext under two different keys. In rare cases, it is feasible to find a key that can decrypt a few bytes of ciphertext into a different, but still intelligible, plaintext.

The AWS Encryption SDK always encrypts each plaintext message under one unique data key. It might encrypt that data key under multiple wrapping keys (or master keys), but the wrapping keys always encrypt the same data key. Nonetheless, a sophisticated, manually crafted **encrypted message** might actually contain different data keys, each encrypted by a different wrapping key. For example, if one user decrypts the encrypted message it returns 0x0 (false) while another user decrypting the same encrypted message gets 0x1 (true).
To prevent this scenario, the AWS Encryption SDK supports key commitment when encrypting and decrypting. When the AWS Encryption SDK encrypts a message with key commitment, it cryptographically binds the unique data key that produced the ciphertext to the *key commitment string*, a non-secret data key identifier. Then it stores key commitment string in the metadata of the encrypted message. When it decrypts a message with key commitment, the AWS Encryption SDK verifies that the data key is the one and only key for that encrypted message. If data key verification fails, the decrypt operation fails.

Support for key commitment is introduced in version 1.7.x, which can decrypt messages with key commitment, but won't encrypt with key commitment. You can use this version to fully deploy the ability to decrypt ciphertext with key commitment. Version 2.0.x includes full support for key commitment. By default, it encrypts and decrypts only with key commitment. This is an ideal configuration for applications that don't need to decrypt ciphertext encrypted by earlier versions of the AWS Encryption SDK.

Although encrypting and decrypting with key commitment is a best practice, we let you decide when it's used, and let you adjust the pace at which you adopt it. Beginning in version 1.7.x, AWS Encryption SDK supports a commitment policy that sets the default algorithm suite and limits the algorithm suites that may be used. This policy determines whether your data is encrypted and decrypted with key commitment.

Key commitment results in a slightly larger (+ 30 bytes) encrypted message and takes more time to process. If your application is very sensitive to size or performance, you might choose to opt out of key commitment. But do so only if you must.

For more information about migrating to versions 1.7.x and 2.0.x, including their key commitment features, see [Migrating your AWS Encryption SDK](#). For technical information about key commitment, see the section called “Algorithms reference” and the section called “Message format reference”.

**Commitment policy**

A *commitment policy* is a configuration setting that determines whether your application encrypts and decrypts with key commitment. Encrypting and decrypting with key commitment is an AWS Encryption SDK best practice.

Commitment policy has three values.
### Commitment policy values

<table>
<thead>
<tr>
<th>Value</th>
<th>Encrypts with key commitment</th>
<th>Encrypts without key commitment</th>
<th>Decrypts with key commitment</th>
<th>Decrypts without key commitment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForbidEncryptAllowDecrypt</td>
<td>![X]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
</tr>
<tr>
<td>RequireEncryptAllowDecrypt</td>
<td>![Checkmark]</td>
<td>![X]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
</tr>
<tr>
<td>RequireEncryptRequireDecrypt</td>
<td>![Checkmark]</td>
<td>![X]</td>
<td>![Checkmark]</td>
<td>![X]</td>
</tr>
</tbody>
</table>

The commitment policy setting is introduced in AWS Encryption SDK version 1.7.x. It's valid in all supported programming languages.

- **ForbidEncryptAllowDecrypt** decrypts with or without key commitment, but it won't encrypt with key commitment. This is the only valid value for commitment policy in version 1.7.x and it is used for all encrypt and decrypt operations. It's designed to prepare all hosts running your application to decrypt with key commitment before they ever encounter a ciphertext encrypted with key commitment.
- **RequireEncryptAllowDecrypt** always encrypts with key commitment. It can decrypt with or without key commitment. This value, introduced in version 2.0.x, lets you start encrypting with key commitment, but still decrypt legacy ciphertexts without key commitment.
- **RequireEncryptRequireDecrypt** encrypts and decrypts only with key commitment. This value is the default for version 2.0.x. Use this value when you are certain that all of your ciphertexts are encrypted with key commitment.
The commitment policy setting determines which algorithm suites you can use. Beginning in version 1.7.x, the AWS Encryption SDK supports algorithm suites for key commitment; with and without signing. If you specify an algorithm suite that conflicts with your commitment policy, the AWS Encryption SDK returns an error.

For help setting your commitment policy, see Setting your commitment policy.

Digital signatures

To ensure the integrity of a digital message as it goes between systems, you can apply a digital signature to the message. Digital signatures are always asymmetric. You use your private key to create the signature, and append it to the original message. Your recipient uses a public key to verify that the message has not been modified since you signed it.

The AWS Encryption SDK encrypts your data using an authenticated encryption algorithm, AES-GCM, and the decryption process verifies the integrity and authenticity of an encrypted message without using a digital signature. But because AES-GCM uses symmetric keys, anyone who can decrypt the data key used to decrypt the ciphertext could also manually create a new encrypted ciphertext, causing a potential security concern. For instance, if you use an AWS KMS key as a wrapping key, this means that it is possible for a user with KMS Decrypt permissions to create encrypted ciphertexts without calling KMS Encrypt.

To avoid this issue, the AWS Encryption SDK supports adding an Elliptic Curve Digital Signature Algorithm (ECDSA) signature to the end of encrypted messages. When a signing algorithm suite is used, the AWS Encryption SDK generates a temporary private key and public key pair for each encrypted message. The AWS Encryption SDK stores the public key in the encryption context of the data key and discards the private key, and no one can create another signature that verifies with the public key. Because the algorithm binds the public key to the encrypted data key as additional authenticated data in the message header, a user who can only decrypt messages cannot alter the public key.

Signature verification adds a significant performance cost on decryption. If the users encrypting data and the users decrypting data are equally trusted, consider using an algorithm suite that does not include signing.

How the AWS Encryption SDK works

The workflows in this section explain how the AWS Encryption SDK encrypts data and decrypts encrypted messages. These workflows describes the basic process using the default features.
For details about defining and using custom components, see the GitHub repository for each supported language implementation.

The AWS Encryption SDK uses envelope encryption to protect your data. Each message is encrypted under a unique data key. Then the data key is encrypted by the wrapping keys you specify. To decrypt the encrypted message, the AWS Encryption SDK uses the wrapping keys you specify to decrypt at least one encrypted data key. Then it can decrypt the ciphertext and return a plaintext message.

Need help with the terminology we use in the AWS Encryption SDK? See the section called “Concepts”.

**How the AWS Encryption SDK encrypts data**

The AWS Encryption SDK provides methods that encrypt strings, byte arrays, and byte streams. For code examples, see the Examples topic in each Programming languages section.

1. Create a keyring (or master key provider) that specifies the wrapping keys that protect your data.
2. Pass the keyring and plaintext data to an encryption method. We recommend that you pass in an optional, non-secret encryption context.
3. The encryption method asks the keyring for encryption materials. The keyring returns unique data encryption keys for the message: one plaintext data key and one copy of that data key encrypted by each of the specified wrapping keys.
4. The encryption method uses the plaintext data key to encrypt the data, and then discards the plaintext data key. If you provide an encryption context (an AWS Encryption SDK best practice), the encryption method cryptographically binds the encryption context to the encrypted data.
5. The encryption method returns an encrypted message that contains the encrypted data, the encrypted data keys, and other metadata, including the encryption context, if you used one.

**How the AWS Encryption SDK decrypts an encrypted message**

The AWS Encryption SDK provides methods that decrypt the encrypted message and return plaintext. For code examples, see the Examples topic in each Programming languages section.

The keyring (or master key provider) that decrypts the encrypted message must be compatible with the one used to encrypt the message. One of its wrapping keys must be able to decrypt an encrypted data key in the encrypted message. For information about compatibility with keyrings and master key providers, see the section called “Keyring compatibility”.
1. Create a keyring or master key provider with wrapping keys that can decrypt your data. You can use the same keyring that you provided to the encryption method or a different one.

2. Pass the encrypted message and the keyring to a decryption method.

3. The decryption method asks the keyring or master key provider to decrypt one of the encrypted data keys in the encrypted message. It passes in information from the encrypted message, including the encrypted data keys.

4. The keyring uses its wrapping keys to decrypt one of the encrypted data keys. If it's successful, the response includes the plaintext data key. If none of the wrapping keys specified by the keyring or master key provider can decrypt an encrypted data key, the decrypt call fails.

5. The decryption method uses the plaintext data key to decrypt the data, discards the plaintext data key, and returns the plaintext data.

**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 AWS Encryption 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.

By default, the AWS Encryption SDK uses an algorithm suite with AES-GCM with an HMAC-based extract-and-expand key derivation function (HKDF), signing, and a 256-bit encryption key. If the commitment policy requires key commitment, the AWS Encryption SDK selects an algorithm suite that also supports key commitment; otherwise, it selects an algorithm suite with key derivation and signing, but not key commitment.

**Recommended: AES-GCM with key derivation, signing, and key commitment**

The AWS Encryption SDK recommends an algorithm suite that derives an AES-GCM encryption key by supplying a 256-bit data encryption key to the HMAC-based extract-and-expand key derivation function (HKDF). The AWS Encryption SDK adds an Elliptic Curve Digital Signature Algorithm (ECDSA) signature. To support key commitment, this algorithm suite also derives a key commitment string – a non-secret data-key identifier – that is stored in the metadata of the encrypted message.
This key commitment string is also derived through HKDF using a procedure similar to deriving the data encryption key.

**AWS Encryption SDK Algorithm Suite**

<table>
<thead>
<tr>
<th>Encryption algorithm</th>
<th>Data encryption key length (in bits)</th>
<th>Key derivation algorithm</th>
<th>Signature algorithm</th>
<th>Key commitment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-GCM</td>
<td>256</td>
<td>HKDF with SHA-384</td>
<td>ECDSA with P-384 and SHA-384</td>
<td>HKDF with SHA-512</td>
</tr>
</tbody>
</table>

The HKDF helps you avoid accidental reuse of a data encryption key and reduces the risk of overusing a data key.

For signing, this algorithm suite uses ECDSA with a cryptographic hash function algorithm (SHA-384). ECDSA is used by default, even when it is not specified by the policy for the underlying master key. **Message signing** verifies the message sender was authorized to encrypt messages and provides non-repudiation. 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.

Algorithm suites with key commitment ensure that each ciphertext decrypts to only one plaintext. They do this by validating the identity of the data key used as input to the encryption algorithm. When encrypting, these algorithm suites derive a key commitment string. Before decrypting, they validate that the data key matches the key commitment string. If it does not, the decrypt call fails.

**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 signing can hinder performance significantly, so we offer a key committing suite with key derivation for those cases. For applications that must make more significant performance tradeoffs, we continue to offer suites that lack signing, key commitment, and key derivation.

**AES-GCM without key commitment**

Algorithm suites without key commitment do not validate the data key before decrypting. As a result, these algorithm suites might decrypt a single ciphertext into different plaintext
messages. However, because algorithm suites with key commitment produce a slightly larger (+30 bytes) encrypted message and take longer to process, they might not be the best choice for every application.

The AWS Encryption SDK supports an algorithm suite with key derivation, key commitment, signing, and one with key derivation and key commitment, but not signing. We do not recommend using an algorithm suite without key commitment. If you must, we recommend an algorithm suite with key derivation and key commitment, but not signing. However, if your application performance profile supports using an algorithm suite, using an algorithm suite with key commitment, key derivation, and signing is a best practice.

**AES-GCM without signing**

Algorithm suites without signing lack the ECDSA signature that provides authenticity and non-repudiation. Use these suites only when the users who encrypt data and those who decrypt data are equally trusted.

When using an algorithm suite without signing, we recommend that you choose one with key derivation and key commitment.

**AES-GCM without key derivation**

Algorithm suites without key derivation use 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 AWS Encryption 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”.
Using the AWS Encryption SDK with AWS KMS

To use the AWS Encryption SDK, you need to configure keyrings or master key providers with wrapping 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 key.

To interact with AWS KMS, the AWS Encryption SDK requires the AWS SDK for your preferred programming language. 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 symmetric encryption AWS KMS key. For help, see Creating Keys in the AWS Key Management Service Developer Guide.

   Tip
   To use the AWS KMS key programmatically, you will need the key ID or Amazon Resource Name (ARN) of the AWS KMS key. For help finding the ID or ARN of an AWS KMS key, see Finding the Key ID and ARN in the AWS Key Management Service Developer Guide.

3. Generate an access key ID and security access key. You can use either the access key ID and secret access key for an IAM user or you can use the AWS Security Token Service to create a new session with temporary security credentials that include an access key ID, secret access key, and session token. As a security best practice, we recommend that you use temporary credentials instead of the long-term credentials associated with your IAM user or AWS (root) user accounts.

   To create an IAM user with an access key, see Creating IAM Users in the IAM User Guide.

   To generate temporary security credentials, see Requesting temporary security credentials in the IAM User Guide.

4. Set your AWS credentials using the instructions in the AWS SDK for Java, AWS SDK for JavaScript, AWS SDK for Python (Boto) or AWS SDK for C++ (for C), and the access key ID and
secret access key that you generated in step 3. If you generated temporary credentials, you will also need to specify the session token.

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](#) that you want to use.
Best practices for the AWS Encryption SDK

The AWS Encryption SDK is designed to make it easy for you to protect your data using industry standards and best practices. While many best practices are selected for you in default values, some practices are optional but recommended whenever it's practical.

Use the latest version

When you start using the AWS Encryption SDK, use the latest version offered in your preferred programming language. If you've been using the AWS Encryption SDK, upgrade to each latest version as soon as possible. This assures that you're using the recommended configuration and taking advantage of new security properties to protect your data. For details about supported versions, including guidance for migration and deployment, see Support and maintenance and Versions of the AWS Encryption SDK.

If a new version deprecates elements in your code, replace them as soon as you can. Deprecation warnings and code comments typically recommend a good alternative.

To make significant upgrades easier and less prone to error, we occasionally provide a temporary or transitional release. Use these releases, and their accompanying documentation, to assure that you can upgrade your application without disrupting your production workflow.

Use default values

The AWS Encryption SDK designs best practices into its default values. Whenever possible, use them. For cases where the default is impractical, we provide alternatives, such as algorithm suites without signing. We also provide opportunities to advanced users for customization, such as custom keyrings, master key providers, and cryptographic material managers (CMMs). Use these advanced alternatives cautiously and have your choices verified by a security engineer whenever possible.

Use an encryption context

To improve the security of your cryptographic operations, include an encryption context with a meaningful value 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 provides additional authenticated data (AAD) for authenticated encryption in the AWS Encryption SDK. Although it is not secret, the encryption context can help you to protect the integrity and authenticity of your encrypted data.
In the AWS Encryption SDK, you specify an encryption context only when encrypting. When decrypting, the AWS Encryption SDK uses the encryption context in the header of the encrypted message that the AWS Encryption SDK returns. Before your application returns plaintext data, verify that the encryption context that you used to encrypt the message is included in the encryption context that was used to decrypt the message. For details, see the examples in your programming language.

When you use the command line interface, the AWS Encryption SDK verifies the encryption context for you.

**Protect your wrapping keys**

The AWS Encryption SDK generates a unique data key to encrypt each plaintext message. Then it encrypts the data key with wrapping keys that you supply. If your wrapping keys are lost or deleted, your encrypted data is unrecoverable. If your keys are not secured, your data might be vulnerable.

Use wrapping keys that are protected by a secure key infrastructure, such as [AWS Key Management Service](https://aws.amazon.com/kms/) (AWS KMS). When using raw AES or raw RSA keys, use a source of randomness and durable storage that meets your security requirements. Generating and storing wrapping keys in a hardware security module (HSM), or a service that provides HSMs, such as AWS CloudHSM, is a best practice.

Use the authorization mechanisms of your key infrastructure to limit access to your wrapping keys to only the users that require it. Implement best practice principles, such as least privilege. When using AWS KMS keys, use key policies and IAM policies that implement [best practice principles](https://aws.amazon.com/articles/877552).

**Specify your wrapping keys**

It's always a best practice to specify your wrapping keys explicitly when decrypting, as well as encrypting. When you do, the AWS Encryption SDK uses only the keys that you specify. This practice assures that you only use the encryption keys that you intend. For AWS KMS wrapping keys, it also improves performance by preventing you from inadvertently using keys in a different AWS account or Region, or attempting to decrypt with keys that you don't have permission to use.

When encrypting, the keyrings and master key providers that the AWS Encryption SDK supplies require that you specify wrapping keys. They use all and only the wrapping keys you specify. You are also required to specify wrapping keys when encrypting and decrypting with raw AES keyrings, raw RSA keyrings, and JCEMasterKeys.
However, when decrypting with AWS KMS keyrings and master key providers, you are not required to specify wrapping keys. The AWS Encryption SDK can get the key identifier from the metadata of the encrypted data key. But specifying wrapping keys is a best practice that we recommend.

To support this best practice when working with AWS KMS wrapping keys, we recommend the following:

- Use AWS KMS keyrings that specify wrapping keys. When encrypting and decrypting, these keyrings use only the specified wrapping keys you specify.

- When using AWS KMS master keys and master key providers, use the strict mode constructors introduced in version 1.7.x of the AWS Encryption SDK. They create providers that encrypt and decrypt only with the wrapping keys you specify. Constructors for master key providers that always decrypt with any wrapping key are deprecated in version 1.7.x and deleted in version 2.0.x.

When specifying AWS KMS wrapping keys for decrypting is impractical, you can use discovery providers. The AWS Encryption SDK in C and JavaScript support AWS KMS discovery keyrings. Master key providers with a discovery mode are available for Java and Python in versions 1.7.x and later. These discovery providers, which are used only for decrypting with AWS KMS wrapping keys, explicitly direct the AWS Encryption SDK to use any wrapping key that encrypted a data key.

If you must use a discovery provider, use its discovery filter features to limit the wrapping keys they use. For example, the AWS KMS regional discovery keyring uses only the wrapping keys in a particular AWS Region. You can also configure AWS KMS keyrings and AWS KMS master key providers to use only the wrapping keys in particular AWS accounts. Also, as always, use key policies and IAM policies to control access to your AWS KMS wrapping keys.

**Use digital signatures**

It's a best practice to use an algorithm suite with signing. Digital signatures verify the message sender was authorized to send the message and protect the integrity of the message. All versions of the AWS Encryption SDK use algorithm suites with signing by default.

If your security requirements don't include digital signatures, you can select an algorithm suite without digital signatures. However, we recommend using digital signatures, especially when one group of users encrypts data and a different set of users decrypts that data.
Use key commitment

It is a best practice to use the key commitment security feature. By verifying the identity of the unique data key that encrypted your data, key commitment prevents you from decrypting any ciphertext that might result in more than one plaintext message.

The AWS Encryption SDK provides full support for encrypting and decrypting with key commitment beginning in version 2.0.x. By default, all of your messages are encrypted and decrypted with key commitment. Version 1.7.x of the AWS Encryption SDK can decrypt ciphertexts with key commitment. It is designed to help users of earlier versions deploy version 2.0.x successfully.

Support for key commitment includes new algorithm suites and a new message format that produces a ciphertext only 30 bytes larger than a ciphertext without key commitment. The design minimizes its impact on performance so most users can enjoy the benefits of key commitment. If your application is very sensitive to size and performance, you might decide to use the commitment policy setting to disable key commitment or allow the AWS Encryption SDK to decrypt messages without commitment, but do so only if you must.

Limit the number of encrypted data keys

It's a best practice to limit the number of encrypted data keys in messages that you decrypt, especially messages from untrusted sources. Decrypting a message with numerous encrypted data keys that you can't decrypt can cause extended delays, run up expenses, throttle your application and others that share your account, and potentially exhaust your key infrastructure. Without limits, an encrypted message can have up to 65,535 (2^16 - 1) encrypted data keys. For details, see Limiting encrypted data keys.

For more information about the AWS Encryption SDK security features that underlie these best practices, see Improved client-side encryption: Explicit KeyIds and key commitment in the AWS Security Blog.
Configuring the AWS Encryption SDK

The AWS Encryption SDK is designed to be easy to use. Although the AWS Encryption SDK has several configuration options, the default values are carefully chosen to be practical and secure for most applications. However, you might need to adjust your configuration to improve performance or include a custom feature in your design.

When configuring your implementation, review the AWS Encryption SDK best practices and implement as many as you can.

Topics

- Selecting a programming language
- Selecting wrapping keys
- Using multi-Region AWS KMS keys
- Choosing an algorithm suite
- Limiting encrypted data keys
- Creating a discovery filter
- Setting a commitment policy
- Working with streaming data
- Caching data keys

Selecting a programming language

The AWS Encryption SDK is available in multiple programming languages. The language implementations are designed to be fully interoperable and to offer the same features, although they might be implemented in different ways. Typically, you use the library that is compatible with your application. However, you might select a programming language for a particular implementation. For example, if you prefer to work with keyrings, you might choose the AWS Encryption SDK for C or the AWS Encryption SDK for JavaScript.

Selecting wrapping keys

The AWS Encryption SDK generates a unique symmetric data key to encrypt each message. Unless you are using data key caching, you don't need to configure, manage, or use the data keys. The AWS Encryption SDK does it for you.
However, you must select one or more wrapping keys to encrypt each data key. The AWS Encryption SDK supports AES symmetric keys and RSA asymmetric keys in different sizes. It also supports AWS Key Management Service (AWS KMS) symmetric encryption AWS KMS keys. You are responsible for the safety and durability of your wrapping keys, so we recommend that you use an encryption key in a hardware security module or a key infrastructure service, such as AWS KMS.

To specify your wrapping keys for encryption and decryption, you use a keyring (C and JavaScript) or a master key provider (Java, Python, AWS Encryption CLI). You can specify one wrapping key or multiple wrapping keys of the same or different types. If you use multiple wrapping keys to wrap a data key, each wrapping key will encrypt a copy of the same data key. The encrypted data keys (one per wrapping key) are stored with the encrypted data in the encrypted message that the AWS Encryption SDK returns. To decrypt the data, the AWS Encryption SDK must first use one of your wrapping keys to decrypt an encrypted data key.

To specify an AWS KMS key in a keyring or master key provider, use a supported AWS KMS key identifier. For details about the key identifiers for an AWS KMS key, see Key Identifiers in the AWS Key Management Service Developer Guide.

- When encrypting with the AWS Encryption SDK for Java, AWS Encryption SDK for JavaScript, AWS Encryption SDK for Python, or the AWS Encryption CLI, you can use any valid key identifier (key ID, key ARN, alias name, or alias ARN) for a KMS key. When encrypting with the AWS Encryption SDK for C, you can only use a key ID or key ARN.

  If you specify an alias name or alias ARN for a KMS key when encrypting, the AWS Encryption SDK saves the key ARN currently associated with that alias; it does not save the alias. Changes to the alias don't affect the KMS key used to decrypt your data keys.

- When decrypting in strict mode (where you specify particular wrapping keys), you must use a key ARN to identify AWS KMS keys. This requirement applies to all language implementations of the AWS Encryption SDK.

  When you encrypt with an AWS KMS keyring, the AWS Encryption SDK stores the key ARN of the AWS KMS key in the metadata of the encrypted data key. When decrypting in strict mode, the AWS Encryption SDK verifies that the same key ARN appears in the keyring (or master key provider) before it attempts to use the wrapping key to decrypt the encrypted data key. If you use a different key identifier, the AWS Encryption SDK will not recognize or use the AWS KMS key, even if the identifiers refer to the same key.
To specify a raw AES key or a raw RSA key pair as a wrapping key in a keyring, you must specify a namespace and a name. In a master key provider, the Provider ID is the equivalent of the namespace and the Key ID is the equivalent of the name. When decrypting, you must use the exact same namespace and name for each raw wrapping key as you used when encrypting. If you use a different namespace or name, the AWS Encryption SDK will not recognize or use the wrapping key, even if the key material is the same.

**Using multi-Region AWS KMS keys**

You can use AWS Key Management Service (AWS KMS) multi-Region keys as wrapping keys in the AWS Encryption SDK. If you encrypt with a multi-Region key in one AWS Region, you can decrypt using a related multi-Region key in a different AWS Region. Support for multi-Region keys is introduced in version 2.3.x of the AWS Encryption SDK and version 3.0.x of the AWS Encryption CLI.

AWS KMS multi-Region keys are a set of AWS KMS keys in different AWS Regions that have the same key material and key ID. You can use these related keys as though they were the same key in different Regions. Multi-Region keys support common disaster recovery and backup scenarios that require encrypting in one Region and decrypting in a different Region without making a cross-Region call to AWS KMS. For information about multi-Region keys, see Using multi-Region keys in the AWS Key Management Service Developer Guide.

To support multi-Region keys, the AWS Encryption SDK includes AWS KMS multi-Region-aware keyrings and master key providers. The new multi-Region-aware symbol in each programming language supports both single-Region and multi-Region keys.

- For single-Region keys, the multi-Region-aware symbol behaves just like the single-Region AWS KMS keyring and master key provider. It attempts to decrypt ciphertext only with the single-Region key that encrypted the data.
- For multi-Region keys, the multi-Region-aware symbol attempts to decrypt ciphertext with the same multi-Region key that encrypted the data or with the related multi-Region key in the Region you specify.

In the multi-Region-aware keyrings and master key providers that take more than one KMS key, you can specify multiple single-Region and multi-Region keys. However, you can specify only one key from each set of related multi-Region keys. If you specify more than one key identifier with the same key ID, the constructor call fails.
You can also use a multi-Region key with the standard, single-Region AWS KMS keyrings and master key providers. However, you must use the same multi-Region key in the same Region to encrypt and decrypt. The single-Region keyrings and master key providers attempt to decrypt ciphertext only with the keys that encrypted the data.

The following examples show how to encrypt and decrypt data using multi-Region keys and the new multi-Region-aware keyrings and master key providers. These examples encrypt data in the us-east-1 Region and decrypt the data in the us-west-2 Region using related multi-Region keys in each Region. Before running these examples, replace the example multi-Region key ARN with a valid value from your AWS account.

C

To encrypt with a multi-Region key, use the
Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder() method to instantiate the keyring. Specify a multi-Region key.

This simple example does not include an encryption context. For an example that uses an encryption context in C, see Encrypting and decrypting strings.

For a complete example, see kms_multi_region_keys.cpp in the AWS Encryption SDK for C repository on GitHub.

```c
/* Encrypt with a multi-Region KMS key in us-east-1 */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Initialize a multi-Region keyring */
const char *mrk_us_east_1 = "arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab";

struct aws_cryptosdk_keyring *mrk_keyring =
    Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder().Build(mrk_us_east_1);

/* Create a session; release the keyring */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(aws_default_allocator(),
    AWS_CRYPTOSDK_ENCRYPT, mrk_keyring);

aws_cryptosdk_keyring_release(mrk_keyring);
```
AWS Encryption SDK

Developer Guide

C# / .NET

To encrypt with a multi-Region key in the US East (N. Virginia) (us-east-1) Region, instantiate a CreateAwsKmsMrkKeyringInput object with a key identifier for the multi-Region key and an AWS KMS client for the specified Region. Then use the CreateAwsKmsMrkKeyring() method to create the keyring.

The CreateAwsKmsMrkKeyring() method creates a keyring with exactly one multi-Region key. To encrypt with multiple wrapping keys, including a multi-Region key, use the CreateAwsKmsMrkMultiKeyring() method.

For a complete example, see AwsKmsMrkKeyringExample.cs in the AWS Encryption SDK for .NET repository on GitHub.

//Encrypt with a multi-Region KMS key in us-east-1 Region

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Multi-Region keys have a distinctive key ID that begins with 'mrk'
// Specify a multi-Region key in us-east-1
string mrkUSEast1 = "arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab";

// Create the keyring
// You can specify the Region or get the Region from the key ARN
var createMrkEncryptKeyringInput = new CreateAwsKmsMrkKeyringInput
    { KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.UEast1), "mrk USEast1"; }
KmsKeyId = mrkUSEast1
};
var mrkEncryptKeyring =
materialProviders.CreateAwsKmsMrkKeyring(createMrkEncryptKeyringInput);

// Define the encryption context
var encryptionContext = new Dictionary<string, string>()
{
    {"purpose", "test"}
};

// Encrypt your plaintext data.
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = mrkEncryptKeyring,
    EncryptionContext = encryptionContext
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);

AWS Encryption CLI

This example encrypts the hello.txt file under a multi-Region key in the us-east-1 Region. Because the example specifies a key ARN with a Region element, this example doesn't use the region attribute of the --wrapping-keys parameter.

When the key ID of the wrapping key doesn't specify a Region, you can use the region attribute of the --wrapping-keys to specify the region, such as --wrapping-keys key=$keyID region=us-east-1.

# Encrypt with a multi-Region KMS key in us-east-1 Region

# To run this example, replace the fictitious key ARN with a valid value.
$ mrkUSEast1=arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab

$ aws-encryption-cli --encrypt \
    --input hello.txt \
    --wrapping-keys key=$mrkUSEast1 \
    --metadata-output ~/metadata \
    --encryption-context purpose=test \
    --output .
Java

To encrypt with a multi-Region key, instantiate an AwsKmsMrkAwareMasterKeyProvider and specify a multi-Region key.

For a complete example, see [BasicMultiRegionKeyEncryptionExample.java](https://github.com/aws/aws-encryption-sdk-java/blob/master/examples/src/main/java/com/amazonaws/cryptoservices/example/BasicMultiRegionKeyEncryptionExample.java) in the AWS Encryption SDK for Java repository on GitHub.

```java
//Encrypt with a multi-Region KMS key in us-east-1 Region

// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

// Multi-Region keys have a distinctive key ID that begins with 'mrk'
// Specify a multi-Region key in us-east-1
final String mrkUSEast1 = "arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab";

// Instantiate an AWS KMS master key provider in strict mode for multi-Region keys
// Configure it to encrypt with the multi-Region key in us-east-1
final AwsKmsMrkAwareMasterKeyProvider kmsMrkProvider =
    AwsKmsMrkAwareMasterKeyProvider.builder()
    .buildStrict(mrkUSEast1);

// Create an encryption context
final Map<String, String> encryptionContext = Collections.singletonMap("Purpose", "Test");

// Encrypt your plaintext data
final CryptoResult<byte[], AwsKmsMrkAwareMasterKey> encryptResult = crypto.encryptData(
    kmsMrkProvider,
    encryptionContext,
    sourcePlaintext);
byte[] ciphertext = encryptResult.getResult();
```
To encrypt with a multi-Region key, use the `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` method to create the keyring and specify a multi-Region key.

For a complete example, see `kms_multi_region_simple.ts` in the AWS Encryption SDK for JavaScript repository on GitHub.

```javascript
/* Encrypt with a multi-Region KMS key in us-east-1 Region */

import {
    buildAwsKmsMrkAwareStrictMultiKeyringBrowser,
    buildClient,
    CommitmentPolicy,
    KMS,
} from '@aws-crypto/client-browser'

/* Instantiate an AWS Encryption SDK client */
const { encrypt } = buildClient(
    CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

declare const credentials: {
    accessKeyId: string
    secretAccessKey: string
    sessionToken: string
}

/* Instantiate an AWS KMS client *
* The AWS Encryption SDK for JavaScript gets the Region from the key ARN *
*/
const clientProvider = (region: string) => new KMS({ region, credentials })

/* Specify a multi-Region key in us-east-1 */
const multiRegionUsEastKey =
    'arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

/* Instantiate the keyring */
const encryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringBrowser({
    generatorKeyId: multiRegionUsEastKey,
})
```
clientProvider,
})

/* Set the encryption context */
const context = {
    purpose: 'test',
}

/* Test data to encrypt */
const cleartext = new Uint8Array([1, 2, 3, 4, 5])

/* Encrypt the data */
const { result } = await encrypt(encryptKeyring, cleartext, {
    encryptionContext: context,
})

JavaScript Node.js

To encrypt with a multi-Region key, use the `buildAwsKmsMrkAwareStrictMultiKeyringNode()` method to create the keyring and specify a multi-Region key.

For a complete example, see `kms_multi_region_simple.ts` in the AWS Encryption SDK for JavaScript repository on GitHub.

//Encrypt with a multi-Region KMS key in us-east-1 Region

import { buildClient } from '@aws-crypto/client-node'

/* Instantiate the AWS Encryption SDK client */
const { encrypt } = buildClient(
    CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* Test string to encrypt */
const cleartext = 'asdf'

/* Multi-Region keys have a distinctive key ID that begins with 'mrk'
* Specify a multi-Region key in us-east-1
*/
const multiRegionUsEastKey =
    'arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'
/* Create an AWS KMS keyring */
const mrkEncryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringNode({
  generatorKeyId: multiRegionUsEastKey,
})

/* Specify an encryption context */
const context = {
  purpose: 'test',
}

/* Create an encryption keyring */
const { result } = await encrypt(mrkEncryptKeyring, cleartext, {
  encryptionContext: context,
})

Python

To encrypt with an AWS KMS multi-Region key, use the MRKAwareStrictAwsKmsMasterKeyProvider() method and specify a multi-Region key.

For a complete example, see mrk_aware_kms_provider.py in the AWS Encryption SDK for Python repository on GitHub.

* Encrypt with a multi-Region KMS key in us-east-1 Region

# Instantiate the client
client =
    aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

# Specify a multi-Region key in us-east-1
mrk_us_east_1 = "arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab"

# Use the multi-Region method to create the master key provider
# in strict mode
strict_mrk_key_provider = MRKAwareStrictAwsKmsMasterKeyProvider(
  key_ids=[mrk_us_east_1]
)

# Set the encryption context
encryption_context = {
  "purpose": "test"
Next, move your ciphertext to the us-west-2 Region. You don't need to re-encrypt the ciphertext.

To decrypt the ciphertext in strict mode in the us-west-2 Region, instantiate the multi-Region-aware symbol with the key ARN of the related multi-Region key in the us-west-2 Region. If you specify the key ARN of a related multi-Region key in a different Region (including us-east-1, where it was encrypted), the multi-Region-aware symbol will make a cross-Region call for that AWS KMS key.

When decrypting in strict mode, the multi-Region-aware symbol requires a key ARN. It accepts only one key ARN from each set of related multi-Region keys.

Before running these examples, replace the example multi-Region key ARN with a valid value from your AWS account.

C

To decrypt in strict mode with a multi-Region key, use the
Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder() method to instantiate the keyring. Specify the related multi-Region key in the local (us-west-2) Region.

For a complete example, see kms_multi_region_keys.cpp in the AWS Encryption SDK for C repository on GitHub.

/* Decrypt with a related multi-Region KMS key in us-west-2 Region */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Initialize a multi-Region keyring */
const char *mrk_us_west_2 = "arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab";
struct aws_cryptosdk_keyring *mrk_keyring =
    Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder().Build(mrk_us_west_2);

/* Create a session; release the keyring */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(aws_default_allocator(),
    AWS_CRYPTOSDK_ENCRYPT, mrk_keyring);

aws_cryptosdk_session_set_commitment_policy(session,
    COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT);

aws_cryptosdk_keyring_release(mrk_keyring);

/* Decrypt the ciphertext
    aws_cryptosdk_session_process_full is designed for non-streaming data
 */
aws_cryptosdk_session_process_full(
    session, plaintext, plaintext_buf_sz, &plaintext_len, ciphertext,
    ciphertext_len));

/* Clean up the session */
aws_cryptosdk_session_destroy(session);

C# / .NET

To decrypt in strict mode with a single multi-Region key, use the same constructors and methods that you used to assemble the input and create the keyring for encrypting. Instantiate a CreateAwsKmsMrkKeyringInput object with the key ARN of a related multi-Region key and an AWS KMS client for the US West (Oregon) (us-west-2) Region. Then use the CreateAwsKmsMrkKeyring() method to create a multi-Region keyring with one multi-Region KMS key.

For a complete example, see AwsKmsMrkKeyringExample.cs in the AWS Encryption SDK for .NET repository on GitHub.

// Decrypt with a related multi-Region KMS key in us-west-2 Region

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();
// Specify the key ARN of the multi-Region key in us-west-2
string mrkUSWest2 = "arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab";

// Instantiate the keyring input
// You can specify the Region or get the Region from the key ARN
var createMrkDecryptKeyringInput = new CreateAwsKmsMrkKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    KmsKeyId = mrkUSWest2
};

// Create the multi-Region keyring
var mrkDecryptKeyring = materialProviders.CreateAwsKmsMrkKeyring(createMrkDecryptKeyringInput);

// Decrypt the ciphertext
var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = mrkDecryptKeyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);

AWS Encryption CLI

To decrypt with the related multi-Region key in the us-west-2 Region, use the key attribute of the --wrapping-keys parameter to specify its key ARN.

# Decrypt with a related multi-Region KMS key in us-west-2 Region

# To run this example, replace the fictitious key ARN with a valid value.
$ mrkUSWest2=arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab

$ aws-encryption-cli --decrypt
    --input hello.txt.encrypted
    --wrapping-keys key=$mrkUSWest2
    --commitment-policy require-encrypt-require-decrypt
    --encryption-context purpose=test
    --metadata-output ~/metadata
    --max-encrypted-data-keys 1

Using multi-Region AWS KMS keys
Java

To decrypt in strict mode, instantiate an `AwsKmsMrkAwareMasterKeyProvider` and specify the related multi-Region key in the local (us-west-2) Region.

For a complete example, see `BasicMultiRegionKeyEncryptionExample.java` in the AWS Encryption SDK for Java repository on GitHub.

```java
// Decrypt with a related multi-Region KMS key in us-west-2 Region

// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

// Related multi-Region keys have the same key ID. Their key ARNs differs only in the Region field.
String mrkUSWest2 = "arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab";

// Use the multi-Region method to create the master key provider
// in strict mode
AwsKmsMrkAwareMasterKeyProvider kmsMrkProvider =
    AwsKmsMrkAwareMasterKeyProvider.builder()
    .buildStrict(mrkUSWest2);

// Decrypt your ciphertext
CryptoResult<byte[], AwsKmsMrkAwareMasterKey> decryptResult = crypto.decryptData(
    kmsMrkProvider,
    ciphertext);
byte[] decrypted = decryptResult.getResult();
```

JavaScript Browser

To decrypt in strict mode, use the `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` method to create the keyring and specify the related multi-Region key in the local (us-west-2) Region.

For a complete example, see `kms_multi_region_simple.ts` in the AWS Encryption SDK for JavaScript repository on GitHub.
/* Decrypt with a related multi-Region KMS key in us-west-2 Region */

import {
    buildAwsKmsMrkAwareStrictMultiKeyringBrowser,
    buildClient,
    CommitmentPolicy,
    KMS,
} from '@aws-crypto/client-browser'

/* Instantiate an AWS Encryption SDK client */
const { decrypt } = buildClient(
    CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

declare const credentials: {
    accessKeyId: string
    secretAccessKey: string
    sessionToken: string
}

/* Instantiate an AWS KMS client
  * The AWS Encryption SDK for JavaScript gets the Region from the key ARN
  */
const clientProvider = (region: string) => new KMS({ region, credentials })

/* Specify a multi-Region key in us-west-2 */
const multiRegionUsWestKey =
    'arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

/* Instantiate the keyring */
const mrkDecryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringBrowser({
    generatorKeyId: multiRegionUsWestKey,
    clientProvider,
})

/* Decrypt the data */
const { plaintext, messageHeader } = await decrypt(mrkDecryptKeyring, result)
To decrypt in strict mode, use the `buildAwsKmsMrkAwareStrictMultiKeyringNode()` method to create the keyring and specify the related multi-Region key in the local (us-west-2) Region.

For a complete example, see `kms_multi_region_simple.ts` in the AWS Encryption SDK for JavaScript repository on GitHub.

```javascript
/* Decrypt with a related multi-Region KMS key in us-west-2 Region */
import { buildClient } from '@aws-crypto/client-node'

/* Instantiate the client */
const { decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* Multi-Region keys have a distinctive key ID that begins with 'mrk' */
const multiRegionUsWestKey =
  'arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

/* Create an AWS KMS keyring */
const mrkDecryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringNode({
  generatorKeyId: multiRegionUsWestKey,
})

/* Decrypt your ciphertext */
const { plaintext, messageHeader } = await decrypt(decryptKeyring, result)
```

Python

To decrypt in strict mode, use the `MRKAwareStrictAwsKmsMasterKeyProvider()` method to create the master key provider. Specify the related multi-Region key in the local (us-west-2) Region.

For a complete example, see `mrk_aware_kms_provider.py` in the AWS Encryption SDK for Python repository on GitHub.

```python
# Decrypt with a related multi-Region KMS key in us-west-2 Region
```
# Instantiate the client
client = aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

# Related multi-Region keys have the same key ID. Their key ARNs differs only in the Region field
mrk_us_west_2 = "arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab"

# Use the multi-Region method to create the master key provider
# in strict mode
strict_mrk_key_provider = MRKAwareStrictAwsKmsMasterKeyProvider(
    key_ids=[mrk_us_west_2]
)

# Decrypt your ciphertext
plaintext, _ = client.decrypt(
    source=ciphertext,
    key_provider=strict_mrk_key_provider
)

You can also decrypt in discovery mode with AWS KMS multi-Region keys. When decrypting in discovery mode, you don't specify any AWS KMS keys. (For information about single-Region AWS KMS discovery keyrings, see Using an AWS KMS discovery keyring.)

If you encrypted with a multi-Region key, the multi-Region-aware symbol in discovery mode will try to decrypt by using a related multi-Region key in the local Region. If none exists; the call fails. In discovery mode, the AWS Encryption SDK will not attempt a cross-Region call for the multi-Region key used for encryption.

Note
If you use a multi-Region-aware symbol in discovery mode to encrypt data, the encrypt operation fails.

The following example shows how to decrypt with the multi-Region-aware symbol in discovery mode. Because you don't specify an AWS KMS key, the AWS Encryption SDK must get the Region from a different source. When possible, specify the local Region explicitly. Otherwise, the AWS
Encryption SDK gets the local Region from the Region configured in the AWS SDK for your programming language.

Before running these examples, replace the example account ID and multi-Region key ARN with valid values from your AWS account.

C

To decrypt in discovery mode with a multi-Region key, use the `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` method to build the keyring, and the `Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder()` method to build the discovery filter. To specify the local Region, define a `ClientConfiguration` and specify it in the AWS KMS client.

For a complete example, see `kms_multi_region_keys.cpp` in the AWS Encryption SDK for C repository on GitHub.

```c
/* Decrypt in discovery mode with a multi-Region KMS key */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct a discovery filter for the account and partition. The
* filter is optional, but it's a best practice that we recommend. */
const char *account_id = "111122223333";
const char *partition = "aws";
const std::shared_ptr<Aws::Cryptosdk::KmsKeyring::DiscoveryFilter> discovery_filter =
    Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder(partition).AddAccount(account_id).Build();

/* Create an AWS KMS client in the desired region. */
const char *region = "us-west-2";

Aws::Client::ClientConfiguration client_config;
client_config.region = region;
const std::shared_ptr<Aws::KMS::KMSClient> kms_client =
    Aws::MakeShared<Aws::KMS::KMSClient>("AWS_SAMPLE_CODE", client_config);

struct aws_cryptosdk_keyring *mrk_keyring =
    Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()
        .WithKmsClient(kms_client)
```
```c
.BuildDiscovery(region, discovery_filter);

/* Create a session; release the keyring */
struct aws_cryptosdk_session *session =
  aws_cryptosdk_session_new_from_keyring_2(aws_default_allocator(),
  AWS_CRYPTOSDK_DECRYPT, mrk_keyring);

aws_cryptosdk_keyring_release(mrk_keyring);
commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
/* Decrypt the ciphertext
 *   aws_cryptosdk_session_process_full is designed for non-streaming data
 */
aws_cryptosdk_session_process_full(
  session, plaintext, plaintext_buf_sz, &plaintext_len, ciphertext,
  ciphertext_len));

/* Clean up the session */
aws_cryptosdk_session_destroy(session);
```

**C# / .NET**

To create a multi-Region-aware discovery keyring in the AWS Encryption SDK for .NET, instantiate a `CreateAwsKmsMrkDiscoveryKeyringInput` object that takes an AWS KMS client for a particular AWS Region, and an optional discovery filter that limits KMS keys to a particular AWS partition and account. Then call the `CreateAwsKmsMrkDiscoveryKeyring()` method with the input object. For a complete example, see [AwsKmsMrkDiscoveryKeyringExample.cs](https://github.com/aws/aws-encryption-library-dotnet/blob/master/samples/AwsKmsMrkDiscoveryKeyringExample.cs) in the AWS Encryption SDK for .NET repository on GitHub.

To create a multi-Region-aware discovery keyring for more than one AWS Region, use the `CreateAwsKmsMrkDiscoveryMultiKeyring()` method to create a multi-keyring, or use `CreateAwsKmsMrkDiscoveryKeyring()` to create several multi-Region-aware discovery keyrings and then use the `CreateMultiKeyring()` method to combine them in a multi-keyring.

For an example, see [AwsKmsMrkDiscoveryMultiKeyringExample.cs](https://github.com/aws/aws-encryption-library-dotnet/blob/master/samples/AwsKmsMrkDiscoveryMultiKeyringExample.cs).

```csharp
// Decrypt in discovery mode with a multi-Region KMS key

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =
```
```csharp
AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

List<string> account = new List<string> { "111122223333" };

// Instantiate the discovery filter
DiscoveryFilter mrkDiscoveryFilter = new DiscoveryFilter()
{
    AccountIds = account,
    Partition = "aws"
}

// Create the keyring
var createMrkDiscoveryKeyringInput = new CreateAwsKmsMrkDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    DiscoveryFilter = mrkDiscoveryFilter
};
var mrkDiscoveryKeyring = materialProviders.CreateAwsKmsMrkDiscoveryKeyring(createMrkDiscoveryKeyringInput);

// Decrypt the ciphertext
var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = mrkDiscoveryKeyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

### AWS Encryption CLI

To decrypt in discovery mode, use the **discovery** attribute of the `--wrapping-keys` parameter. The **discovery-account** and **discovery-partition** attributes create a discovery filter that is optional, but recommended.

To specify the Region, this command includes the **region** attribute of the `--wrapping-keys` parameter.

```shell
# Decrypt in discovery mode with a multi-Region KMS key

$ aws-encryption-cli --decrypt
    --input hello.txt.encrypted
```

Using multi-Region AWS KMS keys
Java

To specify the local Region, use the `builder().withDiscoveryMrkRegion` parameter. Otherwise, the AWS Encryption SDK gets the local Region from the Region configured in the AWS SDK for Java.

For a complete example, see `DiscoveryMultiRegionDecryptionExample.java` in the AWS Encryption SDK for Java repository on GitHub.

```java
// Decrypt in discovery mode with a multi-Region KMS key

// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

DiscoveryFilter discoveryFilter = new DiscoveryFilter("aws", 11112222333);

AwsKmsMrkAwareMasterKeyProvider mrkDiscoveryProvider =
    AwsKmsMrkAwareMasterKeyProvider
    .builder()
    .withDiscoveryMrkRegion(Region.US_WEST_2)
    .buildDiscovery(discoveryFilter);

// Decrypt your ciphertext
final CryptoResult<byte[], AwsKmsMrkAwareMasterKey> decryptResult = crypto
    .decryptData(mrkDiscoveryProvider, ciphertext);
```

JavaScript Browser

To decrypt in discovery mode with a symmetric multi-Region key, use the `AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser()` method.
For a complete example, see `kms_multi_region_discovery.ts` in the AWS Encryption SDK for JavaScript repository on GitHub.

```javascript
/* Decrypt in discovery mode with a multi-Region KMS key */

import {
  AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser,
  buildClient,
  CommitmentPolicy,
  KMS,
} from '@aws-crypto/client-browser'

/* Instantiate an AWS Encryption SDK client */
const { decrypt } = buildClient()

declare const credentials: {
  accessKeyId: string
  secretAccessKey: string
  sessionToken: string
}

/* Instantiate the KMS client with an explicit Region */
const client = new KMS({ region: 'us-west-2', credentials })

/* Create a discovery filter */
const discoveryFilter = { partition: 'aws', accountIDs: ['111122223333'] }

/* Create an AWS KMS discovery keyring */
const mrkDiscoveryKeyring = new AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser({
  client,
  discoveryFilter,
})

/* Decrypt the data */
const { plaintext, messageHeader } = await decrypt(mrkDiscoveryKeyring, ciphertext)
```
To decrypt in discovery mode with a symmetric multi-Region key, use the
`AwsKmsMrkAwareSymmetricDiscoveryKeyringNode()` method.

For a complete example, see `kms_multi_region_discovery.ts` in the AWS Encryption SDK for
JavaScript repository on GitHub.

```javascript
/* Decrypt in discovery mode with a multi-Region KMS key */
import {
    AwsKmsMrkAwareSymmetricDiscoveryKeyringNode,
    buildClient,
    CommitmentPolicy,
    KMS,
} from '@aws-crypto/client-node'

/* Instantiate the Encryption SDK client
const { decrypt } = buildClient()

/* Instantiate the KMS client with an explicit Region */
const client = new KMS({ region: 'us-west-2' })

/* Create a discovery filter */
const discoveryFilter = { partition: 'aws', accountIDs: ['111122223333'] }

/* Create an AWS KMS discovery keyring */
const mrkDiscoveryKeyring = new AwsKmsMrkAwareSymmetricDiscoveryKeyringNode({
    client,
    discoveryFilter,
})

/* Decrypt your ciphertext */
const { plaintext, messageHeader } = await decrypt(mrkDiscoveryKeyring, result)
```

Python

To decrypt in discovery mode with a multi-Region key, use the
`MRKAwareDiscoveryAwsKmsMasterKeyProvider()` method.

For a complete example, see `mrk_aware_kms_provider.py` in the AWS Encryption SDK for
Python repository on GitHub.
Choosing an algorithm suite

The AWS Encryption SDK supports several symmetric and asymmetric encryption algorithms for encrypting your data keys under the wrapping keys you specify. However, when it uses those data keys to encrypt your data, the AWS Encryption SDK defaults to a recommended algorithm suite that uses the AES-GCM algorithm with key derivation, digital signatures, and key commitment. Although the default algorithm suite is likely to be suitable for most applications, you can choose an alternate algorithm suite. For example, some trust models would be satisfied by an algorithm suite without digital signatures. For information about the algorithm suites that the AWS Encryption SDK supports, see Supported algorithm suites in the AWS Encryption SDK.

The following examples show you how to select an alternate algorithm suite when encrypting. These examples select a recommended AES-GCM algorithm suite with key derivation and key commitment, but without digital signatures. When you encrypt with an algorithm suite that does not include digital signatures, use the unsigned-only decryption mode when decrypting. This mode, which fails if it encounters a signed ciphertext, is most useful when streaming decryption.
To specify an alternate algorithm suite in the AWS Encryption SDK for C, you must create a CMM explicitly. Then use the `aws_cryptosdk_default_cmm_set_alg_id` with the CMM and the selected algorithm suite.

```c
/* Specify an algorithm suite without signing */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct an AWS KMS keyring */
struct aws_cryptosdk_keyring *kms_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);

/* To set an alternate algorithm suite, create an cryptographic
   materials manager (CMM) explicitly */
struct aws_cryptosdk_cmm *cmm =
    aws_cryptosdk_default_cmm_new(aws_default_allocator(), kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);

/* Specify the algorithm suite for the CMM */
aws_cryptosdk_default_cmm_set_alg_id(cmm, ALG_AES256_GCM_HKDF_SHA512_COMMIT_KEY);

/* Construct the session with the CMM,
   then release the CMM reference */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_cmm_2(alloc,
        AWS_CRYPTOSDK_ENCRYPT, cmm);
aws_cryptosdk_session_release(session);

/* Encrypt the data
   Use aws_cryptosdk_session_process_full with non-streaming data */
if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(
    session,
    ciphertext,
    ciphertext_buf_sz,
    ciphertext_len,
    plaintext,
    plaintext_len)) {
    aws_cryptosdk_session_destroy(session);
}
```
When decrypting data that was encrypted without digital signatures, use
AWS_CRYPTOSDK_DECRYPT_UNSIGNED. This causes the decrypt to fail if it encounters signed
ciphertext.

```c
/* Decrypt unsigned streaming data */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct an AWS KMS keyring */
struct aws_cryptosdk_keyring *kms_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);

/* Create a session for decrypting with the AWS KMS keyring
   Then release the keyring reference */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(aloc, AWS_CRYPTOSDK_DECRYPT_UNSIGNED,
    kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);

if (!session) {
    return AWS_OP_ERR;
}

/* Limit encrypted data keys */
aws_cryptosdk_session_set_max_encrypted_data_keys(session, 1);

/* Decrypt
   Use aws_cryptosdk_session_process_full with non-streaming data */
    if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(
    session,
    plaintext,
    plaintext_buf_sz,
    &plaintext_len,
    ciphertext,
    ciphertext_len)) {
        aws_cryptosdk_session_destroy(session);
```
C# / .NET

To specify an alternate algorithm suite in the AWS Encryption SDK for .NET, specify the `AlgorithmSuiteId` property of an `EncryptInput` object. The AWS Encryption SDK for .NET includes constants that you can use to identify your preferred algorithm suite.

The AWS Encryption SDK for .NET doesn't have a method to detect signed ciphertext when streaming decryption because this library doesn't support streaming data.

```
// Specify an algorithm suite without signing

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Create the keyring
var keyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var keyring = materialProviders.CreateAwsKmsKeyring(keyringInput);

// Encrypt your plaintext data
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    AlgorithmSuiteId = AlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

AWS Encryption CLI

When encrypting the `hello.txt` file, this example uses the `--algorithm` parameter to specify an algorithm suite without digital signatures.

```
# Specify an algorithm suite without signing
```

Choosing an algorithm suite
AWS Encryption SDK

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

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

$ aws-encryption-cli --encrypt
   --input hello.txt
   --wrapping-keys key=$keyArn
   --algorithm AES_256_GCM_HKDF_SHA512_COMMIT_KEY
   --metadata-output ~/metadata
   --encryption-context purpose=test
   --commitment-policy require-encrypt-require-decrypt
   --output hello.txt.encrypted
   --decode
```

When decrypting, this example uses the `--decrypt-unsigned` parameter. This parameter is recommended to ensure that you are decrypting unsigned ciphertext, especially with the CLI, which is always streaming input and output.

```bash
# Decrypt unsigned streaming data

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

$ aws-encryption-cli --decrypt-unsigned
   --input hello.txt.encrypted
   --wrapping-keys key=$keyArn
   --max-encrypted-data-keys 1
   --commitment-policy require-encrypt-require-decrypt
   --encryption-context purpose=test
   --metadata-output ~/metadata
   --output .
```

Java

To specify an alternate algorithm suite, use the `AwsCrypto.builder().withEncryptionAlgorithm()` method. This example specifies an alternate algorithm suite without digital signatures.

```java
// Specify an algorithm suite without signing

// Instantiate the client
AwsCrypto crypto = AwsCrypto.builder()
```
 AWS Encryption SDK

Developer Guide

.withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
.withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
.build();

String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Create a master key provider in strict mode
KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

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

// Encrypt your plaintext data
CryptoResult<byte[], KmsMasterKey> encryptResult = crypto.encryptData(
    masterKeyProvider,
    sourcePlaintext,
    encryptionContext);
byte[] ciphertext = encryptResult.getResult();

When streaming data for decryption, use the createUnsignedMessageDecryptingStream() method to ensure that all ciphertext that you're decrypting is unsigned.

// Decrypt unsigned streaming data

// Instantiate the client
AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .withMaxEncryptedDataKeys(1)
    .build();

// Create a master key provider in strict mode
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

// Decrypt the encrypted message
FileInputStream in = new FileInputStream(srcFile + ".encrypted");
CryptoInputStream<KmsMasterKey> decryptingStream =
crypto.createUnsignedMessageDecryptingStream(masterKeyProvider, in);

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

JavaScript Browser

To specify an alternate algorithm suite, use the suiteId parameter with an AlgorithmSuiteIdentifier enum value.

// Specify an algorithm suite without signing
// Instantiate the client
const { encrypt } = buildClient( CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT )

// Specify a KMS key
const generatorKeyId = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Create a keyring with the KMS key
const keyring = new KmsKeyringBrowser({ generatorKeyId })

// Encrypt your plaintext data
const { result } = await encrypt(keyring, cleartext, { suiteId: AlgorithmSuiteIdentifier.ALG_AES256_GCM_IV12_TAG16_HKDF_SHA512_COMMIT_KEY,
encryptionContext: context, })

When decrypting, use the standard decrypt method. AWS Encryption SDK for JavaScript in the browser doesn't have a decrypt-unsigned mode because the browser doesn't support streaming.

// Decrypt unsigned streaming data

// Instantiate the client
const { decrypt } = buildClient( CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT )

// Create a keyring with the same KMS key used to encrypt
const generatorKeyId = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
const keyring = new KmsKeyringBrowser({ generatorKeyId })

// Decrypt the encrypted message
const { plaintext, messageHeader } = await decrypt(keyring, ciphertextMessage)

JavaScript Node.js

To specify an alternate algorithm suite, use the suiteId parameter with an AlgorithmSuiteIdentifier enum value.

// Specify an algorithm suite without signing

// Instantiate the client
const { encrypt } = buildClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

// Specify a KMS key
const generatorKeyId = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Create a keyring with the KMS key
const keyring = new KmsKeyringNode({ generatorKeyId })

// Encrypt your plaintext data
const { result } = await encrypt(keyring, cleartext, { suiteId: AlgorithmSuiteIdentifier.ALG_AES256_GCM_IV12_TAG16_HKDF_SHA512_COMMIT_KEY, encryptionContext: context, })

When decrypting data that was encrypted without digital signatures, use decryptUnsignedMessageStream. This method fails if it encounters signed ciphertext.

// Decrypt unsigned streaming data

// Instantiate the client
const { decryptUnsignedMessageStream } = buildClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

// Create a keyring with the same KMS key used to encrypt
const generatorKeyId = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
const keyring = new KmsKeyringNode({ generatorKeyId })

// Decrypt the encrypted message
const outputStream =
createReadStream(filename) .pipe(decryptUnsignedMessageStream(keyring))

Python

To specify an alternate encryption algorithm, use the algorithm parameter with an Algorithm enum value.

```python
# Specify an algorithm suite without signing

# Instantiate a client
client =
    aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT,
        max_encrypted_data_keys=1)

# Create a master key provider in strict mode
aws_kms_key = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
aws_kms_strict_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[aws_kms_key]
)

# Encrypt the plaintext using an alternate algorithm suite
ciphertext, encrypted_message_header = client.encrypt(
    algorithm=Algorithm.AES_256_GCM_HKDF_SHA512_COMMIT_KEY, source=source_plaintext,
    key_provider=kms_key_provider
)

When decrypting messages that were encrypted without digital signatures, use the decrypt-unsigned streaming mode, especially when decrypting while streaming.

```python
# Decrypt unsigned streaming data

# Instantiate the client
client =
    aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT,
        max_encrypted_data_keys=1)

# Create a master key provider in strict mode
aws_kms_key = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
aws_kms_strict_master_key_provider = StrictAwsKmsMasterKeyProvider(
# Decrypt with decrypt-unsigned

```python
with open(ciphertext_filename, "rb") as ciphertext, open(cycled_plaintext_filename, "wb") as plaintext:
    with client.stream(mode="decrypt-unsigned",
                       source=ciphertext,
                       key_provider=master_key_provider) as decryptor:
        for chunk in decryptor:
            plaintext.write(chunk)

# Verify that the encryption context
assert all(
    pair in decryptor.header.encryption_context.items() for pair in
    encryptor.header.encryption_context.items())

return ciphertext_filename, cycled_plaintext_filename
```

## Limiting encrypted data keys

You can limit the number of encrypted data keys in an encrypted message. This best practice feature can help you detect a misconfigured keyring when encrypting or a malicious ciphertext when decrypting. It also prevents unnecessary, expensive, and potentially exhaustive calls to your key infrastructure. Limiting encrypted data keys is most valuable when you are decrypting messages from an untrusted source.

Although most encrypted messages have one encrypted data key for each wrapping key used in the encryption, an encrypted message can contain up to 65,535 encrypted data keys. A malicious actor might construct an encrypted message with thousands of encrypted data keys, none of which can be decrypted. As a result, the AWS Encryption SDK would attempt to decrypt each encrypted data key until it exhausted the encrypted data keys in the message.

To limit encrypted data keys, use the `MaxEncryptedDataKeys` parameter. This parameter is available for all supported programming languages beginning in versions 1.9.x and 2.2.x of the AWS Encryption SDK. It is optional and valid when encrypting and decrypting. The following examples decrypt data that was encrypted under three different wrapping keys. The `MaxEncryptedDataKeys` value is set to 3.
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct an AWS KMS keyring */
struct aws_cryptosdk_keyring *kms_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn1, { key_arn2, key_arn3 });

/* Create a session */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_DECRYPT,
        kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);

/* Limit encrypted data keys */
aws_cryptosdk_session_set_max_encrypted_data_keys(session, 3);

/* Decrypt */
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);
assert(aws_cryptosdk_session_is_done(session));
assert(ciphertext_consumed == ciphertext_len);

C# / .NET

To limit encrypted data keys in the AWS Encryption SDK for .NET, instantiate a client for the AWS Encryption SDK for .NET and set its optional MaxEncryptedDataKeys parameter to the desired value. Then, call the Decrypt() method on the configured AWS Encryption SDK instance.

// Decrypt with limited data keys

// Instantiate the material providers
var materialProviders =
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();
// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig{
    MaxEncryptedDataKeys = 3
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

// Create the keyring
string keyArn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
var createKeyringInput = new CreateAwsKmsKeyringInput{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var decryptKeyring = materialProviders.CreateAwsKmsKeyring(createKeyringInput);

// Decrypt the ciphertext
var decryptInput = new DecryptInput{
    Ciphertext = ciphertext,
    Keyring = decryptKeyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);

AWS Encryption CLI

# Decrypt with limited encrypted data keys

$ aws-encryption-cli --decrypt \
    --input hello.txt.encrypted \
    --wrapping-keys key=$key_arn1 key=$key_arn2 key=$key_arn3 \
    --buffer \
    --max-encrypted-data-keys 3 \
    --encryption-context purpose=test \
    --metadata-output ~/metadata \
    --output .

Java

// Construct a client with limited encrypted data keys
final AwsCrypto crypto = AwsCrypto.builder()
withMaxEncryptedDataKeys(3)
.build();

// Create an AWS KMS master key provider
final KmsMasterKeyProvider keyProvider = KmsMasterKeyProvider.builder()
  .buildStrict(keyArn1, keyArn2, keyArn3);

// Decrypt
final CryptoResult<byte[], KmsMasterKey> decryptResult =
crypto.decryptData(keyProvider, ciphertext)

JavaScript Browser

// Construct a client with limited encrypted data keys
const { encrypt, decrypt } = buildClient({
  maxEncryptedDataKeys: 3
});

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

// Create an AWS KMS keyring
const keyring = new KmsKeyringBrowser({
  clientProvider,
  keyIds: [keyArn1, keyArn2, keyArn3],
});

// Decrypt
const { plaintext, messageHeader } = await decrypt(keyring, ciphertext)

JavaScript Node.js

// Construct a client with limited encrypted data keys
const { encrypt, decrypt } = buildClient({
  maxEncryptedDataKeys: 3
});

// Create an AWS KMS keyring
const keyring = new KmsKeyringBrowser({
  keyIds: [keyArn1, keyArn2, keyArn3],
});
Creating a discovery filter

When decrypting data encrypted with KMS keys, it's a best practice to decrypt in *strict mode*, that is, to limit the wrapping keys used to only those that you specify. However, if necessary, you can also decrypt in *discovery mode*, where you don't specify any wrapping keys. In this mode, AWS KMS can decrypt the encrypted data key using the KMS key that encrypted it, regardless of who owns or has access to that KMS key.

If you must decrypt in discovery mode, we recommend that you always use a *discovery filter*, which limits the KMS keys that can be used to those in a specified AWS account and *partition*. The discovery filter is optional, but it's a best practice.

Use the following table to determine the partition value for your discovery filter.

<table>
<thead>
<tr>
<th>Region</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS Regions</td>
<td>aws</td>
</tr>
<tr>
<td>China Regions</td>
<td>aws-cn</td>
</tr>
<tr>
<td>AWS GovCloud (US) Regions</td>
<td>aws-us-gov</td>
</tr>
</tbody>
</table>
The examples in this section show how to create a discovery filter. Before using the code, replace the example values with valid values for the AWS account and partition.

C

For a complete examples, see `kms_discovery.cpp` in the AWS Encryption SDK for C.

```c
/* Create a discovery filter for an AWS account and partition */

const char *account_id = "111122223333";
const char *partition = "aws";
const std::shared_ptr<Aws::Cryptosdk::KmsKeyring::DiscoveryFilter> discovery_filter
    =
    Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder(partition).AddAccount(account_id).Build();
```

C#/.NET

For a complete example, see `DiscoveryFilterExample.cs` in the AWS Encryption SDK for .NET.

```csharp
// Create a discovery filter for an AWS account and partition

List<string> account = new List<string> { "111122223333" };

DiscoveryFilter exampleDiscoveryFilter = new DiscoveryFilter()
{
    AccountIds = account,
    Partition = "aws"
}
```

AWS Encryption CLI

# Decrypt in discovery mode with a discovery filter

```
$ aws-encryption-cli --decrypt \
    --input hello.txt.encrypted \n    --wrapping-keys discovery=true \n    discovery-account=111122223333 \n    discovery-partition=aws \n    --encryption-context purpose=test \n    --metadata-output ~/metadata \n    --max-encrypted-data-keys 1 \
```
Java

For a complete example, see DiscoveryDecryptionExample.java in the AWS Encryption SDK for Java.

```java
// Create a discovery filter for an AWS account and partition
DiscoveryFilter discoveryFilter = new DiscoveryFilter("aws", 111122223333);
```

JavaScript (Node and Browser)

For complete examples, see kms_filtered_discovery.ts (Node.js) and kms_multi_region_discovery.ts (Browser) in the AWS Encryption SDK for JavaScript.

```javascript
/* Create a discovery filter for an AWS account and partition */
const discoveryFilter = {
  accountIDs: ['111122223333'],
  partition: 'aws',
}
```

Python

For a complete example, see discovery_kms_provider.py in the AWS Encryption SDK for Python.

```python
# Create the discovery filter and specify the region
decrypt_kwargs = dict(
    discovery_filter=DiscoveryFilter(account_ids="111122223333",
    partition="aws"),
    discovery_region="us-west-2",
)
```

**Setting a commitment policy**

A commitment policy is a configuration setting that determines whether your application encrypts and decrypts with key commitment. Encrypting and decrypting with key commitment is an AWS Encryption SDK best practice.
Setting and adjusting your commitment policy is a critical step in migrating from versions 1.7.x and earlier of the AWS Encryption SDK to version 2.0.x and later. This progression is explained in detail in the migration topic.

The default commitment policy value in the latest versions of the AWS Encryption SDK (beginning in version 2.0.x), RequireEncryptRequireDecrypt, is ideal for most situations. However, if you need to decrypt ciphertext that was encrypted without key commitment, you might need to change your commitment policy to RequireEncryptAllowDecrypt. For examples of how to set a commitment policy in each programming language, see Setting your commitment policy.

**Working with streaming data**

When you stream data for decryption, be aware that the AWS Encryption SDK returns decrypted plaintext after the integrity checks are complete, but before the digital signature is verified. To ensure that you don't return or use plaintext until the signature is verified, we recommend that you buffer the streamed plaintext until the entire decryption process is complete.

This issue arises only when you are streaming ciphertext for decryption, and only when you are using an algorithm suite, such as the default algorithm suite, that includes digital signatures.

To make the buffering easier, some AWS Encryption SDK language implementations, such as AWS Encryption SDK for JavaScript in Node.js, include a buffering feature as part of the decrypt method. The AWS Encryption CLI, which always streams input and output introduced a --buffer parameter in versions 1.9.x and 2.2.x. In other language implementations, you can use existing buffering features. (The AWS Encryption SDK for .NET does not support streaming.)

If you are using an algorithm suite without digital signatures, be sure to use the decrypt-unsigned feature in each language implementation. This feature decrypts ciphertext but fails if it encounters signed ciphertext. For details, see Choosing an algorithm suite.

**Caching data keys**

In general, reusing data keys is discouraged, but the AWS Encryption SDK offers a data key caching option that provides limited reuse of data keys. Data key caching can improve the performance of some applications and reduce calls to your key infrastructure. Before using data key caching in production, adjust the security thresholds, and test to make sure that the benefits outweigh the disadvantages of reusing data keys.
Using keyrings

The AWS Encryption SDK for C, the AWS Encryption SDK for JavaScript, the AWS Encryption SDK for Java, and the AWS Encryption SDK for .NET use keyrings to perform envelope encryption. Keyrings generate, encrypt, and decrypt data keys. Keyrings determine the source of the unique data keys that protect each message, and the wrapping keys 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. 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 KMS keys that never leave AWS Key Management Service (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.

Keyrings play the role of master keys and master key providers in the AWS Encryption SDK for Java, AWS Encryption SDK for Python, and the AWS Encryption CLI. If you use different language implementations of the AWS Encryption SDK to encrypt and decrypt your data, be sure to use compatible keyrings and master key providers. For details, see Keyring compatibility.

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 and JavaScript topics.

Topics

- How keyrings work
- Keyring compatibility
- Choosing a keyring
How keyrings work

When you encrypt data, the AWS Encryption SDK asks the keyring for encryption materials. The keyring returns a plaintext data key and a copy of the data 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 then destroys the plaintext data key. Then, the AWS Encryption SDK returns an encrypted message that includes the encrypted data keys and the encrypted data.

![Diagram showing how keyrings work]

When you decrypt data, you can use the same keyring that you used to encrypt the data, or a different one. To decrypt the data, a decryption keyring must include (or have access to) at least one wrapping key in the encryption keyring.

The AWS Encryption SDK passes the encrypted data keys from the encrypted message to the keyring, 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. 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 with any one of the wrapping keys in the multi-keyring.

**Keyring compatibility**

Although the different language implementations of the AWS Encryption SDK have some architectural differences, they are fully compatible, subject to language constraints. You can encrypt your data using one language implementation and decrypt it with any other language implementation. However, you must use the same or corresponding wrapping 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” in the AWS Encryption SDK for JavaScript topic.
Varying requirements for encryption keyrings

In AWS Encryption SDK language implementations other than the AWS Encryption SDK for C, all wrapping keys in an encryption keyring (or multi-keyring) or master key provider must be able to encrypt the data key. If any wrapping key fails to encrypt, the encrypt method fails. As a result, the caller must have the required permissions for all keys in the keyring. If you use a discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails.

The exception is the AWS Encryption SDK for C, where the encrypt operation ignores a standard discovery keyring, but fails if you specify a multi-Region discovery keyring, alone or in a multi-keyring.

Compatible Keyrings and Master Key Providers

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

<table>
<thead>
<tr>
<th>Keyring:</th>
<th>Master Key Provider:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS KMS keyring</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>
</tbody>
</table>

Note

The AWS Encryption SDK for Python and AWS Encryption SDK for Java don't include a master key or master key provider that is equivalent to the AWS KMS regional discovery keyring.

<table>
<thead>
<tr>
<th>Keyring:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS KMS Hierarchical keyring</td>
<td>Only available with version 4.x of the AWS Encryption SDK for .NET and version 3.x of the AWS Encryption SDK for Java.</td>
</tr>
<tr>
<td>Raw AES keyring</td>
<td>When they are used with symmetric encryption keys:</td>
</tr>
</tbody>
</table>
### Keyring: Master Key Provider:

<table>
<thead>
<tr>
<th>Keyring</th>
<th>Master Key Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>JceMasterKey (Java)</td>
<td></td>
</tr>
<tr>
<td>RawMasterKey (Python)</td>
<td></td>
</tr>
</tbody>
</table>

### Raw RSA keyring

When they are used with asymmetric encryption keys:

<table>
<thead>
<tr>
<th>Master Key Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>JceMasterKey (Java)</td>
</tr>
<tr>
<td>RawMasterKey (Python)</td>
</tr>
</tbody>
</table>

**Note**

The Raw RSA keyring does not support asymmetric KMS keys. If you want to use asymmetric RSA KMS keys, version 4.x of the AWS Encryption SDK for .NET supports AWS KMS keyrings that use symmetric encryption (SYMMETRIC_DEFAULT) or asymmetric RSA AWS KMS keys.

## Choosing a keyring

Your keyring determines the wrapping keys that protect your data keys, and ultimately, your data. Use the most secure wrapping keys that are practical for your task. Whenever possible use wrapping keys that are protected by a hardware security module or a key management infrastructure, such as KMS keys in AWS Key Management Service (AWS KMS) or encryption keys AWS CloudHSM.

The AWS Encryption SDK provides several keyrings and keyring configurations in multiple programming languages, and you can create your own custom keyrings. You can also create a multi-keyring that includes one or more keyrings of the same or a different type.

### Topics

- AWS KMS keyrings
- AWS KMS Hierarchical keyrings
- Raw AES keyrings
- Raw RSA keyrings
Multi-keyrings

AWS KMS keyrings

An AWS KMS keyring uses symmetric encryption AWS KMS keys to generate, encrypt, and decrypt data keys. AWS Key Management Service (AWS KMS) protects your KMS 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.

You can use an AWS KMS multi-Region key in an AWS KMS keyring or master key provider beginning in version 2.3.x of the AWS Encryption SDK and version 3.0.x of the AWS Encryption CLI. For details and examples of using the new multi-Region-aware symbol, see Using multi-Region AWS KMS keys. For information about multi-Region keys, see Using multi-Region keys in the AWS Key Management Service Developer Guide.

Note

Version 4.x of the AWS Encryption SDK for .NET and version 3.x of the AWS Encryption SDK for Java are the only programming language implementations that support AWS KMS keyrings that use asymmetric RSA AWS KMS keys.

If you try to include an asymmetric KMS key in an encryption keyring in any other language implementation, the encrypt call fails. If you include it in a decryption keyring, it is ignored.

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

AWS KMS keyrings can include two types of wrapping keys:

- **Generator key**: Generates a plaintext data key and encrypts it. A keyring that encrypts data must have one generator key.

- **Additional keys**: Encrypts the plaintext data key that the generator key generated. AWS KMS keyrings can have zero or more additional keys.

When encrypting, the AWS KMS keyring that you use must have a generator key. When decrypting, the generator key is optional, and the distinction between generators keys and additional keys is ignored.

When an AWS KMS encryption keyring has just one AWS KMS key, that key is used to generate and encrypt the data key.
Like all keyrings, AWS KMS keyrings can be used independently or in a multi-keyring with other keyrings of the same or a different type.

**Topics**

- Required permissions for AWS KMS keyrings
- Identifying AWS KMS keys in an AWS KMS keyring
- Creating an AWS KMS keyring for encryption
- Creating an AWS KMS keyring for decryption
- Using an AWS KMS discovery keyring
- Using an AWS KMS regional discovery keyring

**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 KMS keyring, you need an AWS account and the following minimum permissions on the AWS KMS keys 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 keys, see [Authentication and access control](#) in the *AWS Key Management Service Developer Guide*.

**Identifying AWS KMS keys in an AWS KMS keyring**

An AWS KMS keyring can include one or more AWS KMS keys. To specify an AWS KMS key in an AWS KMS keyring, use a supported AWS KMS key identifier. The key identifiers you can use to identify an AWS KMS key in a keyring vary with the operation and the language implementation. For details about the key identifiers for an AWS KMS key, see [Key Identifiers](#) in the *AWS Key Management Service Developer Guide*. 
As a best practice, use the most specific key identifier that is practical for your task.

- In an encryption keyring for the AWS Encryption SDK for C, you can use a key ARN or alias ARN to identify KMS keys. In all other language implementations, you can use a key ID, key ARN, alias name, or alias ARN to encrypt data.

- In a decryption keyring, you must use a key ARN to identify AWS KMS keys. This requirement applies to all language implementations of the AWS Encryption SDK. For details, see Selecting wrapping keys.

- In a keyring used for encryption and decryption, you must use a key ARN to identify AWS KMS keys. This requirement applies to all language implementations of the AWS Encryption SDK.

If you specify an alias name or alias ARN for a KMS key in an encryption keyring, the encrypt operation saves the key ARN currently associated with the alias in the metadata of the encrypted data key. It does not save the alias. Changes to the alias don't affect the KMS key used to decrypt your encrypted data keys.

**Creating an AWS KMS keyring for encryption**

You can configure each AWS KMS keyring with a single AWS KMS key or multiple AWS KMS keys in the same or different AWS accounts and AWS Regions. The AWS KMS keys must be symmetric encryption keys (SYMMETRIC_DEFAULT). You can also use a symmetric encryption multi-Region KMS key. As with all keyrings, you can use one or more AWS KMS keyrings in a multi-keyring.

When you create an AWS KMS keyring to encrypt data, you must specify a generator key, which is an AWS KMS key that is used to generate a plaintext data key and encrypt it. The data key is mathematically unrelated to the KMS key. Then, if you choose, you can specify additional AWS KMS keys that encrypt the same plaintext data key.

To decrypt the encrypted message protected by this keyring, the keyring that you use must include at least one of the AWS KMS keys defined in the keyring, or no AWS KMS keys. (An AWS KMS keyring with no AWS KMS keys is known as an AWS KMS discovery keyring.)

In AWS Encryption SDK language implementations other than the AWS Encryption SDK for C, all wrapping keys in an encryption keyring or multi-keyring must be able to encrypt the data key. If any wrapping key fails to encrypt, the encrypt method fails. As a result, the caller must have the required permissions for all keys in the keyring. If you use a discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails. The exception is the AWS Encryption SDK
The following examples create an AWS KMS keyring with one generator key and one additional key. These examples use key ARNs to identify the KMS keys. This is a best practice for AWS KMS keyrings used for encryption, and a requirement for AWS KMS keyrings used for decryption. For details, see Identifying AWS KMS keys in an AWS KMS keyring.

C

To identify an AWS KMS key in an encryption keyring in the AWS Encryption SDK for C, specify a key ARN or alias ARN. In a decryption keyring, you must use a key ARN. For details, see Identifying AWS KMS keys in an AWS KMS keyring.

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});
```

C#/.NET

To create an AWS KMS keyring with one or multiple AWS KMS keys in the AWS Encryption SDK for .NET, create a multi-keyring. The AWS Encryption SDK for .NET includes a multi-keyring just for AWS KMS keys.

When you specify an AWS KMS key for an encryption keyring in the AWS Encryption SDK for .NET, you can use any valid key identifier: a key ID, key ARN, alias name, or alias ARN. For help identifying the AWS KMS keys in an AWS KMS keyring, see Identifying AWS KMS keys in an AWS KMS keyring.

The following example uses version 4.x of the AWS Encryption SDK for .NET to create an AWS KMS keyring with a generator key and additional keys. For a complete example, see AwsKmsMultiKeyringExample.cs.

```csharp
// Instantiate the AWS Encryption SDK and material provider
```
JavaScript Browser

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

For a complete example, see [kms_simple.ts](#) in the AWS Encryption SDK for JavaScript repository in GitHub.

```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 key for an encryption keyring in the AWS Encryption SDK for JavaScript, you can use any valid key identifier: a **key ID**, **key ARN**, **alias name**, or **alias ARN**. For
help identifying the AWS KMS keys in an AWS KMS keyring, see [Identifying AWS KMS keys in an AWS KMS keyring](#). For a complete example, see [kms_simple.ts](#) in the AWS Encryption SDK for JavaScript repository in GitHub.

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

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

Java

To create an AWS KMS keyring with one or multiple AWS KMS keys in the AWS Encryption SDK for Java, create a multi-keyring. The AWS Encryption SDK for Java includes a multi-keyring just for AWS KMS keys.

When you specify an AWS KMS key for an encryption keyring in the AWS Encryption SDK for Java, you can use any valid key identifier: a key ID, key ARN, alias name, or alias ARN. For help identifying the AWS KMS keys in an AWS KMS keyring, see [Identifying AWS KMS keys in an AWS KMS keyring](#). For a complete example, see [BasicEncryptionKeyringExample.java](#) in the AWS Encryption SDK for Java repository in GitHub.

```java
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder().build();
final MaterialProviders materialProviders = MaterialProviders.builder()
  .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
  .build();

String generatorKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
List<String> additionalKey = Collections.singletonList("alias/exampleAlias");

// Create the AWS KMS keyring
```
final CreateAwsKmsMultiKeyringInput keyringInput =
CreateAwsKmsMultiKeyringInput.builder()
  .generator(generatorKey)
  .kmsKeyIds(additionalKey)
  .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

Creating an AWS KMS keyring for decryption

You also specify an AWS KMS keyring when decrypting the encrypted message that the AWS Encryption SDK returns. If the decryption keyring specifies AWS KMS keys, the AWS Encryption SDK will use only those wrapping keys to decrypt the encrypted data keys in the encrypted message. (You can also use an AWS KMS discovery keyring, which doesn't specify any AWS KMS keys.)

When decrypting, the AWS Encryption SDK searches the AWS KMS keyring for an AWS KMS key 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 gets the key ARN of the AWS KMS key that encrypted the data key from the metadata of the encrypted message.
- The AWS Encryption SDK searches the decryption keyring for an AWS KMS key with a matching key ARN.
- If it finds an AWS KMS key with a matching key ARN in the keyring, the AWS Encryption SDK asks AWS KMS to use the KMS key 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 AWS KMS key that encrypted the data key is included in the decryption keyring. If the decryption keyring doesn't include the ARNs of any of the AWS KMS keys that encrypted any of the data keys, the AWS Encryption SDK fails the decrypt call without ever calling AWS KMS.

Beginning in version 1.7.x, when decrypting an encrypted data key, the AWS Encryption SDK always passes the key ARN of the AWS KMS key to the KeyId parameter of the AWS KMS Decrypt operation. Identifying the AWS KMS key when decrypting is an AWS KMS best practice that assures that you decrypt the encrypted data key with the wrapping key you intend to use.

A decrypt call with an AWS KMS keyring succeeds when at least one AWS KMS key 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 AWS KMS key. This behavior enables you to encrypt data under multiple AWS KMS keys 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 an AWS KMS key in a decryption keyring, you must use its key ARN. Otherwise, the AWS KMS key 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 AWS KMS keys 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. However, instead of referring to the additional key by its alias, alias/exampleAlias, the example uses the additional key's key ARN as required by decrypt calls.

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.

C

```c
const char * additional_key = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"

struct aws_cryptosdk_keyring *kms_decrypt_keyring =
    Aws::Cryptosdk::KmsKeyring::Builder().Build(additional_key);
```

C# / .NET

Because this decrypt keyring includes only one AWS KMS key, the example uses the CreateAwsKmsKeyring() method with an instance of its CreateAwsKmsKeyringInput object. To create a AWS KMS keyring with one AWS KMS key, you can use a single-key or multi-key keyring. For details, see Encrypting data in the AWS Encryption SDK for .NET. The following example uses version 4.x of the AWS Encryption SDK for .NET to create an AWS KMS keyring for decryption.

```c
// Instantiate the AWS Encryption SDK and material providers
```
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

string additionalKey = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321";

// Instantiate a KMS keyring for one AWS KMS key.
var kmsDecryptKeyringInput = new CreateAwsKmsKeyringInput {
  KmsClient = new AmazonKeyManagementServiceClient(),
  KmsKeyId = additionalKey
};

var kmsDecryptKeyring = materialProviders.CreateAwsKmsKeyring(kmsDecryptKeyringInput);

Java

Because this decrypt keyring includes only one AWS KMS key, the example uses the
CreateAwsKmsKeyring() method with an instance of its CreateAwsKmsKeyringInput
object. To create a AWS KMS keyring with one AWS KMS key, you can use a single-key or multi-key keyring.

// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder().build();
final MaterialProviders materialProviders = MaterialProviders.builder().MaterialProvidersConfig(MaterialProvidersConfig.builder().build());
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 AWS KMS keys to decrypt an encrypted data key. The call to AWS KMS succeeds only when the caller has permission to use that AWS KMS key to decrypt data.

C

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

C#/ .NET

The following example uses version 4.x of the AWS Encryption SDK for .NET.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

string generatorKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate a KMS keyring for one AWS KMS key.
var kmsDecryptKeyringInput = new CreateAwsKmsKeyringInput
    { 
        KmsClient = new AmazonKeyManagementServiceClient(),
        KmsKeyId = generatorKey
    };
```
var kmsDecryptKeyring =
   materialProviders.CreateAwsKmsKeyring(kmsDecryptKeyringInput);

JavaScript Browser

const clientProvider = getClient(KMS, { credentials })

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

JavaScript Node.js

const keyring = new KmsKeyringNode({
   generatorKeyId,
   keyIds: [additionalKey, otherKey]
});

Java

// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder().build();
final MaterialProviders materialProviders = MaterialProviders.builder()
   .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
   .build();

String generatorKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Create a AwsKmsKeyring
CreateAwsKmsKeyringInput kmsDecryptKeyringInput = CreateAwsKmsKeyringInput.builder()
   .generator(generatorKey)
   .kmsClient(KmsClient.create())
   .build();
IKeyring kmsKeyring = materialProviders.CreateAwsKmsKeyring(kmsDecryptKeyringInput);

Unlike an encryption keyring that uses all of the specified AWS KMS keys, you can decrypt an encrypted message using a decryption keyring that includes AWS KMS keys that are unrelated
to the encrypted message, and AWS KMS keys 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**

When decrypting, it's a **best practice** to specify the wrapping keys that the AWS Encryption SDK can use. To follow this best practice, use an AWS KMS decryption keyring that limits the AWS KMS wrapping keys 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 wrapping keys.

The AWS Encryption SDK provides a standard AWS KMS discovery keyring and a discovery keyring for AWS KMS multi-Region keys. For information about using multi-Region keys with the AWS Encryption SDK, see [Using multi-Region AWS KMS keys](#).

Because it doesn't specify any wrapping keys, a discovery keyring can't encrypt data. If you use a discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails. The exception is the AWS Encryption SDK for C, where the encrypt operation ignores a standard discovery keyring, but fails if you specify a multi-Region discovery keyring, alone or in a multi-keyring.

When decrypting, a discovery keyring allows the AWS Encryption SDK to ask AWS KMS to decrypt any encrypted data key by using the AWS KMS key that encrypted it, regardless of who owns or has access to that AWS KMS key. The call succeeds only when the caller has `kms:Decrypt` permission on the AWS KMS key.

⚠️ **Important**

If you include an AWS KMS discovery keyring in a decryption multi-keyring, the discovery keyring overrides all KMS key restrictions specified by other keyrings in the multi-keyring. The multi-keyring behaves like its least restrictive keyring. An AWS KMS discovery keyring has no effect on encryption when used by itself or in a multi-keyring.

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 AWS KMS key that was used to encrypt a data key in the encrypted message, just so the caller has permission to use that AWS
KMS key to decrypt. This might not be the AWS KMS key that the caller intends to use. For example, one of the encrypted data keys might have been encrypted under a less secure AWS KMS key 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 AWS KMS keys in other AWS accounts and Regions, and AWS KMS keys that the caller doesn’t have permission to use for decryption.

If you use a discovery keyring, we recommend that you use a discovery filter to limit the KMS keys that can be used to those in specified AWS accounts and partitions. Discovery filters are supported in versions 1.7.x and later of the AWS Encryption SDK. For help finding your account ID and partition, see [Your AWS account identifiers](https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html) and [ARN format](https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html) in the [AWS General Reference](https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html).

The following code instantiates an AWS KMS discovery keyring with a discovery filter that limits the KMS keys the AWS Encryption SDK can use to those in the aws partition and 111122223333 example account.

Before using this code, replace the example AWS account and partition values with valid values for your AWS account and partition. If your KMS keys are in China Regions, use the aws-cn partition value. If your KMS keys are in AWS GovCloud (US) Regions, use the aws-us-gov partition value. For all other AWS Regions, use the aws partition value.

For a complete example, see: [kms_discovery.cpp](https://github.com/aws/aws-encryption-sdk-cpp/tree/master/cpp-sdk/encryption/docs/examples).

```c
std::shared_ptr<KmsKeyring::> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
    .AddAccount("111122223333")
    .Build());

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

### C# / .NET

The following example uses version 4.x of the AWS Encryption SDK for .NET.

```csharp
// Instantiate the AWS Encryption SDK and material providers
```
```javascript
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

List<string> account = new List<string> { "111122223333" };

// In a discovery keyring, you specify an AWS KMS client and a discovery filter, 
// but not a AWS KMS key
var kmsDiscoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput {
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = account,
        Partition = "aws"
    }
};

var kmsDiscoveryKeyring = materialProviders.CreateAwsKmsDiscoveryKeyring(kmsDiscoveryKeyringInput);
```

**JavaScript Browser**

In JavaScript, you must explicitly specify the discovery property.

```javascript
const clientProvider = getClient(KMS, { credentials })

const discovery = true
const keyring = new KmsKeyringBrowser(clientProvider, {
    discovery,
    discoveryFilter: { accountIDs: [111122223333], partition: 'aws' }
})
```

**JavaScript Node.js**

In JavaScript, you must explicitly specify the discovery property.

```javascript
const discovery = true

const keyring = new KmsKeyringNode({
    discovery,
    discoveryFilter: { accountIDs: ['111122223333'], partition: 'aws' }
})
```
// Create discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
    .partition("aws")
    .accountIds("111122223333")
    .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
    = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
    .discoveryFilter(discoveryFilter)
    .build();
IKeyring decryptKeyring =
    matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);

Using an AWS KMS regional discovery keyring

An AWS KMS regional discovery keyring is a keyring that doesn't specify the ARNs of KMS keys. Instead, it allows the AWS Encryption SDK to decrypt using only the KMS keys in particular AWS Regions.

When decrypting with an AWS KMS regional discovery keyring, the AWS Encryption SDK decrypts any encrypted data key that was encrypted under an AWS KMS key in the specified AWS Region. To succeed, the caller must have kms:Decrypt permission on at least one of the AWS KMS keys in the specified AWS Region that encrypted a data key.

Like other discovery keyrings, the regional discovery keyring has no effect on encryption. It works only when decrypting encrypted messages. If you use a regional discovery keyring in a multi-keyring that is used for encrypting and decrypting, it is effective only when decrypting. If you use a multi-Region discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails.

⚠️ Important
If you include an AWS KMS regional discovery keyring in a decryption multi-keyring, the regional discovery keyring overrides all KMS key restrictions specified by other keyrings in the multi-keyring. The multi-keyring behaves like its least restrictive keyring. An AWS KMS discovery keyring has no effect on encryption when used by itself or in a multi-keyring.
The regional discovery keyring in the AWS Encryption SDK for C attempts to decrypt only with KMS keys in the specified Region. When you use a discovery keyring in the AWS Encryption SDK for JavaScript and AWS Encryption SDK for .NET, you configure the Region on the AWS KMS client. These AWS Encryption SDK implementations don't filter KMS keys by Region, but AWS KMS will fail a decrypt request for KMS keys outside of the specified Region.

If you use a discovery keyring, we recommend that you use a discovery filter to limit the KMS keys used in decryption to those in specified AWS accounts and partitions. Discovery filters are supported in versions 1.7.x and later of the AWS Encryption SDK.

For example, the following code creates an AWS KMS regional discovery keyring with a discovery filter. This keyring limits the AWS Encryption SDK to KMS keys in account 111122223333 in the US West (Oregon) Region (us-west-2).

```c
std::shared_ptr<KmsKeyring::DiscoveryFilter> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
    .AddAccount("111122223333")
    .Build());

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

C# / .NET

The AWS Encryption SDK for .NET does not have a dedicated regional discovery keyring. However, you can use several techniques to limit the KMS keys used when decrypting to a particular Region.

The most efficient way to limit the Regions in a discovery keyring is to use a multi-Region-aware discovery keyring, even if you encrypted the data using only single-Region keys. When it encounters single-Region keys, the multi-Region-aware keyring does not use any multi-Region features.
The keyring returned by the `CreateAwsKmsMrkDiscoveryKeyring()` method filters KMS keys by Region before calling AWS KMS. It sends a decrypt request to AWS KMS only when the encrypted data key was encrypted by a KMS key in the Region specified by the `Region` parameter in the `CreateAwsKmsMrkDiscoveryKeyringInput` object.

The following examples uses version 4.x of the AWS Encryption SDK for .NET.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

List<string> account = new List<string> { "111122223333" };

// Create the discovery filter
var filter = DiscoveryFilter = new DiscoveryFilter
{
    AccountIds = account,
    Partition = "aws"
};

var regionalDiscoveryKeyringInput = new CreateAwsKmsMrkDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    Region = RegionEndpoint.USWest2,
    DiscoveryFilter = filter
};

var kmsRegionalDiscoveryKeyring = materialProviders.CreateAwsKmsMrkDiscoveryKeyring(regionalDiscoveryKeyringInput);
```

You can also limit KMS keys to a particular AWS Region by specifying a Region in your instance of the AWS KMS client (`AmazonKeyManagementServiceClient`). However, this configuration is less efficient and potentially more costly than using a multi-Region-aware discovery keyring. Instead of filtering KMS keys by Region before calling AWS KMS, the AWS Encryption SDK for .NET calls AWS KMS for each encrypted data key (until it decrypts one) and relies on AWS KMS to limit the KMS keys it uses to the specified Region.

The following example uses version 4.x of the AWS Encryption SDK for .NET.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
```
var mpl = new MaterialProviders(new MaterialProvidersConfig());

List<string> account = new List<string> { "111122223333" };

// Create the discovery filter,
// but not a AWS KMS key
var createRegionalDiscoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = account,
        Partition = "aws"
    }
};

var kmsRegionalDiscoveryKeyring =
materialProviders.CreateAwsKmsDiscoveryKeyring(createRegionalDiscoveryKeyringInput);

JavaScript Browser

const clientProvider = getClient(KMS, { credentials })

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser(clientProvider, {
    discovery,
    discoveryFilter: { accountIDs: ['111122223333'], partition: 'aws' }
})

JavaScript Node.js

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

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: ['111122223333'], partition: 'aws' }
})
Java

```java
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
    .partition("aws")
    .accountIds("111122223333")
    .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
    = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
        .discoveryFilter(discoveryFilter)
        .regions("us-west-2")
        .build();
IKeyring decryptKeyring =
    matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

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 AWS KMS keys in particular regions. The following example creates an AWS KMS regional discovery keyring that can use AWS KMS keys in account 111122223333 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,
    discoveryFilter: { accountIDs: ["111122223333"], partition: 'aws' }
})
```
AWS KMS Hierarchical keyrings

⚠️ Important

The AWS KMS Hierarchical keyring is only supported by version 4.x of the AWS Encryption SDK for .NET and version 3.x of the AWS Encryption SDK for Java.

With the AWS KMS Hierarchical keyring, you can protect your cryptographic materials under a symmetric encryption KMS key without calling AWS KMS every time you encrypt or decrypt data. It is a good choice for applications that need to minimize calls to AWS KMS, and applications that can reuse some cryptographic materials without violating their security requirements.

The Hierarchical keyring is a cryptographic materials caching solution that reduces the number of AWS KMS calls by using AWS KMS protected branch keys persisted in an Amazon DynamoDB table, and then locally caching branch key materials used in encrypt and decrypt operations. The DynamoDB table serves as the branch key store that manages and protects branch keys. It stores the active branch key and all previous versions of the branch key. The active branch key is the most recent branch key version. The Hierarchical keyring uses a unique data key to encrypt each message and encrypts each data key with a unique wrapping key derived from the active branch key. The Hierarchical keyring is dependent on the hierarchy established between active branch keys and their derived wrapping keys.

The Hierarchical keyring typically uses each branch key version to satisfy multiple requests. But you control the extent to which active branch keys are reused and determine how often the active branch key is rotated. The active version of the branch key remains active until you rotate it. Previous versions of the active branch key will not be used to perform encrypt operations, but they can still be queried and used in decrypt operations.

When you instantiate the Hierarchical keyring, it creates a local cache. You specify a cache limit that defines the maximum amount of time that the branch key materials are stored within the local cache before they expire and are evicted from the cache. The Hierarchical keyring makes one AWS KMS call to decrypt the branch key and assemble the branch key materials the first time a branch-key-id is specified in an operation. Then, the branch key materials are stored in the local cache and reused for all encrypt and decrypt operations that specify that branch-key-id until the cache limit expires. Storing branch key materials in the local cache reduces AWS KMS calls. For example, consider a cache limit of 15 minutes. If you perform 10,000 encrypt operations within that cache limit, the traditional AWS KMS keyring would need to make 10,000 AWS KMS calls to
satisfy 10,000 encrypt operations. If you have one active branch-key-id, the Hierarchical keyring only needs to make one AWS KMS call to satisfy 10,000 encrypt operations.

The local cache consists of two partitions, one for encrypt operations and a second for decrypt operations. The encrypt partition stores the branch key materials assembled from the active branch key and reuses them for all encrypt operations until the cache limit expires. The decrypt partition stores the branch key materials assembled for other branch key versions identified in decrypt operations. The decryption partition can store multiple active branch key materials versions at a time. When it's configured to use a branch key ID supplier for a multitenant environment, the encrypt partition can also store multiple branch key materials versions at a time. For more information, see Using the Hierarchical keyring in multitenant environments.

Note

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

Topics

• How it works
• Prerequisites
• Create a Hierarchical keyring
• Rotate your active branch key
• Using the Hierarchical keyring in multitenant environments

How it works

The following walkthroughs describe how the Hierarchical keyring assembles encryption and decryption materials, and the different calls that the keyring makes for encrypt and decrypt operations. For technical details on the wrapping key derivation and plaintext data key encryption processes, see AWS KMS Hierarchical keyring technical details.

Encrypt and sign

The following walkthrough describes how the Hierarchical keyring assembles encryption materials and derives a unique wrapping key.
1. The encryption method asks the Hierarchical keyring for encryption materials. The keyring generates a plaintext data key, then checks to see if there are valid branch materials in the local cache to generate the wrapping key. If there are valid branch key materials, the keyring proceeds to Step 5.

2. If there are no valid branch key materials, the Hierarchical keyring queries the branch key store for the active branch key.
   a. The branch key store calls AWS KMS to decrypt the active branch key and returns the plaintext active branch key. Data identifying the active branch key is serialized to provide additional authenticated data (AAD) in the decrypt call to AWS KMS.
   b. The branch key store returns the plaintext branch key and data that identifies it, such as the branch key version.

3. The Hierarchical keyring assembles branch key materials (the plaintext branch key and branch key version) and stores a copy of them in the local cache.

4. The Hierarchical keyring derives a unique wrapping key from the plaintext branch key and a 16-byte random salt. It uses the derived wrapping key to encrypt a copy of the plaintext data key.

The encryption method uses the encryption materials to encrypt the data. For more information, see How the AWS Encryption SDK encrypts data.

Decrypt and verify

The following walkthrough describes how the Hierarchical keyring assembles decryption materials and decrypts the encrypted data key.

1. The decryption method identifies the encrypted data key from the encrypted message, and passes it to the Hierarchical keyring.

2. The Hierarchical keyring deserializes data identifying the encrypted data key, including the branch key version, the 16-byte salt, and other information describing how the data key was encrypted.
   For more information, see AWS KMS Hierarchical keyring technical details.

3. The Hierarchical keyring checks to see if there are valid branch key materials in the local cache that match the branch key version identified in Step 2. If there are valid branch key materials, the keyring proceeds to Step 6.
4. If there are no valid branch key materials, the Hierarchical keyring queries the branch key store for the branch key that matches the branch key version identified in Step 2.

   a. The branch key store calls AWS KMS to decrypt the branch key and returns the plaintext active branch key. Data identifying the active branch key is serialized to provide additional authenticated data (AAD) in the decrypt call to AWS KMS.

   b. The branch key store returns the plaintext branch key and data that identifies it, such as the branch key version.

5. The Hierarchical keyring assembles branch key materials (the plaintext branch key and branch key version) and stores a copy of them in the local cache.

6. The Hierarchical keyring uses the assembled branch key materials and the 16-byte salt identified in Step 2 to reproduce the unique wrapping key that encrypted the data key.

7. The Hierarchical keyring uses the reproduced wrapping key to decrypt the data key and returns the plaintext data key.

The decryption method uses the decryption materials and plaintext data key to decrypt the encrypted message. For more information, see How the AWS Encryption SDK decrypts an encrypted message.

**Prerequisites**

The AWS Encryption SDK doesn't require an AWS account and it doesn't depend on any AWS service. However, the Hierarchical keyring depends on AWS KMS and Amazon DynamoDB.

To use a Hierarchical keyring, you need a symmetric encryption AWS KMS key with kms:Decrypt permissions. You can also use a symmetric encryption multi-Region key. For detailed information about permissions for AWS KMS keys, see Authentication and access control in the AWS Key Management Service Developer Guide.

Before you can create and use a Hierarchical keyring, you must create your branch key store and populate it with your first active branch key.

**Step 1: Configure a new key store service**

The key store service provides several API operations, such as CreateKeyStore and CreateKey, to help you assemble the Hierarchical keyring prerequisites and manage your branch key store.
The following example creates a key store service. You must specify a DynamoDB table name to serve as the name of your branch key store, a logical name for the branch key store, and the KMS key ARN that identifies the KMS key that will protect your branch keys.

The logical key store name is cryptographically bound to all data stored in the table to simplify DynamoDB restore operations. The logical key store name can be the same as your DynamoDB table name, but it does not have to be. We recommend specifying your DynamoDB table name as the logical table name when you first configure your key store service. You must always specify the same logical table name. In the event that your branch key store name changes after restoring your DynamoDB table from a backup, the logical key store name maps to the DynamoDB table name you specify to ensure that the Hierarchical keyring can still access your branch key store.

### Note

The logical key store name is included in the encryption context of all key store service API operations that call AWS KMS. The encryption context is not secret, its values—including the logical key store name—appear in plaintext in AWS CloudTrail logs.

**C# / .NET**

```csharp
var kmsConfig = new KMSConfiguration { KmsKeyArn = kmsKeyArn }; var keystoreConfig = new KeyStoreConfig { KmsClient = new AmazonKeyManagementServiceClient(), KmsConfiguration = kmsConfig, DdbTableName = keyStoreName, DdbClient = new AmazonDynamoDBClient(), LogicalKeyStoreName = logicalKeyStoreName }; var keystore = new KeyStore(keystoreConfig);
```

**Java**

```java
final KeyStore keystore = KeyStore.builder().KeyStoreConfig( KeyStoreConfig.builder() .ddbClient(DynamoDbClient.create()) .ddbTableName(keyStoreName) .logicalKeyStoreName(logicalKeyStoreName)
```
Step 2: Call `CreateKeyStore` to create a branch key store

The following operation creates the branch key store that will persist and protect your branch keys.

C# / .NET

```csharp
var createKeyStoreOutput = keystore.CreateKeyStore(new CreateKeyStoreInput());
```

Java

```java
keystore.CreateKeyStore(CreateKeyStoreInput.builder().build());
```

The `CreateKeyStore` operation creates a DynamoDB table with the table name you specified in Step 1 and the following required values.

<table>
<thead>
<tr>
<th></th>
<th>Partition key</th>
<th>Sort key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base table</td>
<td>branch-key-id</td>
<td>type</td>
</tr>
</tbody>
</table>

**Note**

You can manually create the DynamoDB table that serves as your branch key store instead of using the `CreateKeyStore` operation. If you choose to manually create the branch key store, you must specify the following string values for the partition and sort keys:

- Partition key: `branch-key-id`
- Sort key: `type`
Step 3: Call CreateKey to create a new active branch key

The following operation creates a new active branch key using the KMS key you specified in Step 1, and adds the active branch key to the DynamoDB table you created in Step 2.

When you call CreateKey, you can choose to specify the following optional values.

- **Branch key identifier**: defines a custom `branch-key-id`.

  To create a custom `branch-key-id`, you must also include an additional encryption context with the `encryptionContext` parameter.

- **Encryption context**: defines an optional set of non-secret key–value pairs that provides additional authenticated data (AAD) in the encryption context included in the `kms:GenerateDataKeyWithoutPlaintext` call.

  This additional encryption context is displayed with the `aws-crypto-ec:` prefix.

**C# / .NET**

```csharp
var additionalEncryptionContext = new Dictionary<string, string>();
additionalEncryptionContext.Add("Additional Encryption Context for", "custom branch key id");

var branchKeyId = keystore.CreateKey(new CreateKeyInput
{
    BranchKeyIdentifier = "custom-branch-key-id", // OPTIONAL
    EncryptionContext = additionalEncryptionContext // OPTIONAL
});
```

**Java**

```java
final Map<String, String> additionalEncryptionContext = Collections.singletonMap("Additional Encryption Context for", "custom branch key id");

final String BranchKey = keystore.CreateKey(
    CreateKeyInput.builder()
        .branchKeyIdentifier("custom-branch-key-id") //OPTIONAL
        .encryptionContext(additionalEncryptionContext) //OPTIONAL
        .build()).branchKeyIdentifier();
```

First, the `CreateKey` operation generates the following values.
• A version 4 **Universally Unique Identifier** (UUID) for the `branch-key-id` (unless you specified a custom `branch-key-id`).

• A version 4 UUID for the branch key version

• A timestamp in the [ISO 8601 date and time format](https://en.wikipedia.org/wiki/ISO_8601) in Coordinated Universal Time (UTC).

Then, the `CreateKey` operation calls `kms:GenerateDataKeyWithoutPlaintext` using the following request.

```json
{
    "EncryptionContext": {
        "branch-key-id": "branch-key-id",
        "type": "type",
        "create-time": "timestamp",
        "logical-key-store-name": "the logical table name for your branch key store",
        "kms-arn": "the KMS key ARN",
        "hierarchy-version": "1",
        "aws-crypto-ec:contextKey": "contextValue"
    },
    "KeyId": "the KMS key ARN you specified in Step 1",
    "NumberOfBytes": "32"
}
```

Next, the `CreateKey` operation calls `kms:ReEncrypt` to create an active record for the branch key by updating the encryption context.

Last, the `CreateKey` operation calls `ddb:TransactWriteItems` to write a new item that will persist the branch key in the table you created in **Step 2**. The item has the following attributes.

```json
{
    "branch-key-id": "branch-key-id",
    "type": "branch:ACTIVE",
    "enc": "the branch key returned by the GenerateDataKeyWithoutPlaintext call",
    "version": "branch:version:the branch key version UUID",
    "create-time": "timestamp",
    "kms-arn": "the KMS key ARN you specified in Step 1",
    "hierarchy-version": "1",
    "aws-crypto-ec:contextKey": "contextValue"
}
```
Create a Hierarchical keyring

To initialize the Hierarchical keyring, you must provide the following values:

- **A branch key store name**
  
  The name of the DynamoDB table you created to serve as your branch key store.

- **A cache limit time to live (TTL)**
  
  The amount of time in seconds that a branch key materials entry within the local cache can be used before it expires. This value must be greater than zero. When the cache limit TTL expires, the entry is evicted from the local cache.

- **A branch key identifier**
  
  The `branch-key-id` that identifies the active branch key in your branch key store.

  **Note**
  
  To initialize the Hierarchical keyring for multitenant use, you must specify a branch key ID supplier instead of a `branch-key-id`. For more information, see Using the Hierarchical keyring in multitenant environments.

- **(Optional) A cache**
  
  If you want to customize your cache type or the number of branch key materials entries that can be stored in the local cache, specify the cache type and entry capacity when you initialize the keyring.

  Cache type defines the threading model. The Hierarchical keyring provides three cache types that support multitenant environments: Default, MultiThreaded, StormTracking.

  If you do not specify a cache, the Hierarchical keyring automatically uses the Default cache type and sets the entry capacity to 1000.

  Default (Recommended)
  
  For most users, the Default cache fulfills their threading requirements. The Default cache is designed to support heavily multithreaded environments. When a branch key materials entry expires, the Default cache prevents multiple threads from calling AWS KMS and Amazon...
DynamoDB by notifying one thread that the branch key materials entry is going to expire 10 seconds in advance. This ensures that only one thread sends a request to AWS KMS to refresh the cache.

To initialize your Hierarchical keyring with a Default cache, specify the following value:

- **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

**C# / .NET**

```csharp
CacheType defaultCache = new CacheType
{
    Default = new DefaultCache{EntryCapacity = 100}
};
```

**Java**

```java
.cache(CacheType.builder()
    .Default(DefaultCache.builder()
        .entryCapacity(100)
        .build())
```

The Default and StormTracking caches support the same threading model, but you only need to specify the entry capacity to initialize the Hierarchical keyring with the Default cache. For more granular cache customizations, use the StormTracking cache.

**MultiThreaded**

The MultiThreaded cache is safe to use in multithreaded environments, but it does not provide any functionality to minimize AWS KMS or Amazon DynamoDB calls. As a result, when a branch key materials entry expires, all threads will be notified at the same time. This can result in multiple AWS KMS calls to refresh the cache.

To initialize your Hierarchical keyring with a MultiThreaded cache, specify the following values:

- **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

- **Entry pruning tail size**: defines the number of entries to prune if the entry capacity is reached.
C# / .NET

CacheType multithreadedCache = new CacheType
{
    MultiThreaded = new MultiThreadedCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1
    }
};

Java

.cache(CacheType.builder()
    .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .build())

StormTracking

The StormTracking cache is designed to support heavily multithreaded environments. When a branch key materials entry expires, the StormTracking cache prevents multiple threads from calling AWS KMS and Amazon DynamoDB by notifying one thread that the branch key materials entry is going to expire in advance. This ensures that only one thread sends a request to AWS KMS to refresh the cache.

To initialize your Hierarchical keyring with a StormTracking cache, specify the following values:

- **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

- **Entry pruning tail size**: defines the number of branch key materials entries to prune at a time.

  Default value: 1 entry

- **Grace period**: defines the number of seconds before expiration that an attempt to refresh branch key materials is made.

  Default value: 10 seconds
• **Grace interval**: defines the number of seconds between attempts to refresh the branch key materials.

Default value: 1 second

• **Fan out**: defines the number of simultaneous attempts that can be made to refresh the branch key materials.

Default value: 20 attempts

• **In flight time to live (TTL)**: defines the number of seconds until an attempt to refresh the branch key materials times out. Any time the cache returns `NoSuchEntry` in response to a `GetCacheEntry`, that branch key is considered to be *in flight* until the same key is written with a `PutCache` entry.

Default value: 20 seconds

• **Sleep**: defines the number of seconds that a thread should sleep if the `fanOut` is exceeded.

Default value: 20 milliseconds

**C# / .NET**

```csharp
CacheType stormTrackingCache = new CacheType
{
    StormTracking = new StormTrackingCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1,
        FanOut = 20,
        GraceInterval = 1,
        GracePeriod = 10,
        InFlightTTL = 20,
        SleepMilli = 20
    }
};
```

**Java**

```java
.cache(CacheType.builder()
    .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
    ));
```
• (Optional) A list of Grant Tokens

If you control access to the KMS key in your Hierarchical keyring with grants, you must provide all necessary grant tokens when you initialize the keyring.

The following example initializes a Hierarchical keyring with a cache limit TLL of 600 seconds, and an entry capacity of 1000.

C#/ .NET

```csharp
// Instantiate the AWS Encryption SDK and material providers
var mpl = new MaterialProviders(new MaterialProvidersConfig());
var esdk = new ESDK(new AwsEncryptionSdkConfig());

// Instantiate the keyring
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
    KeyStore = branchKeyStoreName,
    BranchKeyId = branch-key-id,
    Cache = new CacheType { Default = new DefaultCache{EntryCapacity = 1000} },
    TtlSeconds = 600
};
```

Java

```java
final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
CreateAwsKmsHierarchicalKeyringInput.builder()
    .keyStore(branchKeyStoreName)
    .branchKeyId(branch-key-id)
    .ttlSeconds(600)
    .cache(CacheType.builder() //OPTIONAL
        .Default(DefaultCache.builder())
        .entryCapacity(1000)
```
Rotate your active branch key

There can only be one active version for each branch key at a time. The Hierarchical keyring typically uses each active branch key version to satisfy multiple requests. But you control the extent to which active branch keys are reused and determine how often the active branch key is rotated.

Branch keys are not used to encrypt plaintext data keys. They are used to derive the unique wrapping keys that encrypt plaintext data keys. The wrapping key derivation process produces a unique 32 byte wrapping key with 28 bytes of randomness. This means that a branch key can derive more than 79 octillion, or $2^{96}$, unique wrapping keys before cryptographic wear-out occurs. Despite this very low exhaustion risk, you might be required to rotate your active branch keys due to business or contract rules or government regulations.

The active version of the branch key remains active until you rotate it. Previous versions of the active branch key will not be used to perform encrypt operations and cannot be used to derive new wrapping keys. But they can still be queried and provide wrapping keys to decrypt the data keys that they encrypted while active.

Use the key store service VersionKey operation to rotate your active branch key. When you rotate the active branch key, a new branch key is created to replace the previous version. The branch-key-id does not change when you rotate the active branch key. You must specify the branch-key-id that identifies the current active branch key when you call VersionKey.

**C# / .NET**

```csharp
keystore.VersionKey(new VersionKeyInput{BranchKeyIdentifier = branchKeyId});
```

**Java**

```java
keystore.VersionKey(
    VersionKeyInput.builder()
    .branchKeyIdentifier("branch-key-id")
    .build()
)```
Using the Hierarchical keyring in multitenant environments

You can use the key hierarchy established between active branch keys and their derived wrapping keys to support multitenant environments by creating a branch key for each tenant in your environment. The Hierarchical keyring then encrypts all of the data for a given tenant with their distinct branch key. This enables you to isolate tenant data by branch key.

Each tenant has their own branch key that is defined by a unique branch-key-id. There can only be one active version of each branch-key-id at a time.

Before you can initialize your Hierarchical keyring for multitenant use, you must create a branch key for each tenant and create a branch key ID supplier. Use the branch key ID supplier to create a friendly name for your branch-key-ids to make it easier to recognize the correct branch-key-id for a tenant. For example, the friendly name enables you to refer to a branch key as tenant1 instead of b3f61619-4d35-48ad-a275-050f87e15122.

For decrypt operations, you can either statically configure a single Hierarchical keyring to restrict decryption to a single tenant, or you can use the branch key ID supplier to identify which tenant is responsible for decrypting a message.

First, follow Step 1 and Step 2 of the Prerequisites procedures. Then, use the following procedures to create a branch key for each tenant, create a branch key ID supplier, and initialize your Hierarchical keyring for multitenant use.

Step 1: Create a branch key for each tenant in your environment

Call CreateKey for each tenant.

The following operation creates two branch keys using the KMS key you specified when creating your key store service, and adds the branch keys to the DynamoDB table you created to serve as your branch key store. The same KMS key must protect all branch keys.

C# / .NET

```csharp
var branchKeyId1 = keystore.CreateKey(new CreateKeyInput());
var branchKeyId2 = keystore.CreateKey(new CreateKeyInput());
```
Java

```java
CreateKeyOutput branchKeyId1 = keystore.CreateKey(CreateKeyInput.builder().build());
CreateKeyOutput branchKeyId2 = keystore.CreateKey(CreateKeyInput.builder().build());
```

**Step 2: Create a branch key ID supplier**

The following example creates a branch key ID supplier.

C# / .NET

```csharp
var branchKeySupplier =
    new ExampleBranchKeySupplier(branchKeyId1.BranchKeyIdentifier,
    branchKeyId2.BranchKeyIdentifier);
```

Java

```java
IBranchKeyIdSupplier branchKeyIdSupplier = new ExampleBranchKeyIdSupplier(
    branchKeyId1.branchKeyIdentifier(), branchKeyId2.branchKeyIdentifier());
```

**Step 3: Initialize your Hierarchical keyring with the branch key ID supplier**

To initialize the Hierarchical keyring you must provide the following values:

- A branch key store name
- A **cache limit time to live (TTL)**
- A branch key ID supplier
- (Optional) A cache

If you want to customize your cache type or the number of branch key materials entries that can be stored in the local cache, specify the cache type and entry capacity when you initialize the keyring.

Cache type defines the threading model. The Hierarchical keyring provides three cache types that support multitenant environments: Default, MultiThreaded, StormTracking.

If you do not specify a cache, the Hierarchical keyring automatically uses the Default cache type and sets the entry capacity to 1000.
Default (Recommended)

For most users, the Default cache fulfills their threading requirements. The Default cache is designed to support heavily multithreaded environments. When a branch key materials entry expires, the Default cache prevents multiple threads from calling AWS KMS and Amazon DynamoDB by notifying one thread that the branch key materials entry is going to expire 10 seconds in advance. This ensures that only one thread sends a request to AWS KMS to refresh the cache.

To initialize your Hierarchical keyring with a Default cache, specify the following value:

- **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

### C# / .NET

```csharp
CacheType defaultCache = new CacheType
{
    Default = new DefaultCache{EntryCapacity = 100}
};
```

### Java

```java
.cache(CacheType.builder()
    .Default(DefaultCache.builder()
        .entryCapacity(100)
        .build())
```

The Default and StormTracking caches support the same threading model, but you only need to specify the entry capacity to initialize the Hierarchical keyring with the Default cache. For more granular cache customizations, use the StormTracking cache.

MultiThreaded

The MultiThreaded cache is safe to use in multithreaded environments, but it does not provide any functionality to minimize AWS KMS or Amazon DynamoDB calls. As a result, when a branch key materials entry expires, all threads will be notified at the same time. This can result in multiple AWS KMS calls to refresh the cache.

To initialize your Hierarchical keyring with a MultiThreaded cache, specify the following values:
• **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

• **Entry pruning tail size**: defines the number of entries to prune if the entry capacity is reached.

### C# / .NET

```csharp
CacheType multithreadedCache = new CacheType
{
    MultiThreaded = new MultiThreadedCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1
    }
};
```

### Java

```java
.cache(CacheType.builder()
    .MultiThreaded(MultiThreadedCache.builder()
    .entryCapacity(100)
    .entryPruningTailSize(1)
    .build())
```

### StormTracking

The StormTracking cache is designed to support heavily multithreaded environments. When a branch key materials entry expires, the StormTracking cache prevents multiple threads from calling AWS KMS and Amazon DynamoDB by notifying one thread that the branch key materials entry is going to expire in advance. This ensures that only one thread sends a request to AWS KMS to refresh the cache.

To initialize your Hierarchical keyring with a StormTracking cache, specify the following values:

• **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

• **Entry pruning tail size**: defines the number of branch key materials entries to prune at a time.

  Default value: 1 entry
• **Grace period**: defines the number of seconds before expiration that an attempt to refresh branch key materials is made.

Default value: 10 seconds

• **Grace interval**: defines the number of seconds between attempts to refresh the branch key materials.

Default value: 1 second

• **Fan out**: defines the number of simultaneous attempts that can be made to refresh the branch key materials.

Default value: 20 attempts

• **In flight time to live (TTL)**: defines the number of seconds until an attempt to refresh the branch key materials times out. Any time the cache returns NoSuchEntry in response to a GetCacheEntry, that branch key is considered to be *in flight* until the same key is written with a PutCache entry.

Default value: 20 seconds

• **Sleep**: defines the number of seconds that a thread should sleep if the fanOut is exceeded.

Default value: 20 milliseconds

C# / .NET

```csharp
CacheType stormTrackingCache = new CacheType
{
    StormTracking = new StormTrackingCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1,
        FanOut = 20,
        GraceInterval = 1,
        GracePeriod = 10,
        InFlightTTL = 20,
        SleepMilli = 20
    }
};
```
Java

```java
CacheType.builder()
    .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(20)
        .sleepMilli(20)
        .build())
```

- (Optional) A list of Grant Tokens

If you control access to the KMS key in your Hierarchical keyring with grants, you must provide all necessary grant tokens when you initialize the keyring.

The following example initializes a Hierarchical keyring with the branch key ID supplier created in Step 2, a cache limit TLL of 600 seconds, and an entry capacity of 1000.

C# / .NET

```csharp
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{  
    KeyStore = keystore,
    BranchKeyIdSupplier = branchKeySupplier,
    Cache = new CacheType { Default = new DefaultCache{EntryCapacity = 1000} },
    TtlSeconds = 600
};
var keyring = mpl.CreateAwsKmsHierarchicalKeyring(createKeyringInput);
```

Java

```java
final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
    CreateAwsKmsHierarchicalKeyringInput.builder()
    .keyStore(branchKeyStoreName)
    .branchKeyIdSupplier(branchKeyIdSupplier)
    .ttlSeconds(600)
    .cache(CacheType.builder() //OPTIONAL
```
Step 4: Create friendly names for each branch key

The following example creates friendly names for the two branch keys created in Step 1. The AWS Encryption SDK uses encryption contexts to map the friendly name that you define to the associated branch-key-id.

C# / .NET

```csharp
// Create encryption contexts for the two branch keys created in Step 1
var encryptionContextA = new Dictionary<string, string>();
{
    // We will encrypt with branchKeyTenantA
    "tenant", "TenantA"),
    "encryption", "context"),
    "is not", "secret"),
    "but adds", "useful metadata"),
    "that can help you", "be confident that"),
    "the data you are handling", "is what you think it is"
};
var encryptionContextB = new Dictionary<string, string>();
{
    // We will encrypt with branchKeyTenantB
    "tenant", "TenantB"),
    "encryption", "context"),
    "is not", "secret"),
    "but adds", "useful metadata"),
    "that can help you", "be confident that"),
    "the data you are handling", "is what you think it is"
};

// Instantiate the AWS Encryption SDK
var esdk = new ESDK(new AwsEncryptionSdkConfig());

var encryptInputA = new EncryptInput
{
    Plaintext = plaintext,
```
Keyring = keyring,
// Encrypt with branchKeyId1
EncryptionContext = encryptionContextA
;

var encryptInputB = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    // Encrypt with branchKeyId2
    EncryptionContext = encryptionContextB
};

var encryptOutput = esdk.Encrypt(encryptInputA);
encryptOutput = esdk.Encrypt(encryptInputB);

// Use the encryption contexts to define friendly names for each branch key
public class ExampleBranchKeySupplier : IBranchKeyIdSupplier
{
    private string branchKeyTenantA;
    private string branchKeyTenantB;

    public ExampleBranchKeySupplier(string branchKeyTenantA, string
branchKeyTenantB)
    {
        this.branchKeyTenantA = branchKeyTenantA;
        this.branchKeyTenantB = branchKeyTenantB;
    }

    public GetBranchKeyIdOutput GetBranchKeyId(GetBranchKeyIdInput input)
    {
        Dictionary<string, string> encryptionContext = input.EncryptionContext;

        if (!encryptionContext.ContainsKey("tenant"))
        {
            throw new Exception("EncryptionContext invalid, does not contain expected tenant key value pair.");
        }

        string tenant = encryptionContext["tenant"];
        string branchKeyId;

        if (tenant.Equals("TenantA"))
        {
            branchKeyId = branchKeyTenantA;
        } else
        {
            branchKeyId = branchKeyTenantB;
        }

        return new GetBranchKeyIdOutput
        {
            BranchKeyId = branchKeyId
        };
    }
}
GetBranchKeyIdOutput output = new GetBranchKeyIdOutput();
output.BranchKeyId = branchKeyTenantA;
return output;
} else if (tenant.Equals("TenantB"))
{
    GetBranchKeyIdOutput output = new GetBranchKeyIdOutput();
    output.BranchKeyId = branchKeyTenantB;
    return output;
}
else
{
    throw new Exception("Item does not have a valid tenantID.");
}
}

Java

// Create encryption context for branchKeyTenantA
Map<String, String> encryptionContextA = new HashMap<>();
encryptionContextA.put("tenant", "TenantA");
encryptionContextA.put("encryption", "context");
encryptionContextA.put("is not", "secret");
encryptionContextA.put("but adds", "useful metadata");
encryptionContextA.put("that can help you", "be confident that");
encryptionContextA.put("the data you are handling", "is what you think it is");

// Create encryption context for branchKeyTenantB
Map<String, String> encryptionContextB = new HashMap<>();
encryptionContextB.put("tenant", "TenantB");
encryptionContextB.put("encryption", "context");
encryptionContextB.put("is not", "secret");
encryptionContextB.put("but adds", "useful metadata");
encryptionContextB.put("that can help you", "be confident that");
encryptionContextB.put("the data you are handling", "is what you think it is");

// Instantiate the AWS Encryption SDK
final AwsCrypto crypto = AwsCrypto.builder().build();

final CryptoResult<byte[], ?> encryptResultA = crypto.encryptData(keyring,
plaintext, encryptionContextA);
final CryptoResult<byte[], ?> encryptResultB = crypto.encryptData(keyring, plaintext, encryptionContextB);

// Use the encryption contexts to define friendly names for each branch key
public class ExampleBranchKeyIdSupplier implements IBranchKeyIdSupplier {
    private static String branchKeyIdForTenantA;
    private static String branchKeyIdForTenantB;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this.branchKeyIdForTenantA = tenant1Id;
        this.branchKeyIdForTenantB = tenant2Id;
    }

    @Override
    public GetBranchKeyIdOutput GetBranchKeyId(GetBranchKeyIdInput input) {

        Map<String, String> encryptionContext = input.encryptionContext();

        if (!encryptionContext.containsKey("tenant")) {
            throw new IllegalArgumentException("EncryptionContext invalid, does not contain expected tenant key value pair.");
        }

        String tenantKeyId = encryptionContext.get("tenant");
        String branchKeyId;

        if (tenantKeyId.equals("TenantA")) {
            branchKeyId = branchKeyIdForTenantA;
        } else if (tenantKeyId.equals("TenantB")) {
            branchKeyId = branchKeyIdForTenantB;
        } else {
            throw new IllegalArgumentException("Item does not contain valid tenant ID");
        }

        return GetBranchKeyIdOutput.builder().branchKeyId(branchKeyId).build();
    }
}
Raw AES keyrings

The AWS Encryption SDK lets you use an AES symmetric key that you provide as a wrapping key that protects your data key. You need to generate, store, and protect the key material, preferably in a hardware security module (HSM) or key management system. Use a Raw AES keyring when you need to provide the wrapping key and encrypt the data keys locally or offline.

The Raw AES keyring encrypts data by using the AES-GCM algorithm and a wrapping key that you specify as a byte array. You can specify only one wrapping key in each Raw AES keyring, but you can include multiple Raw AES keyrings, alone or with other keyrings, in a multi-keyring.

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

Key namespaces and names

To identify the AES key in a keyring, the Raw AES keyring uses a key namespace and key name that you provide. These values are not secret. They appear in plain text in the header of the encrypted message that the encrypt operation returns. We recommend using a key namespace your HSM or key management system and a key name that identifies the AES key in that system.

- **Note**

  The key namespace and key name are equivalent to the Provider ID (or Provider) and Key ID fields in the JceMasterKey and RawMasterKey.

  The AWS Encryption SDK for C and AWS Encryption SDK for .NET reserve the aws-kms key namespace value for KMS keys. Do not use this namespace value in a Raw AES keyring or Raw RSA keyring with these libraries.

If you construct different keyrings to encrypt and decrypt a given message, the namespace and name values are critical. If the key namespace and key name in the decryption keyring isn't an exact, case-sensitive match for the key namespace and key name in the encryption keyring, the decryption keyring isn't used, even if the key material bytes are identical.
For example, you might define a Raw AES keyring with key namespace HSM_01 and key name AES_256_012. Then, you use that keyring to encrypt some data. To decrypt that data, construct a Raw AES keyring with the same key namespace, key name, and key material.

The following examples show how to create a Raw AES keyring. The AESWrappingKey variable represents the key material you provide.

C

To instantiate a Raw AES keyring in the AWS Encryption SDK for C, use `aws_cryptosdk_raw_aes_keyring_new()`. For a complete example, see `raw_aes_keyring.c`.

```c
struct aws_allocator *alloc = aws_default_allocator();

AWS_STATIC_STRING_FROM_LITERAL(wrapping_key_namespace, "HSM_01");
AWS_STATIC_STRING_FROM_LITERAL(wrapping_key_name, "AES_256_012");

struct aws_cryptosdk_keyring *raw_aes_keyring = aws_cryptosdk_raw_aes_keyring_new(
    alloc, wrapping_key_namespace, wrapping_key_name, aes_wrapping_key,
    wrapping_key_len);
```

C#/.NET

To create a Raw AES keyring in AWS Encryption SDK for .NET, use the `materialProviders.CreateRawAesKeyring()` method. For a complete example, see `RawAESKeyringExample.cs`.

The following example uses version 4.x of the AWS Encryption SDK for .NET.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

// This example uses the key generator in Bouncy Castle to generate the key material.
// In production, use key material from a secure source.
var aesWrappingKey = new MemoryStream(GeneratorUtilities.GetKeyGenerator("AES256").GenerateKey());
```
// Create the keyring that determines how your data keys are protected.
var createKeyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = aesWrappingKey,
    WrappingAlg = AesWrappingAlg.AES256_GCM_IV12_TAG16
};

var keyring = materialProviders.CreateRawAesKeyring(createKeyringInput);

JavaScript Browser

The AWS Encryption SDK for JavaScript in the browser gets its cryptographic
primitives from the WebCrypto API. Before you construct the keyring, you must use
RawAesKeyringWebCrypto.importCryptoKey() to import the raw key material into
the WebCrypto backend. This assures that the keyring is complete even though all calls to
WebCrypto are asynchronous.

Then, to instantiate a Raw AES keyring, use the RawAesKeyringWebCrypto() method. You
must specify the AES wrapping algorithm ("wrapping suite) based on the length of your key
material. For a complete example, see aes_simple.ts (JavaScript Browser).

const keyNamespace = 'HSM_01'
const keyName = 'AES_256_012'

const wrappingSuite =
    RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

/* Import the plaintext AES key into the WebCrypto backend. */
const aesWrappingKey = await RawAesKeyringWebCrypto.importCryptoKey(
    rawAesKey,
    wrappingSuite
)

const rawAesKeyring = new RawAesKeyringWebCrypto(
    {
        keyName,
        keyNamespace,
        wrappingSuite,
        aesWrappingKey
    })
JavaScript Node.js

To instantiate a Raw AES keyring in the AWS Encryption SDK for JavaScript for Node.js, create an instance of the `RawAesKeyringNode` class. You must specify the AES wrapping algorithm ("wrapping suite") based on the length of your key material. For a complete example, see `aes_simple.ts` (JavaScript Node.js).

```javascript
const keyName = 'AES_256_012'
const keyNamespace = 'HSM_01'

const wrappingSuite =
    RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

const rawAesKeyring = new RawAesKeyringNode({
    keyName,
    keyNamespace,
    wrappingKey,
    wrappingSuite,
})
```

Java

To instantiate a Raw AES keyring in the AWS Encryption SDK for Java, use `matProv.CreateRawAesKeyring()`.

```java
final CreateRawAesKeyringInput keyringInput = CreateRawAesKeyringInput.builder()
    .keyName("AES_256_012")
    .keyNamespace("HSM_01")
    .wrappingKey(AESWrappingKey)
    .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
    .build();

final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();

IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

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 provide. You need to generate, store, and
protect the private key, preferably in a hardware security module (HSM) or key management system. 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. However, 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. You can include any Raw RSA keyring in a multi-keyring. If you configure a Raw RSA keyring with a public and private key, be sure that they are part of the same key pair. Some language implementations of the AWS Encryption SDK will not construct a Raw RSA keyring with keys from different pairs. Others rely on you to verify that your keys are from the same key pair.

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 RSA asymmetric encryption keys. You can encrypt data with one implementation and decrypt the data with any other implementation using the same wrapping key. For details, see Keyring compatibility.

Note

The Raw RSA keyring does not support asymmetric KMS keys. If you want to use asymmetric RSA KMS keys, version 4.x of the AWS Encryption SDK for .NET and version 3.x of the AWS Encryption SDK for Java support AWS KMS keyrings that use symmetric encryption (SYMMETRIC_DEFAULT) or asymmetric RSA AWS KMS keys.

If you encrypt data with a Raw RSA keyring that includes the public key of an RSA KMS key, neither the AWS Encryption SDK nor AWS KMS can decrypt it. You cannot export the private key of an AWS KMS asymmetric KMS key into a Raw RSA keyring. The AWS KMS Decrypt operation cannot decrypt the encrypted message that the AWS Encryption SDK returns.

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 with other language implementations.

Namespaces and names
To identify the RSA key material in a keyring, the Raw RSA keyring uses a *key namespace* and *key name* that you provide. These values are not secret. They appear in plain text in the header of the *encrypted message* that the encrypt operation returns. We recommend using the key namespace and key name that identifies the RSA key pair (or its private key) in your HSM or key management system.

**Note**

The key namespace and key name are equivalent to the *Provider ID* (or *Provider*) and *Key ID* fields in the JceMasterKey and RawMasterKey.

The AWS Encryption SDK for C reserves the `aws-kms` key namespace value for KMS keys. Do not use it in a Raw AES keyring or Raw RSA keyring with the AWS Encryption SDK for C.

If you construct different keyrings to encrypt and decrypt a given message, the namespace and name values are critical. If the key namespace and key name in the decryption keyring isn't an exact, case-sensitive match for the key namespace and key name in the encryption keyring, the decryption keyring isn't used, even if the keys are from the same key pair.

The key namespace and key name of the key material in the encryption and decryption keyrings must be same whether the keyring contains the RSA public key, the RSA private key, or both keys in the key pair. For example, suppose you encrypt data with a Raw RSA keyring for an RSA public key with key namespace HSM_01 and key name RSA_2048_06. To decrypt that data, construct a Raw RSA keyring with the private key (or key pair), and the same key namespace and name.

**Padding mode**

You must specify a padding mode for Raw RSA keyrings used for encryption and decryption, or use features of your language implementation that specify it for you.

The AWS Encryption SDK supports the following padding modes, subjects to the constraints of each language. We recommend an [OAEP](#) padding mode, particularly OAEP with SHA-256 and MGF1 with SHA-256 Padding. The [PKCS1](#) padding mode is supported only for backward compatibility.

- OAEP with SHA-1 and MGF1 with SHA-1 Padding
- OAEP with SHA-256 and MGF1 with SHA-256 Padding
- OAEP with SHA-384 and MGF1 with SHA-384 Padding
• OAEP with SHA-512 and MGF1 with SHA-512 Padding
• PKCS1 v1.5 Padding

The following examples show how to create a Raw RSA keyring with the public and private key of an RSA key pair and the OAEP with SHA-256 and MGF1 with SHA-256 padding mode. The RSAPublicKey and RSAPrivatekey variables represent the key material you provide.

**C**

To create a Raw RSA keyring in the AWS Encryption SDK for C, use `aws_cryptosdk_raw_rsa_keyring_new`.

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. For a complete example, see `raw_rsa_keyring.c`.

```c
struct aws_allocator *alloc = aws_default_allocator();
AWS_STATIC_STRING_FROM_LITERAL(key_namespace, "HSM_01");
AWS_STATIC_STRING_FROM_LITERAL(key_name, "RSA_2048_06");

struct aws_cryptosdk_keyring *rawRsaKeyring = aws_cryptosdk_raw_rsa_keyring_new(
    alloc,
    key_namespace,
    key_name,
    private_key_from_pem,
    public_key_from_pem,
    AWS_CRYPTOSDK_RSA_OAEP_SHA256_MGF1);
```

**C# / .NET**

To instantiate a Raw RSA keyring in the AWS Encryption SDK for .NET, use the `materialProviders.CreateRawRsaKeyring()` method. For a complete example, see `RawRSAKeyringExample.cs`.

The following example uses version 4.x of the AWS Encryption SDK for .NET.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());
```
```javascript
var keyNamespace = "HSM_01";
var keyName = "RSA_2048_06";

// Get public and private keys from PEM files
var publicKey = new MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePublicKey.pem"));
var privateKey = new MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePrivateKey.pem"));

// Create the keyring input
var createRawRsaKeyringInput = new CreateRawRsaKeyringInput {
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    PaddingScheme = PaddingScheme.OAEP_SHA512_MGF1,
    PublicKey = publicKey,
    PrivateKey = privateKey
};

// Create the keyring
var rawRsaKeyring = materialProviders.CreateRawRsaKeyring(createRawRsaKeyringInput);
```

**JavaScript Browser**

The AWS Encryption SDK for JavaScript in the browser gets its cryptographic primitives from the [WebCrypto](https://developer.mozilla.org/en-US/docs/Web/API/WebCrypto_API) library. Before you construct the keyring, you must use `importPublicKey()` and/or `importPrivateKey()` to import the raw key material into the WebCrypto backend. This assures that the keyring is complete even though all calls to WebCrypto are asynchronous. The object that the import methods take includes the wrapping algorithm and its padding mode.

After importing the key material, use the `RawRsaKeyringWebCrypto()` method to instantiate the keyring. When constructing a Raw RSA keyring in JavaScript, be aware of potential incompatibility with other language implementations.

For a complete example, see [rsa_simple.ts](javascriptbrowser) (JavaScript Browser).

```javascript
const privateKey = await RawRsaKeyringWebCrypto.importPrivateKey(
    privateRsaJwKKey
)

const publicKey = await RawRsaKeyringWebCrypto.importPublicKey(
```
JavaScript Node.js

To instantiate a Raw RSA keyring in AWS Encryption SDK for JavaScript for Node.js, create a new instance of the RawRsaKeyringNode class. The wrapKey parameter holds the public key. The unwrapKey parameter holds the private key. The RawRsaKeyringNode constructor calculates a default padding mode for you, although you can specify a preferred padding mode.

When constructing a raw RSA keyring in JavaScript, be aware of potential incompatibility with other language implementations.

For a complete example, see rsa_simple.ts (JavaScript Node.js).

Java

```
final CreateRawRsaKeyringInput keyringInput = CreateRawRsaKeyringInput.builder()
    .keyName("RSA_2048_06")
    .keyNamespace("HSM_01")
    .paddingScheme(PaddingScheme.OAEP_SHA256_MGF1)
    .publicKey(RSAPublicKey)
    .privateKey(RSAPrivateKey)
    .build();
final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build());
```
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, the generator key in the AWS KMS keyring generates and encrypts the plaintext key. Then, all additional AWS KMS keys 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.

Beginning in version 1.7.x, when an encrypted data key is encrypted under an AWS Key Management Service (AWS KMS) keyring (or master key provider), the AWS Encryption SDK always passes the key ARN of the AWS KMS key to the KeyId parameter of the AWS KMS Decrypt operation. This is an AWS KMS best practice that assures that you decrypt the encrypted data key with the wrapping key you intend to use.

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

- C: multi_keyring.cpp
- C#/.NET: MultiKeyringExample.cs
- JavaScript Node.js: multi_keyring.ts
- JavaScript Browser: multi_keyring.ts
- Java: MultiKeyringExample.java
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

/* Define an AWS KMS keyring. For details, see string.cpp */
struct aws_cryptosdk_keyring *kms_keyring =
  Aws::Cryptosdk::KmsKeyring::Builder().Build(example_key);

// 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);

C#/.NET

// Define an AWS KMS keyring. For details, see AwsKmsKeyringExample.cs.
var kmsKeyring = materialProviders.CreateAwsKmsKeyring(createKmsKeyringInput);

// Define a Raw AES keyring. For details, see RawAESKeyringExample.cs.
var aesKeyring = materialProviders.CreateRawAesKeyring(createAesKeyringInput);

JavaScript Browser

const clientProvider = getClient(KMS, { credentials })

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

// 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: exampleKey })

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

```java
// Define the raw AES keyring.
final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateRawAesKeyringInput createRawAesKeyringInput =
    CreateRawAesKeyringInput.builder()
    .keyName("AES_256_012")
    .keyNamespace("HSM_01")
    .wrappingKey(AESWrappingKey)
    .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
    .build();
IKKeyring rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// Define the AWS KMS keyring.
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
    CreateAwsKmsMrkMultiKeyringInput.builder()
    .generator(kmsKeyArn)
    .build();
IKKeyring awsKmsMrkMultiKeyring =
    matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

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.

```c
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.

```c
// Add the Raw AES keyring (C only)
aws_cryptosdk_multi_keyring_add_child(multi_keyring, aes_keyring);
```
C#/.NET

The .NET CreateMultiKeyringInput constructor lets you define a generator keyring and child keyrings. The resulting CreateMultiKeyringInput object is immutable.

```csharp
var createMultiKeyringInput = new CreateMultiKeyringInput
{
    Generator = kmsKeyring,
    ChildKeyrings = new List<IKeyring>() {aesKeyring}
};

var multiKeyring = materialProviders.CreateMultiKeyring(createMultiKeyringInput);
```

JavaScript Browser

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

```javascript
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]);
```

Java

The Java CreateMultiKeyringInput constructor lets you define a generator keyring and child keyrings. The resulting createMultiKeyringInput object is immutable.

```java
final CreateMultiKeyringInput createMultiKeyringInput = CreateMultiKeyringInput.builder()
    .generator(awsKmsMrkMultiKeyring)
    .childKeyrings(Collections.singletonList(rawAesKeyring))
    .build();
```
IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

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

Topics

- AWS Encryption SDK for C
- AWS Encryption SDK for .NET
- AWS Encryption SDK for Java
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK for Python
- AWS Encryption SDK command line interface

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, 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, including the AWS Encryption CLI.

The AWS Encryption SDK for C requires the AWS SDK for C++ to interact with AWS Key Management Service (AWS KMS). You need to use it only if you're using the optional AWS KMS keyring. 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, 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
• Using the AWS Encryption SDK for C
• AWS Encryption SDK for C examples

Installing the AWS Encryption SDK for C

Install the latest version of the AWS Encryption SDK for C.

⚠️ Note

All versions of the AWS Encryption SDK for C earlier than 2.0.0 are in the end-of-support phase.
You can safely update from version 2.0.x and later to the latest version of the AWS Encryption SDK for C without any code or data changes. However, new security features introduced in version 2.0.x are not backward-compatible. To update from versions earlier than 1.7.x to version 2.0.x and later, you must first update to the latest 1.x version of the AWS Encryption SDK for C. For details, see Migrating your AWS Encryption SDK.

You can find detailed instructions for installing and building the AWS Encryption SDK for C in the README file of the aws-encryption-sdk-c repository. It includes instructions for building on Amazon Linux, Ubuntu, macOS, and Windows platforms.

Before you begin, decide whether you want to use AWS KMS keyrings in the AWS Encryption SDK. If you use an AWS KMS keyring, you need to install the AWS SDK for C++. The AWS SDK is required to interact with AWS Key Management Service (AWS KMS). When you use AWS KMS keyrings, the
AWS Encryption SDK uses AWS KMS to generate and protect the encryption keys that protect your data.

You do not need to install the AWS SDK for C++ if you are using another keyring type, such as a raw AES keyring, a raw RSA keyring, or a multi-keyring that doesn't include an AWS KMS keyring. However, when using a raw keyring type, you need to generate and protect your own raw wrapping keys.

For help deciding which keyring types to use, see the section called “Choosing a keyring”.

If you're having trouble with your installation, file an issue in the aws-encryption-sdk-c repository or use any of the feedback links on this page.

**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.

The examples in this section show how to use version 2.0.x and later of the AWS Encryption SDK for C. For examples that use earlier versions, find your release in the Releases list of the aws-encryption-sdk-c repository repository on GitHub.

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

See also: Using keyrings

**Topics**

- Patterns for encrypting and decrypting data
- Reference counting

**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, create a CMM that uses the keyring, create a session that uses the CMM (and keyring), and then process the session.
1. Load error strings.

   Call the `aws_cryptosdk_load_error_strings()` method in your C or C++ code. It loads error information that is very useful for debugging.

   You only need to call it once, such as in your `main` method.

   ```c
   /* Load error strings for debugging */
   aws_cryptosdk_load_error_strings();
   ```

2. Create a keyring.

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

   To identify an AWS KMS key in an encryption keyring in the AWS Encryption SDK for C, specify a key ARN or alias ARN. In a decryption keyring, you must use a key ARN. For details, see Identifying AWS KMS keys in an AWS KMS keyring.

   ```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);
   ```

3. 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`. 
struct aws_cryptosdk_session * session =
    aws_cryptosdk_session_new_from_keyring_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
        kms_keyring);

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 can call `aws_cryptosdk_session_process_full`. However, if you need to handle streaming data, you can call `aws_cryptosdk_session_process` in a loop. For an example, see the `file_streaming.cpp` example. The `aws_cryptosdk_session_process_full` is introduced in AWS Encryption SDK versions 1.9.x and 2.2.x.

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 that the encrypt method returns.

```c
/* Encrypting data */
aws_cryptosdk_session_process_full(session,
    ciphertext,
    ciphertext_buffer_size,
    &ciphertext_length,
    plaintext,
    plaintext_length)
```

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 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_full` method.

```c
/* Decrypting data */
aws_cryptosdk_session_process_full(session,
    plaintext,
    plaintext_buffer_size,
    &plaintext_length,
```
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**, such as creating a session with a keyring
- A default **cryptographic materials manager** (CMM), such as creating a session or custom CMM with a default CMM
- A **data key cache**, such as creating a caching CMM with a keyring and cache

Unless you need an independent reference to the child object, you can release your reference to the child object as soon as you create the parent object. The remaining reference to the child object is released when the parent object is destroyed. This pattern ensures that you maintain the reference to each object only for as long as you need it, and you don't leak memory due to unreleased references.

You are only responsible for releasing references 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**, 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.

```
// The session gets a reference to the keyring.
```
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(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. 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.

// 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_2(allocator,
    AWS_CRYPTOSDK_ENCRYPT, caching_cmm);

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

// ...
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.

The examples in this section show how to use versions 2.0.x and later of the AWS Encryption SDK for C. For examples that use earlier versions, find your release in the Releases list of the aws-encryption-sdk-c repository repository on GitHub.

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

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 AWS KMS keyring, a type of keyring that uses an AWS KMS key in AWS Key Management Service (AWS KMS) to generate and encrypt data keys. The example includes code written in C++. The AWS Encryption SDK for C requires the AWS SDK for C++ to call AWS KMS when using AWS KMS keyrings. If you're using a keyring that doesn't interact with AWS KMS, such as a raw AES keyring, a raw RSA keyring, or a multi-keyring that doesn't include an AWS KMS keyring, the AWS SDK for C++ is not required.

For help creating an AWS KMS key, see Creating Keys in the AWS Key Management Service Developer Guide. For help identifying the AWS KMS keys in an AWS KMS keyring, see Identifying AWS KMS keys in an AWS KMS keyring.

See the complete code sample: string.cpp

Topics

• Encrypt a string
• Decrypt a string

Encrypt a string

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

Step 1. Load error strings.

Call the `aws_cryptosdk_load_error_strings()` method in your C or C++ code. It loads error information that is very useful for debugging.

You only need to call it once, such as in your `main` method.

```c
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();
```

Step 2: Construct the keyring.

Create an AWS KMS keyring for encryption. The keyring in this example is configured with one AWS KMS key, but you can configure an AWS KMS keyring with multiple AWS KMS keys, including AWS KMS keys in different AWS Regions and different accounts.

To identify an AWS KMS key in an encryption keyring in the AWS Encryption SDK for C, specify a key ARN or alias ARN. In a decryption keyring, you must use a key ARN. For details, see Identifying AWS KMS keys in an AWS KMS keyring.

**Identifying AWS KMS keys in an AWS KMS keyring**

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

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 3: 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 technique helps to prevent memory leaks and to prevent the objects from being released while they are in use.

```
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(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 4: Set the encryption context.

An encryption context 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.

```
/* 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)
```
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 aws_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_full method with the session in encryption mode. This method, introduced in AWS Encryption SDK versions 1.9.x and 2.2.x, is designed for non-streaming encryption and decryption. To handle streaming data, call the aws_cryptosdk_session_process in a loop.

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, 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 ciphertext_len_output;
if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
    ciphertext_output,
    ciphertext_buf_sz_output,
    &ciphertext_len_output,
    plaintext_input,
    plaintext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 8;
}
```
Step 6: Clean up the session.

The final step destroys the session including 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`.

Decrypt a string

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

Step 1: Load error strings.

Call the `aws_cryptosdk_load_error_strings()` method in your C or C++ code. It loads error information that is very useful for debugging.

You only need to call it once, such as in your main method.

```c
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();
```

Step 2: Construct the keyring.

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

On decrypt, you can configure a keyring with only the AWS KMS keys that you want to use to decrypt the encrypted message. For example, you might want to create a keyring with only the AWS KMS key that is used by a particular role in your organization. The AWS Encryption SDK will never use an AWS KMS key unless it appears in the decryption keyring. If the SDK can't decrypt the encrypted data keys by using the AWS KMS keys in the keyring that you provide, either because none of AWS KMS keys in the keyring were used to encrypt any of the data keys, or because the caller doesn't have permission to use the AWS KMS keys in the keyring to decrypt, the decrypt call fails.
When you specify an AWS KMS key for a decryption keyring, you must use its key ARN. Alias ARNs are permitted only in encryption keyrings. For help identifying the AWS KMS keys in an AWS KMS keyring, see Identifying AWS KMS keys in an AWS KMS keyring.

In this example, we specify a keyring configured with the same AWS KMS key 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 3: 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_2(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 4: Decrypt the string.

To decrypt the string, use the aws_cryptosdk_session_process_full method with the session that is configured for decryption. This method, introduced in AWS Encryption SDK versions 1.9.x and 2.2.x, is designed for non-streaming encryption and decryption. To handle streaming data, call the aws_cryptosdk_session_process in a loop.

When decrypting, the ciphertext fields are input fields and the plaintext fields are output fields. The ciphertext_input field holds the encrypted message that the encrypt method returned.
When the processing is complete, the `plaintext_output` field contains the plaintext (decrypted) string.

```c
size_t plaintext_len_output;

if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
    plaintext_output,
    plaintext_buf_sz_output,
    &plaintext_len_output,
    ciphertext_input,
    ciphertext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 13;
}
```

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** (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 encrypt 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);
Step 6: Clean up the session.

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

```c
aws_cryptosdk_session_destroy(session);
```

**AWS Encryption SDK for .NET**

The AWS Encryption SDK for .NET is a client-side encryption library for developers who are writing applications in C# and other .NET programming languages. It is supported on Windows, macOS, and Linux.

All [programming language](#) implementations of the AWS Encryption SDK are fully interoperable. However, if you encrypt data using the [required encryption context CMM](#) in version 4.x of the AWS Encryption SDK for .NET, you can only decrypt it with version 4.x of the AWS Encryption SDK for .NET or version 3.x of the AWS Encryption SDK for Java.

**Note**

Version 4.0.0 of the AWS Encryption SDK for .NET deviates from the AWS Encryption SDK Message Specification. As a result, messages encrypted by version 4.0.0 can only be decrypted by version 4.0.0 or later of the AWS Encryption SDK for .NET. They cannot be decrypted by any other programming language implementation.

Version 4.0.1 of the AWS Encryption SDK for .NET writes messages according to the AWS Encryption SDK Message Specification, and is interoperable with other programming
language implementations. By default, version 4.0.1 can read messages encrypted by version 4.0.0. However, if you do not want to decrypt messages encrypted by version 4.0.0, you can specify the `NetV4_0_0_RetryPolicy` property to prevent the client from reading these messages. For more information, see the v4.0.1 release notes in the aws-encryption-sdk-dafny repository on GitHub.

The AWS Encryption SDK for .NET differs from some of the other programming language implementations of the AWS Encryption SDK in the following ways:

- No support for data key caching

  **Note**

  Version 4.x of the AWS Encryption SDK for .NET supports the AWS KMS Hierarchical keyring, an alternative cryptographic materials caching solution.

- No support for streaming data
- No logging or stack traces from the AWS Encryption SDK for .NET
- Requires the AWS SDK for .NET

The AWS Encryption SDK for .NET includes all of the security features introduced in versions 2.0.x and later of other language implementations of the AWS Encryption SDK. However, if you are using the AWS Encryption SDK for .NET to decrypt data that was encrypted by a pre-2.0.x version another language implementation of the AWS Encryption SDK, you might need to adjust your commitment policy. For details, see How to set your commitment policy.

The AWS Encryption SDK for .NET is a product of the AWS Encryption SDK in Dafny, a formal verification language in which you write specifications, the code to implement them, and the proofs to test them. The result is a library that implements the features of the AWS Encryption SDK in a framework that assures functional correctness.

**Learn More**

- For examples showing how to configure options in the AWS Encryption SDK, such as specifying an alternate algorithm suite, limiting encrypted data keys, and using AWS KMS multi-Region keys, see Configuring the AWS Encryption SDK.
For details about programming with the AWS Encryption SDK for .NET, see the [aws-encryption-sdk-net](https://github.com/aws-encryption-sdk-dafny) directory of the aws-encryption-sdk-dafny repository on GitHub.

**Topics**

- Installing the AWS Encryption SDK for .NET
- Debugging the AWS Encryption SDK for .NET
- AWS KMS keyrings in the AWS Encryption SDK for .NET
- Required encryption contexts in version 4.x
- AWS Encryption SDK for .NET examples

**Installing the AWS Encryption SDK for .NET**

The AWS Encryption SDK for .NET is available as the `AWS.Cryptography.EncryptionSDK` package in NuGet. For details about installing and building the AWS Encryption SDK for .NET, see the [README.md](https://github.com/aws-encryption-sdk-net) file in the aws-encryption-sdk-net repository.

**Version 3.x**

Version 3.x of the AWS Encryption SDK for .NET supports .NET Framework 4.5.2 – 4.8 only on Windows. It supports .NET Core 3.0+ and .NET 5.0 and later on all supported operating systems.

**Version 4.x**

Version 4.x of the AWS Encryption SDK for .NET supports .NET 6.0 and .NET Framework net48 and later.

The AWS Encryption SDK for .NET requires the AWS SDK for .NET even if you aren't using AWS Key Management Service (AWS KMS) keys. It's installed with the NuGet package. However, unless you are using AWS KMS keys, AWS Encryption SDK for .NET does not require an AWS account, AWS credentials, or interaction with any AWS service. For help setting up an AWS account if you need it, see [Using the AWS Encryption SDK with AWS KMS](https).

**Debugging the AWS Encryption SDK for .NET**

The AWS Encryption SDK for .NET does not generate any logs. Exceptions in the AWS Encryption SDK for .NET generate an exception message, but no stack traces.
To help you debug, be sure to enable logging in the AWS SDK for .NET. The logs and error messages from the AWS SDK for .NET can help you distinguish errors arising in the AWS SDK for .NET from those in the AWS Encryption SDK for .NET. For help with AWS SDK for .NET logging, see AWSLogging in the AWS SDK for .NET Developer Guide. (To see the topic, expand the Open to view .NET Framework content section.)

AWS KMS keyrings in the AWS Encryption SDK for .NET

The basic AWS KMS keyrings in the AWS Encryption SDK for .NET take only one KMS key. They also require an AWS KMS client, which gives you an opportunity to configure the client for the AWS Region of the KMS key.

To create a AWS KMS keyring with one or more wrapping keys, use a multi-keyring. The AWS Encryption SDK for .NET has a special multi-keyring that takes one or more AWS KMS keys, and a standard multi-keyring that takes one or more keyrings of any supported type. Some programmers prefer to use a multi-keyring method to create all of their keyrings, and the AWS Encryption SDK for .NET supports that strategy.

The AWS Encryption SDK for .NET provides basic single-key keyrings and multi-keyrings for all typical use-cases, including AWS KMS multi-Region keys.

For example, to create a AWS KMS keyring with one AWS KMS key, you can use the CreateAwsKmsKeyring() method.

Version 3.x

The following example uses version 3.x of the AWS Encryption SDK for .NET to create a default AWS KMS client for the Region that contains the specified key.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

string keyArn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKey = new KmsKeyInput
    {
        KeyArn = keyArn,
    }
};
```

AWS KMS keyrings

147
KmsKeyId = keyArn
};

// Create the keyring
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);

Version 4.x

The following example uses version 4.x of the AWS Encryption SDK for .NET to create an AWS KMS client for the Region that contains the specified key.

// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

string keyArn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate the keyring input object
var createKeyringInput = new CreateAwsKmsKeyringInput{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = kmsArn
};

// Create the keyring
var kmsKeyring = mpl.CreateAwsKmsKeyring(createKeyringInput);

To create a keyring with one or more AWS KMS keys, use the CreateAwsKmsMultiKeyring() method. This example uses two AWS KMS keys. To specify one KMS key, use only the Generator parameter. The KmsKeyIds parameter that specifies additional KMS keys is optional.

The input for this keyring doesn't take an AWS KMS client. Instead, the AWS Encryption SDK uses the default AWS KMS client for each Region represented by a KMS key in the keyring. For example, if the KMS key identified by the value of the Generator parameter is in the US West (Oregon) Region (us-west-2), the AWS Encryption SDK creates a default AWS KMS client for the us-west-2 Region. If you need to customize the AWS KMS client, use the CreateAwsKmsKeyring() method.

The following example uses version 4.x of the AWS Encryption SDK for .NET and the CreateAwsKmsKeyring() method to customize the AWS KMS client.
Version 4.x of the AWS Encryption SDK for .NET supports AWS KMS keyrings that use symmetric encryption (SYMmetric_DEFAULT) or asymmetric RSA KMS keys. AWS KMS keyrings created with asymmetric RSA KMS keys can only contain one key pair.

To encrypt with an asymmetric RSA AWS KMS keyring, you do not need `kms:GenerateDataKey` or `kms:Encrypt` because you must specify the public key material that you want to use for encryption when you create the keyring. No AWS KMS calls are made when encrypting with this keyring. To decrypt with an asymmetric RSA AWS KMS keyring, you need `kms:Decrypt` permission.

To create an asymmetric RSA AWS KMS keyring, you must provide the public key and private key ARN from your asymmetric RSA KMS key. The public key must be PEM encoded. The following example creates an AWS KMS keyring with an asymmetric RSA key pair.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

string generatorKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate the keyring input object
var createEncryptKeyringInput = new CreateAwsKmsMultiKeyringInput
{
    Generator = generatorKey,
    KmsKeyIds = additionalKeys
};

var kmsEncryptKeyring =
    materialProviders.CreateAwsKmsMultiKeyring(createEncryptKeyringInput);
```
var createKeyringInput = new CreateAwsKmsRsaKeyringInput {
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = AWS KMS RSA private key ARN,
    PublicKey = publicKey,
    EncryptionAlgorithm = EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256
};

// Create the keyring
var kmsRsaKeyring = mpl.CreateAwsKmsRsaKeyring(createKeyringInput);

### Required encryption contexts in version 4.x

With version 4.x of the AWS Encryption SDK for .NET, you can use the required encryption context CMM to require encryption contexts in your cryptographic operations. An encryption context is a set of non-secret key–value pairs. The encryption context is cryptographically bound to the encrypted data so that the same encryption context is required to decrypt the field. When you use the required encryption context CMM, you can specify one or more required encryption context keys (required keys) that must be included in all encrypt and decrypt calls.

**Note**

The required encryption context CMM is only interoperable with version 3.x of the AWS Encryption SDK for Java. It is not interoperable with any other programming language implementation. If you encrypt data using the required encryption context CMM, you can only decrypt it with version 3.x of the AWS Encryption SDK for Java or version 4.x of the AWS Encryption SDK for .NET.

On encrypt, the AWS Encryption SDK verifies that all required encryption context keys are included in the encryption context that you specified. The AWS Encryption SDK signs the encryption contexts that you specified. Only the key-value pairs that are not required keys are serialized and stored in plaintext in the header of the encrypted message that the encrypt operation returns.

On decrypt, you must provide an encryption context that contains all of the key-value pairs that represent the required keys. The AWS Encryption SDK uses this encryption context and the key-value pairs stored in the encrypted message's header to reconstruct the original encryption context that you specified in the encrypt operation. If the AWS Encryption SDK cannot reconstruct the original encryption context, then the decrypt operation fails. If you provide a key-value pair that
contains the required key with an incorrect value, the encrypted message cannot be decrypted. You must provide the same key-value pair that was specified on encrypt.

⚠️ **Important**

Carefully consider which values you choose for the required keys in your encryption context. You must be able to provide the same keys and their corresponding values again on decrypt. If you're unable to reproduce the required keys, the encrypted message cannot be decrypted.

The following example initializes an AWS KMS keyring with the required encryption context CMM.

```csharp
var encryptionContext = new Dictionary<string, string>()
{
    {"encryption", "context"},
    {"is not", "secret"},
    {"but adds", "useful metadata"},
    {"that can help you", "be confident that"},
    {"the data you are handling", "is what you think it is"}
};

// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

// Instantiate the keyring input object
var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = kmsKey
};

// Create the keyring
var kmsKeyring = mpl.CreateAwsKmsKeyring(createKeyringInput);

var createCMMInput = new CreateRequiredEncryptionContextCMMInput
{
    UnderlyingCMM = mpl.CreateDefaultCryptographicMaterialsManager(new
    CreateDefaultCryptographicMaterialsManagerInput{Keyring = kmsKeyring}),
    // If you pass in a keyring but no underlying cmm, it will result in a failure
    // because only cmm is supported.
}```
If you use an AWS KMS keyring, the AWS Encryption SDK for .NET also uses the encryption context to provide additional authenticated data (AAD) in the calls the keyring makes to AWS KMS.

AWS Encryption SDK for .NET examples

The following examples show the basic coding patterns that you use when programming with the AWS Encryption SDK for .NET. Specifically, you instantiate the AWS Encryption SDK and the material providers library. Then, before calling each method, you instantiate an object that defines the input for the method. This is much like the coding pattern you use in the AWS SDK for .NET.

For examples showing how to configure options in the AWS Encryption SDK, such as specifying an alternate algorithm suite, limiting encrypted data keys, and using AWS KMS multi-Region keys, see Configuring the AWS Encryption SDK.

For more examples of programming with the AWS Encryption SDK for .NET, see the examples in the aws-encryption-sdk-net directory of the aws-encryption-sdk-dafny repository on GitHub.

Encrypting data in the AWS Encryption SDK for .NET

This example shows the basic pattern for encrypting data. It encrypts a small file with data keys that are protected by one AWS KMS wrapping key.

Step 1: Instantiate the AWS Encryption SDK and the material providers library.

Begin by instantiating the AWS Encryption SDK and the material providers library. You'll use the methods in the AWS Encryption SDK to encrypt and decrypt data. You'll use the methods in the material providers library to create the keyrings that specify which keys protect your data.

The way you instantiate the AWS Encryption SDK and the material providers library differs between versions 3.x and 4.x of the AWS Encryption SDK for .NET. All of the following steps are the same for both version 3.x and 4.x of the AWS Encryption SDK for .NET.

Version 3.x

```csharp
// Instantiate the AWS Encryption SDK and material providers
```
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

Version 4.x

// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

Step 2: Create an input object for the keyring.

Each method that creates a keyring has a corresponding input object class. For example, to create the input object for the CreateAwsKmsKeyring() method, create an instance of the CreateAwsKmsKeyringInput class.

Even though the input for this keyring doesn't specify a generator key, the single KMS key specified by the KmsKeyId parameter is the generator key. It generates and encrypts the data key that encrypts the data.

This input object requires an AWS KMS client for the AWS Region of the KMS key. To create an AWS KMS client, instantiate the AmazonKeyManagementServiceClient class in the AWS SDK for .NET. Calling the AmazonKeyManagementServiceClient() constructor with no parameters creates a client with the default values.

In an AWS KMS keyring used for encrypting with the AWS Encryption SDK for .NET, you can identify the KMS keys by using the key ID, key ARN, alias name, or alias ARN. In an AWS KMS keyring used for decrypting, you must use a key ARN to identify each KMS key. If you plan to reuse your encryption keyring for decrypting, use a key ARN identifier for all KMS keys.

string keyArn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
Step 3: Create the keyring.

To create the keyring, call the keyring method with the keyring input object. This example uses the `CreateAwsKmsKeyring()` method, which takes just one KMS key.

```csharp
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

Step 4: Define an encryption context.

An encryption context is an optional, but strongly recommended element of cryptographic operations in the AWS Encryption SDK. You can define one or more non-secret key-value pairs.

**Note**
With version 4.x of the AWS Encryption SDK for .NET, you can require an encryption context in all encrypt requests with the required encryption context CMM.

```csharp
// Define the encryption context
var encryptionContext = new Dictionary<string, string>()
{
    ["purpose", "test"]
};
```

Step 5: Create the input object for encrypting.

Before calling the `Encrypt()` method, create an instance of the `EncryptInput` class.

```csharp
string plaintext = File.ReadAllText("C:\Documents\CryptoTest\TestFile.txt");

// Define the encrypt input
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
};
```
Step 6: Encrypt the plaintext.

Use the Encrypt() method of the AWS Encryption SDK to encrypt the plaintext using the keyring you defined.

The EncryptOutput that the Encrypt() method returns has methods for getting the encrypted message (Ciphertext), encryption context, and algorithm suite.

```javascript
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

Step 7: Get the encrypted message.

The Decrypt() method in the AWS Encryption SDK for .NET takes the Ciphertext member of the EncryptOutput instance.

The Ciphertext member of the EncryptOutput object is the encrypted message, a portable object that includes the encrypted data, encrypted data keys, and metadata, including the encryption context. You can safely store the encrypted message for an extended time or submit it to the Decrypt() method to recover the plaintext.

```javascript
var encryptedMessage = encryptOutput.Ciphertext;
```

### Decrypting in strict mode in the AWS Encryption SDK for .NET

Best practices recommend that you specify the keys that you use to decrypt data, an option known as *strict mode*. The AWS Encryption SDK uses only the KMS keys that you specify in your keyring to decrypt the ciphertext. The keys in your decryption keyring must include at least one of the keys that encrypted the data.

This example shows the basic pattern of decrypting in strict mode with the AWS Encryption SDK for .NET.

Step 1: Instantiate the AWS Encryption SDK and material providers library.

```javascript
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());
```
Step 2: Create the input object for your keyring.

To specify the parameters for the keyring method, create an input object. Each keyring method in the AWS Encryption SDK for .NET has a corresponding input object. Because this example uses the CreateAwsKmsKeyring() method to create the keyring, it instantiates the CreateAwsKmsKeyringInput class for the input.

In a decryption keyring, you must use a key ARN to identify KMS keys.

```csharp
string keyArn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
```

Step 3: Create the keyring.

To create the decryption keyring, this example uses the CreateAwsKmsKeyring() method and the keyring input object.

```csharp
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

Step 4: Create the input object for decrypting.

To create the input object for the Decrypt() method, instantiate the DecryptInput class.

The Ciphertext parameter of the DecryptInput() constructor takes the Ciphertext member of the EncryptOutput object that the Encrypt() method returned. The Ciphertext property represents the encrypted message, which includes the encrypted data, encrypted data keys, and metadata that the AWS Encryption SDK needs to decrypt the message.

With version 4.x of the AWS Encryption SDK for .NET, you can use the optional EncryptionContext parameter to specify your encryption context in the Decrypt() method.

Use the EncryptionContext parameter to verify that the encryption context used on encrypt is included in the encryption context used to decrypt the ciphertext. The AWS Encryption SDK
adds pairs to the encryption context, including the digital signature if you're using an algorithm suite with signing, such as the default algorithm suite.

```csharp
var encryptedMessage = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = encryptedMessage,
    Keyring = keyring,
    EncryptionContext = encryptionContext // OPTIONAL
};
```

Step 5: Decrypt the ciphertext.

```csharp
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

Step 6: Verify the encryption context – Version 3.x

The `Decrypt()` method of version 3.x of the AWS Encryption SDK for .NET does not take an encryption context. It gets the encryption context values from the metadata in the encrypted message. However, before returning or using the plaintext, it's a best practice to verify that encryption context that was used to decrypt the ciphertext includes the encryption context you provided when encrypting.

Verify that the encryption context used on encrypt is included in the encryption context that used to decrypt the ciphertext. The AWS Encryption SDK adds pairs to the encryption context, including the digital signature if you're using an algorithm suite with signing, such as the default algorithm suite.

```csharp
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```
Decrypting with a discovery keyring in the AWS Encryption SDK for .NET

Rather than specifying the KMS keys for decryption, you can provide a AWS KMS *discovery keyring*, which is a keyring that doesn't specify any KMS keys. A discovery keyring lets the AWS Encryption SDK decrypt the data by using whichever KMS key encrypted it, provided that the caller has decrypt permission on the key. For best practices, add a discovery filter that limits the KMS keys that can be used to those in particular AWS accounts of a specified partition.

The AWS Encryption SDK for .NET provides a basic discovery keyring that requires an AWS KMS client and a discovery multi-keyring that requires you to specify one or more AWS Regions. The client and Regions both limit the KMS keys that can be used to decrypt the encrypted message. The input objects for both keyrings take the recommended discovery filter.

The following example shows the pattern for decrypting data with an AWS KMS discovery keyring and discovery filter.

Step 1: Instantiate the AWS Encryption SDK and the material providers library.

```csharp
// Instantiate the AWS Encryption SDK and material providers
var esdk = new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());
```

Step 2: Create the input object for the keyring.

To specify the parameters for the keyring method, create an input object. Each keyring method in the AWS Encryption SDK for .NET has a corresponding input object. Because this example uses the `CreateAwsKmsDiscoveryKeyring()` method to create the keyring, it instantiates the `CreateAwsKmsDiscoveryKeyringInput` class for the input.

```csharp
List<string> accounts = new List<string> { "111122223333" };

var discoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = accounts,
        Partition = "aws"
    }
};
```
Step 3: Create the keyring.

To create the decryption keyring, this example uses the `CreateAwsKmsDiscoveryKeyring()` method and the keyring input object.

```
var discoveryKeyring =
  materialProviders.CreateAwsKmsDiscoveryKeyring(discoveryKeyringInput);
```

Step 4: Create the input object for decrypting.

To create the input object for the `Decrypt()` method, instantiate the `DecryptInput` class. The value of the `Ciphertext` parameter is the `Ciphertext` member of the `EncryptOutput` object that the `Encrypt()` method returns.

With version 4.x of the AWS Encryption SDK for .NET, you can use the optional `EncryptionContext` parameter to specify your encryption context in the `Decrypt()` method.

Use the `EncryptionContext` parameter to verify that the encryption context used on encrypt is included in the encryption context used to decrypt the ciphertext. The AWS Encryption SDK adds pairs to the encryption context, including the digital signature if you're using an algorithm suite with signing, such as the default algorithm suite.

```
var ciphertext = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
  Ciphertext = ciphertext,
  Keyring = discoveryKeyring,
  EncryptionContext = encryptionContext // OPTIONAL
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

Step 5: Verify the encryption context – Version 3.x

The `Decrypt()` method of version 3.x of the AWS Encryption SDK for .NET does not take an encryption context on `Decrypt()`. It gets the encryption context values from the metadata in the encrypted message. However, before returning or using the plaintext, it's a best practice to verify that encryption context that was used to decrypt the ciphertext includes the encryption context you provided when encrypting.
Verify that the encryption context used on encrypt is included in the encryption context that was used to decrypt the ciphertext. The AWS Encryption SDK adds pairs to the encryption context, including the digital signature if you’re using an algorithm suite with signing, such as the default algorithm suite.

```java
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var 
decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```

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](https://github.com/aws/aws-encryption-sdk-java) repository on GitHub. For API documentation, see the [Javadoc](https://aws.amazon.com) for the AWS Encryption SDK for Java.

**Topics**

- [Prerequisites](#)
- [Installation](#)
- [AWS KMS keyrings in the AWS Encryption SDK for Java](#)
- [Required encryption contexts in version 3.x](#)
- [AWS Encryption SDK for Java examples](#)

**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](https://www.oracle.com/java/), 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**

Version 3.x of the AWS Encryption SDK for Java requires the AWS SDK for Java 2.x, even if you don't use AWS KMS keyrings.

Version 2.x or earlier of the AWS Encryption SDK for Java does not require the AWS SDK for Java. However, the AWS SDK for Java is required to use AWS Key Management Service (AWS KMS) as a master key provider. Beginning in the AWS Encryption SDK for Java version 2.4.0, the AWS Encryption SDK for Java supports both version 1.x and 2.x of the AWS SDK for Java. AWS Encryption SDK code for the AWS SDK for Java 1.x and 2.x are interoperable. For example, you can encrypt data with AWS Encryption SDK code that supports AWS SDK for Java 1.x and decrypt it using code that supports AWS SDK for Java 2.x (or vice versa). Versions of the AWS Encryption SDK for Java earlier than 2.4.0 support only AWS SDK for Java 1.x. For information about updating your version of the AWS Encryption SDK, see Migrating your AWS Encryption SDK.

When updating your AWS Encryption SDK for Java code from the AWS SDK for Java 1.x to AWS SDK for Java 2.x, replace references to the AWSKMS interface in AWS SDK for Java 1.x with references to the KmsClient interface in AWS SDK for Java 2.x. The AWS Encryption SDK for Java does not support the KmsAsyncClient interface. Also, update your code to use the AWS KMS-related objects in the kms-sdkv2 namespace, instead of the kms namespace.
To install the AWS SDK for Java, use Apache Maven.

- To import the entire AWS SDK for Java as a dependency, declare it in your `pom.xml` file.

- To create a dependency only for the AWS KMS module in AWS SDK for Java 1.x, follow the instructions for specifying particular modules, and set the `artifactId` to `aws-java-sdk-kms`.

- To create a dependency only for the AWS KMS module in AWS SDK for Java 2.x, follow the instructions for specifying particular modules. Set the `groupId` to `software.amazon.awssdk` and the `artifactId` to `kms`.

For more changes, see What's different between the AWS SDK for Java 1.x and 2.x in the AWS SDK for Java 2.x Developer Guide.

Java examples in the AWS Encryption SDK Developer Guide use the AWS SDK for Java 2.x.

**Installation**

Install the latest version of the AWS Encryption SDK for Java.

**Note**

All versions of the AWS Encryption SDK for Java earlier than 2.0.0 are in the end-of-support phase.

You can safely update from version 2.0.x and later to the latest version of the AWS Encryption SDK for Java without any code or data changes. However, new security features introduced in version 2.0.x are not backward-compatible. To update from versions earlier than 1.7.x to version 2.0.x and later, you must first update to the latest 1.x version of the AWS Encryption SDK. For details, see Migrating your AWS Encryption SDK.

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>3.0.0</version>
</dependency>
```

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

AWS KMS keyrings in the AWS Encryption SDK for Java

Version 3.x of the AWS Encryption SDK for Java uses keyrings to perform envelope encryption. The basic AWS KMS keyrings in the AWS Encryption SDK for Java take only one KMS key. They also require an AWS KMS client, which gives you an opportunity to configure the client for the AWS Region of the KMS key.

To create a AWS KMS keyring with one or more wrapping keys, use a multi-keyring. The AWS Encryption SDK for Java has a special multi-keyring that takes one or more AWS KMS keys, and a standard multi-keyring that takes one or more keyrings of any supported type. Some programmers prefer to use a multi-keyring method to create all of their keyrings, and the AWS Encryption SDK for Java supports that strategy.

The AWS Encryption SDK for Java provides basic single-key keyrings and multi-keyrings for all typical use-cases, including AWS KMS multi-Region keys.

For example, to create a AWS KMS keyring with one AWS KMS key, you can use the CreateAwsKmsKeyring() method.

```java
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder().build();
final MaterialProviders materialProviders = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();

// Create the keyring
```
To create a keyring with one or more AWS KMS keys, use the `CreateAwsKmsMultiKeyring()` method. This example uses two KMS keys. To specify one KMS key, use only the `generator` parameter. The `kmsKeyIds` parameter that specifies additional KMS keys is optional.

The input for this keyring doesn't take an AWS KMS client. Instead, the AWS Encryption SDK uses the default AWS KMS client for each Region represented by a KMS key in the keyring. For example, if the KMS key identified by the value of the `Generator` parameter is in the US West (Oregon) Region (`us-west-2`), the AWS Encryption SDK creates a default AWS KMS client for the `us-west-2` Region. If you need to customize the AWS KMS client, use the `CreateAwsKmsKeyring()` method.

```
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder().build();
final MaterialProviders materialProviders = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();

String generatorKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
List<String> additionalKey = Collections.singletonList("arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");

// Create the keyring
final CreateAwsKmsMultiKeyringInput keyringInput =
    CreateAwsKmsMultiKeyringInput.builder()
    .generator(generatorKey)
    .kmsKeyIds(additionalKey)
    .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);
```

AWS Encryption SDK for Java supports AWS KMS keyrings that use symmetric encryption (SYMMETRIC_DEFAULT) or asymmetric RSA KMS keys. AWS KMS keyrings created with asymmetric RSA KMS keys can only contain one key pair.

To encrypt with an asymmetric RSA AWS KMS keyring, you do not need `kms:GenerateDataKey` or `kms:Encrypt` because you must specify the public key material that you want to use for encryption.
when you create the keyring. No AWS KMS calls are made when encrypting with this keyring. To decrypt with an asymmetric RSA AWS KMS keyring, you need `kms:Decrypt` permission.

To create an asymmetric RSA AWS KMS keyring, you must provide the public key and private key ARN from your asymmetric RSA KMS key. The public key must be PEM encoded. The following example creates an AWS KMS keyring with an asymmetric RSA key pair.

```java
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder()
    // Specify algorithmSuite without asymmetric signing here
    //
    // ALG_AES_128_GCM_IV12_TAG16_NO_KDF("0x0014"),
    // ALG_AES_192_GCM_IV12_TAG16_NO_KDF("0x0046"),
    // ALG_AES_256_GCM_IV12_TAG16_NO_KDF("0x0078"),
    // ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256("0x0014"),
    // ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA256("0x0046"),
    // ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256("0x0078")
    .withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256)
    .build();

final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();

// Create a KMS RSA keyring.
// This keyring takes in:
// - kmsClient
// - kmsKeyId: Must be an ARN representing an asymmetric RSA KMS key
// - publicKey: A ByteBuffer of a UTF-8 encoded PEM file representing the public
//   key for the key passed into kmsKeyId
// - encryptionAlgorithm: Must be either RSAES_OAEP_SHA_256 or RSAES_OAEP_SHA_1
final CreateAwsKmsRsaKeyringInput createAwsKmsRsaKeyringInput =
    CreateAwsKmsRsaKeyringInput.builder()
    .kmsClient(KmsClient.create())
    .kmsKeyId(rsaKeyArn)
    .publicKey(publicKey)
    .encryptionAlgorithm(EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256)
    .build();
IKeyring awsKmsRsaKeyring =
    matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```
**Required encryption contexts in version 3.x**

With version 3.x of the AWS Encryption SDK for Java, you can use the required encryption context CMM to require encryption contexts in your cryptographic operations. An encryption context is a set of non-secret key–value pairs. The encryption context is cryptographically bound to the encrypted data so that the same encryption context is required to decrypt the field. When you use the required encryption context CMM, you can specify one or more *required encryption context keys* (required keys) that must be included in all encrypt and decrypt calls.

---

**Note**

The required encryption context CMM is only interoperable with version 4.x of the AWS Encryption SDK for .NET. It is not interoperable with any other programming language implementation. If you encrypt data using the required encryption context CMM, you can only decrypt it with version 3.x of the AWS Encryption SDK for Java or version 4.x of the AWS Encryption SDK for .NET.

---

On encrypt, the AWS Encryption SDK verifies that all required encryption context keys are included in the encryption context that you specified. The AWS Encryption SDK signs the encryption contexts that you specified. Only the key-value pairs that are not required keys are serialized and stored in plaintext in the header of the encrypted message that the encrypt operation returns.

On decrypt, you must provide an encryption context that contains all of the key-value pairs that represent the required keys. The AWS Encryption SDK uses this encryption context and the key-value pairs stored in the encrypted message’s header to reconstruct the original encryption context that you specified in the encrypt operation. If the AWS Encryption SDK cannot reconstruct the original encryption context, then the decrypt operation fails. If you provide a key-value pair that contains the required key with an incorrect value, the encrypted message cannot be decrypted. You must provide the same key-value pair that was specified on encrypt.

---

**Important**

Carefully consider which values you choose for the required keys in your encryption context. You must be able to provide the same keys and their corresponding values again on decrypt. If you’re unable to reproduce the required keys, the encrypted message cannot be decrypted.
The following example initializes an AWS KMS keyring with the required encryption context CMM.

```java
// Instantiate the AWS Encryption SDK
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

// Create your encryption context
final Map<String, String> encryptionContext = new HashMap<>();
encryptionContext.put("encryption", "context");
encryptionContext.put("is not", "secret");
encryptionContext.put("but adds", "useful metadata");
encryptionContext.put("that can help you", "be confident that");
encryptionContext.put("the data you are handling", "is what you think it is");

// Create a list of required encryption contexts
final List<String> requiredEncryptionContextKeys = Arrays.asList("encryption", "context");

// Create the keyring
final MaterialProviders materialProviders = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsKeyringInput keyringInput = CreateAwsKmsKeyringInput.builder()
    .kmsKeyId(keyArn)
    .kmsClient(KmsClient.create())
    .build();
IKeyring kmsKeyring = materialProviders.CreateAwsKmsKeyring(keyringInput);

// Create the required encryption context CMM
ICryptographicMaterialsManager cmm =
    materialProviders.CreateDefaultCryptographicMaterialsManager(
        CreateDefaultCryptographicMaterialsManagerInput.builder()
            .keyring(kmsKeyring)
            .build()
    );
ICryptographicMaterialsManager requiredCMM =
    materialProviders.CreateRequiredEncryptionContextCMM(
        CreateRequiredEncryptionContextCMMInput.builder()
            .requiredEncryptionContextKeys(requiredEncryptionContextKeys)
            .underlyingCMM(cmm)
            .build()
    );
```
AWS Encryption SDK for Java examples

The following examples show you how to use the AWS Encryption SDK for Java to encrypt and decrypt data. These examples show how to use version 3.x and later of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java replaces master key providers with keyrings. For examples that use earlier versions, find your release in the Releases list of the aws-encryption-sdk-java repository on GitHub.

Topics
- Encrypting and decrypting strings
- Encrypting and decrypting byte streams
- Encrypting and decrypting byte streams with a multi-keyring

Encrypting and decrypting strings

The following example shows you how to use version 3.x of the AWS Encryption SDK for Java to encrypt and decrypt strings. Before using the string, convert it into a byte array.

This example uses an AWS KMS keyring. When you encrypt with an AWS KMS keyring, you can use a key ID, key ARN, alias name, or alias ARN to identify the KMS keys. When decrypting, you must use a key ARN to identify KMS keys.

When you call the encryptData() method, it returns an encrypted message (CryptoResult) that includes the ciphertext, the encrypted data keys, and the encryption context. When you call getResult on the CryptoResult object, it returns a base-64-encoded string version of the encrypted message that you can pass to the decryptData() method.

Similarly, when you call decryptData(), the CryptoResult object it returns contains the plaintext message and an AWS KMS key ID. Before your application returns the plaintext, verify that the AWS KMS key ID and the encryption context in the encrypted message are the ones that you expect.

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
```
import com.amazonaws.encryptionsdk.CryptoResult;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;
import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.Collections;
import java.util.Map;

/**
 * Encrypts and then decrypts data using an AWS KMS Keyring.
 * 
 * Arguments:
 * 
 * <ol>
 *   <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS customer master
 *       key (CMK), see 'Viewing Keys' at
 *       http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html
 * </ol>
 */
public class BasicEncryptionKeyringExample {

    private static final byte[] EXAMPLE_DATA = "Hello World".getBytes(StandardCharsets.UTF_8);

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

        encryptAndDecryptWithKeyring(keyArn);
    }

    public static void encryptAndDecryptWithKeyring(final String keyArn) {
        // 1. Instantiate the SDK
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt commitment policy,
        // which means this client only encrypts using committing algorithm suites and enforces
        // that the client will only decrypt encrypted messages that were created with a committing
        // algorithm suite.
// This is the default commitment policy if you build the client with
// `AwsCrypto.builder().build()`
// or `AwsCrypto.standard()`.
final AwsCrypto crypto =
    AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

// 2. Create the AWS KMS keyring.
// This example creates a multi keyring, which automatically creates the KMS
// client.
final MaterialProviders materialProviders =
    MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsMultiKeyringInput keyringInput =
    CreateAwsKmsMultiKeyringInput.builder().generator(keyArn).build();
final IKeyring kmsKeyring =
    materialProviders.CreateAwsKmsMultiKeyring(keyringInput);

// 3. Create an encryption context
// We recommend using an encryption context whenever possible
// 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> encryptionContext =
    Collections.singletonMap("ExampleContextKey", "ExampleContextValue");

// 4. Encrypt the data
final CryptoResult<byte[], ?> encryptResult =
    crypto.encryptData(kmsKeyring, EXAMPLE_DATA, encryptionContext);
final byte[] ciphertext = encryptResult.getResult();

// 5. Decrypt the data
final CryptoResult<byte[], ?> decryptResult =
    crypto.decryptData(
        kmsKeyring,
        ciphertext,
        // Verify that the encryption context in the result contains the
        // encryption context supplied to the encryptData method
        encryptionContext);

// 6. Verify that the decrypted plaintext matches the original 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 uses a Raw AES keyring.

When encrypting, this example uses the
AwsCrypto.builder() .withEncryptionAlgorithm() method to specify an algorithm suite without digital signatures. When decrypting, to ensure that the ciphertext is unsigned, this example uses the createUnsignedMessageDecryptingStream() method. The createUnsignedMessageDecryptingStream() method, fails if it encounters a ciphertext with a digital signature.

If you're encrypting with the default algorithm suite, which includes digital signatures, use the createDecryptingStream() method instead, as shown in the next example.

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoAlgorithm;
import com.amazonaws.encryptionsdk.CryptoInputStream;
import com.amazonaws.encryptionsdk.jce.JceMasterKey;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import software.amazon.cryptography.materialproviders.model.AesWrappingAlg;
import software.amazon.cryptography.materialproviders.model.CreateRawAesKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
```
import java.security.SecureRandom;
import java.util.Collections;
import java.util.Map;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

/**
 * Encrypts and then decrypts a file under a random key.
 *
 * Arguments:
 * <ol>
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 *
 * This program demonstrates using a standard Java {@link SecretKey} object as a {@link
 * IKeyring} to encrypt and decrypt streaming data.
 */
public class FileStreamingKeyringExample {
    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 Raw Aes Keyring using the random key and an AES-GCM encryption
        // algorithm
        final MaterialProviders materialProviders = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateRawAesKeyringInput keyringInput = CreateRawAesKeyringInput.builder()
                .wrappingKey(ByteBuffer.wrap(cryptoKey.getEncoded()))
                .keyNamespace("Example")
                .keyName("RandomKey")
                .wrappingAlg(AesWrappingAlg.ALG_AES128_GCM_IV12_TAG16)
                .build();
IKeyring keyring = materialProviders.CreateRawAesKeyring(keyringInput);

// Instantiate the SDK.
// This builds the AwsCrypto client with the RequireEncryptRequireDecrypt commitment policy,
// which means this client only encrypts using committing algorithm suites and enforces
// that the client will only decrypt encrypted messages that were created with a committing
// algorithm suite.
// This is the default commitment policy if you build the client with
// `AwsCrypto.builder().build()`
// or `AwsCrypto.standard()`.
// This example encrypts with an algorithm suite that doesn’t include signing
for faster decryption,
// since this use case assumes that the contexts that encrypt and decrypt are equally trusted.
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
    .build();

// Create an encryption context to identify the 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(keyring, 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.
// Since the data was encrypted using an unsigned algorithm suite, use the recommended
// createUnsignedMessageDecryptingStream method, which only accepts unsigned messages.
  in = new FileInputStream(srcFile + ".encrypted");
  CryptoInputStream<JceMasterKey> decryptingStream =
  crypto.createUnsignedMessageDecryptingStream(keyring, in);
  // Does it contain the expected encryption context?
  if
  (!"FileStreaming".equals(decryptingStream.getCryptoResult().getEncryptionContext().get("Example"))
  {
    throw new IllegalStateException("Bad encryption context");
  }

  // Write the plaintext data to disk.
  out = new FileOutputStream(srcFile + ".decrypted");
  IOUtils.copy(decryptingStream, out);
  decryptingStream.close();
  out.close();
}

/**
 * In practice, this key would be saved in a secure location.
 * For this demo, we generate a new random key for each operation.
 */
private static SecretKey retrieveEncryptionKey() {
  SecureRandom rnd = new SecureRandom();
  byte[] rawKey = new byte[16]; // 128 bits
  rnd.nextBytes(rawKey);
  return new SecretKeySpec(rawKey, "AES");
}

Encrypting and decrypting byte streams with a multi-keyring

The following example shows you how to use the AWS Encryption SDK with a multi-keyring. When you use a multi-keyring to encrypt data, any of the wrapping keys in any of its keyrings can decrypt that data. This example uses an AWS KMS keyring and a Raw RSA keyring as the child keyrings.

This example encrypts with the default algorithm suite, which includes a digital signature. When streaming, the AWS Encryption SDK releases plaintext after integrity checks, but before it has verified the digital signature. To avoid using the plaintext until the signature is verified, this example buffers the plaintext, and writes it to disk only when decryption and verification are complete.
```java
package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoOutputStream;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateRawRsaKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;
import software.amazon.cryptography.materialproviders.model.PaddingScheme;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.security.GeneralSecurityException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.Collections;

/**
 * <p>
 * Encrypts a file using both AWS KMS Key and an asymmetric key pair.
 * </p>
 * <p>
 * Arguments:
 * <ol>
 * <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS key, see 'Viewing Keys' at http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html</li>
 * <li> Name of file containing plaintext data to encrypt</li>
 * </ol>
 * </p>
 */
```
You might use AWS Key Management Service (AWS KMS) for most encryption and decryption operations, but still want the option of decrypting your data offline independently of AWS KMS. This sample demonstrates one way to do this.

The sample encrypts data under both an AWS KMS key and an "escrowed" RSA key pair so that either key alone can decrypt it. You might commonly use the AWS KMS key for decryption. However, at any time, you can use the private RSA key to decrypt the ciphertext independent of AWS KMS.

This sample uses the RawRsaKeyring 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 EscrowedEncryptKeyringExample {
    private static ByteBuffer publicEscrowKey;
    private static ByteBuffer 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 key and the escrowed public key
        // 1. Instantiate the SDK
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt commitment policy,
    }

    private static void standardDecrypt(final String kmsArn, final String fileName) throws Exception {
        // Decrypt with the KMS key and the escrowed private key
    }

    private static void escrowDecrypt(final String fileName) throws Exception {
        // Decrypt with the escrowed private key
    }

    private static void generateEscrowKeyPair() {
        // Generate a new RSA key pair
    }
}
```
// which means this client only encrypts using committing algorithm suites and
// enforces
// that the client will only decrypt encrypted messages that were created with
// a committing
// algorithm suite.
// This is the default commitment policy if you build the client with
// 'AwsCrypto.builder().build()'
// or 'AwsCrypto.standard()'.
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

// 2. Create the AWS KMS keyring.
// This example creates a multi keyring, which automatically creates the KMS
client.
final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsMultiKeyringInput keyringInput =
CreateAwsKmsMultiKeyringInput.builder()
    .generator(kmsArn)
    .build();
IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

// 3. Create the Raw Rsa Keyring with Public Key.
final CreateRawRsaKeyringInput encryptingKeyringInput =
CreateRawRsaKeyringInput.builder()
    .keyName("Escrow")
    .keyNamespace("Escrow")
    .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
    .publicKey(publicEscrowKey)
    .build();
IKeyring rsaPublicKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);

final CreateMultiKeyringInput createMultiKeyringInput =
CreateMultiKeyringInput.builder()
    .generator(kmsKeyring)
    .childKeyrings(Collections.singletonList(rsaPublicKeyring))
    .build();
IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

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

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

private static void standardDecrypt(final String kmsArn, final String fileName) throws Exception {
    // Decrypt with the AWS KMS key and the escrow public key.

    // 1. Instantiate the SDK.
    // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt commitment policy,
    // which means this client only encrypts using committing algorithm suites and enforces
    // that the client will only decrypt encrypted messages that were created with a committing
    // algorithm suite. 
    // This is the default commitment policy if you build the client with
    // `AwsCrypto.builder().build()` 
    // or `AwsCrypto.standard()`.
    final AwsCrypto crypto = AwsCrypto.builder()
        .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
        .build();

    // 2. Create the AWS KMS keyring.
    // This example creates a multi keyring, which automatically creates the KMS client.
    final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
    final CreateAwsKmsMultiKeyringInput keyringInput = CreateAwsKmsMultiKeyringInput.builder()
        .generator(kmsArn)
        .build();
    IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);
// 3. Create the Raw Rsa Keyring with Public Key.
final CreateRawRsaKeyringInput encryptingKeyringInput =
    CreateRawRsaKeyringInput.builder()
    .keyName("Escrow")
    .keyNamespace("Escrow")
    .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
    .publicKey(publicEscrowKey)
    .build();

IKeyring rsaPublicKeyring =
    matProv.CreateRawRsaKeyring(encryptingKeyringInput);

final CreateMultiKeyringInput createMultiKeyringInput =
    CreateMultiKeyringInput.builder()
    .generator(kmsKeyring)
    .childKeyrings(Collections.singletonList(rsaPublicKeyring))
    .build();

IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

// 5. Decrypt the file
// To simplify this code example, we omit the encryption context. Production code should always
// use an encryption context.
final FileInputStream in = new FileInputStream(fileName + ".encrypted庭由grown");
final FileOutputStream out = new FileOutputStream(fileName + ".decrypted庭由grown");
// Since we are using a signing algorithm suite, we avoid streaming decryption
directly to the output file,
// to ensure that the trailing signature is verified before writing any untrusted plaintext to disk.
final ByteArrayOutputStream plaintextBuffer = new ByteArrayOutputStream();
final CryptoOutputStream<?> decryptingStream =
    crypto.createDecryptingStream(multiKeyring, plaintextBuffer);
IOUtils.copy(in, decryptingStream);
in.close();
decryptingStream.close();
final ByteArrayInputStream plaintextReader = new
    ByteArrayInputStream(plaintextBuffer.toByteArray());
IOUtils.copy(plaintextReader, out);
out.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 AWS KMS.
```java
// 1. Instantiate the SDK
final AwsCrypto crypto = AwsCrypto.standard();

// 2. Create the Raw Rsa Keyring with Private Key.
final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateRawRsaKeyringInput encryptingKeyringInput =
    CreateRawRsaKeyringInput.builder()
    .keyName("Escrow")
    .keyNamespace("Escrow")
    .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
    .publicKey(publicEscrowKey)
    .privateKey(privateEscrowKey)
    .build();
IKeyring escrowPrivateKeyring =
    matProv.CreateRawRsaKeyring(encryptingKeyringInput);

// 3. Decrypt the file
// To simplify this code example, we omit the encryption context. Production
code should always
// use an encryption context.
final FileInputStream in = new FileInputStream(fileName + ".encrypted");
final FileOutputStream out = new FileOutputStream(fileName + ".deescrowed");
final CryptoOutputStream<> decryptingStream =
    crypto.createDecryptingStream(escrowPrivateKeyring, 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 = RawRsaKeyringExample.getPEMPublicKey(keyPair.getPublic());
    privateEscrowKey = RawRsaKeyringExample.getPEMPrivateKey(keyPair.getPrivate());
}
```
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, 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”.

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” 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
- Installing the AWS Encryption SDK for JavaScript
- Modules in the AWS Encryption SDK for JavaScript
- AWS Encryption SDK for JavaScript examples

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. And you can use the AWS Encryption SDK for JavaScript to decrypt encrypted messages 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.

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. For a complete example, see `fallback.ts`.

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

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 and the README.md file in each of the modules in the aws-encryption-sdk-javascript repository on GitHub.

Note

All versions of the AWS Encryption SDK for JavaScript earlier than 2.0.0 are in the end-of-support phase. You can safely update from version 2.0.x and later to the latest version of the AWS Encryption SDK for JavaScript without any code or data changes. However, new security features introduced in version 2.0.x are not backward-compatible. To update from versions earlier than 1.7.x to version 2.0.x and later, you must first update to the latest 1.x version of the AWS Encryption SDK for JavaScript. For details, see Migrating your AWS Encryption SDK.

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](#) 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](#) in Node.js.
raw-rsa-keyring-node

Exports functions that support Raw RSA keyrings 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 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 in the browser.

raw-aes-keyring-browser

Exports functions that support Raw AES keyrings in the browser.

raw-rsa-keyring-browser

Exports functions that support Raw RSA keyrings in the browser.
Modules for all implementations

**cache-material**

Supports the data key caching feature. Provides code for assembling the cryptographic materials that are cached with each data key.

**kms-keyring**

Exports functions that support KMS keyrings.

**material-management**

Implements the cryptographic materials manager (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.

See the complete code samples: Node: kms_simple.ts, Browser: kms_simple.ts

Topics

- Encrypting data with an AWS KMS keyring
• Decrypting data with an AWS KMS keyring

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, a type of keyring that uses an AWS KMS key to generate and encrypt data keys. For help creating an AWS KMS key, see Creating Keys in the AWS Key Management Service Developer Guide. For help identifying the AWS KMS keys in an AWS KMS keyring, see Identifying AWS KMS keys in an AWS KMS keyring

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 key 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 AWS KMS key 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 AWS KMS keys for an encryption keyring in the AWS Encryption SDK for JavaScript, you can use any supported AWS KMS key identifier. 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 AWS KMS keys in the keyring.

Before running this code, replace the example AWS KMS key identifiers with valid identifiers. You must have the permissions required to use the AWS KMS keys 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 keys for the generator key and additional key. Then, create an AWS KMS keyring using the AWS KMS client and the AWS KMS keys.

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

### JavaScript Node.js

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

### Step 2: Set the encryption context.

An encryption context 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.
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 (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 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'

const { result } = await encrypt(keyring, plaintext, { encryptionContext: context })
```
Decoding 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”* example.

Step 1: Construct the keyring.

To decrypt the data, pass in the `encrypted message` (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 when decrypting. You can use the same keyring that was used to encrypt the data or a different keyring. To succeed, at least one AWS KMS key 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 an AWS KMS key for a decryption keyring in the AWS Encryption SDK for JavaScript, you must use the key ARN. Otherwise, the AWS KMS key is not recognized. For help identifying the AWS KMS keys in an AWS KMS keyring, see Identifying AWS KMS keys in an AWS KMS keyring

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

In this example, we create a keyring that includes only one of the AWS KMS keys 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 AWS KMS key.

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.
declare 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 AWS KMS keys from the encryption keyring.

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` 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

```javascript
const { plaintext, messageHeader } = await decrypt(keyring, result)
```
Step 3: Verify the encryption context.

The encryption context 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 (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

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

JavaScript Node.js

```javascript
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
- Installation
- AWS Encryption SDK for Python example code

Prerequisites

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

A supported version of Python

Python 3.6 or later is required by the AWS Encryption SDK for Python versions 3.1.0 and later.

Earlier versions of the AWS Encryption SDK support Python 2.7 and Python 3.4 and later, but we recommend that you use the latest version of the AWS Encryption SDK.

To download Python, see Python downloads.

The pip installation tool for Python

pip is included in Python 3.6 and later versions, although you might want to upgrade it. For more information about upgrading or installing pip, see Installation in the pip documentation.

Installation

Install the latest version of the AWS Encryption SDK for Python.
Note

All versions of the AWS Encryption SDK for Python earlier than 3.0.0 are in the end-of-support phase. You can safely update from version 2.0.x and later to the latest version of the AWS Encryption SDK without any code or data changes. However, new security features introduced in version 2.0.x are not backward-compatible. To update from versions earlier than 1.7.x to version 2.0.x and later, you must first update to the latest 1.x version of the AWS Encryption SDK. For details, see Migrating your AWS Encryption SDK.

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 AWS Encryption SDK for Python requires the cryptography library (pyca/cryptography) on all platforms. All versions of pip automatically install and build the cryptography library on Windows. pip 8.1 and later automatically 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.

Versions 1.10.0 and 2.5.0 of the AWS Encryption SDK for Python pin the cryptography dependency between 2.5.0 and 3.3.2. Other versions of the AWS Encryption SDK for Python install the latest version of cryptography. If you require a version of cryptography later than 3.3.2, we recommend that you use the latest major version of the AWS Encryption SDK for Python.

For the latest development version of the AWS Encryption SDK for Python, go to the aws-encryption-sdk-python repository in GitHub.

After you install the AWS Encryption SDK for Python, get started by looking at the Python example code 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.

The examples in this section show how to use version 2.0.x and later of the AWS Encryption SDK for Python. For examples that use earlier versions, find your release in the Releases list of the aws-encryption-sdk-python repository on GitHub.

Topics

- Encrypting and decrypting strings
- Encrypting and decrypting byte streams
- Encrypting and decrypting byte streams with multiple master key providers
- Using data key caching to encrypt messages

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

When encrypting, the StrictAwsKmsMasterKeyProvider constructor takes a key ID, key ARN, alias name, or alias ARN. When decrypting, it requires a key ARN. In this case, because the keyArn parameter is used for encrypting and decrypting, its value must be a key ARN. For information about IDs for AWS KMS keys, see Key identifiers in the AWS Key Management Service Developer Guide.

# Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License"). You
# may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# or in the "license" file accompanying this file. This file is
# distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF
# ANY KIND, either express or implied. See the License for the specific
# language governing permissions and limitations under the License.
import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy

def cycle_string(key_arn, source_plaintext, botocore_session=None):
    """Encrypts and then decrypts a string under an &KMS; key."
    
    :param str key_arn: Amazon Resource Name (ARN) of the &KMS; key
    :param bytes source_plaintext: Data to encrypt
    :param botocore_session: existing botocore session instance
    :type botocore_session: botocore.session.Session
    """
    client = aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

    # Create an AWS 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.StrictAwsKmsMasterKeyProvider(**kms_kwargs)

    # Encrypt the plaintext source data
    ciphertext, encryptor_header = client.encrypt(source=source_plaintext, key_provider=master_key_provider)

    # Decrypt the ciphertext
    cycled_plaintext, decrypted_header = client.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

Examples
# encryption context (no key pairs).
assert all(
    pair in decrypted_header.encryption_context.items()
    for pair in encryptor_header.encryption_context.items()
)

## 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.

When encrypting, this example uses an alternate algorithm suite without digital signatures (AES_256_GCM_HKDF_SHA512_COMMIT_KEY). This algorithm suite is appropriate when the users who are encrypting and decrypting data are equally trusted. Then, when decrypting, the example uses the decrypt-unsigned streaming mode, which fails if it encounters signed ciphertext. The decrypt-unsigned streaming mode is introduced in AWS Encryption SDK versions 1.9.x and 2.2.x.

```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 filecmp
import os

import aws_encryption_sdk
from aws_encryption_sdk.identifiers import Algorithm, CommitmentPolicy,
    EncryptionKeyType, WrappingAlgorithm
from aws_encryption_sdk.internal.crypto.wrapping_keys import WrappingKey
from aws_encryption_sdk.key_providers.raw import RawMasterKeyProvider

class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
```
provider_id = "static-random"

def __init__(self, **kwargs):  # pylint: disable=unused-argument
    """Initialize empty map of keys."""
    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.
    """
    # Set up an encryption client with an explicit commitment policy. Note that if you
do not explicitly choose a
    # commitment policy, REQUIRE_ENCRYPT_REQUIRE_DECRYPT is used by default.
    client =
    aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)
    # 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
# We can use an unsigning algorithm suite here under the assumption that the contexts that encrypt
# and decrypt are equally trusted.

```python
with open(source_plaintext_filename, "rb") as plaintext, open(ciphertext_filename, "wb") as ciphertext:
    with client.stream(
        algorithm=Algorithm.AES_256_GCM_HKDF_SHA512_COMMIT_KEY,
        mode="e",
        source=plaintext,
        key_provider=master_key_provider,
    ) as encryptor:
        for chunk in encryptor:
            ciphertext.write(chunk)
```

# Decrypt the ciphertext
# We can use the recommended "decrypt-unsigned" streaming mode since we encrypted with an unsigned algorithm suite.

```python
with open(ciphertext_filename, "rb") as ciphertext, open(cycled_plaintext_filename, "wb") as plaintext:
    with client.stream(mode="decrypt-unsigned", 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
```python
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 key and an RSA key pair as the master keys.

This example encrypts with the default algorithm suite, which includes a digital signature. When streaming, the AWS Encryption SDK releases plaintext after integrity checks, but before it has verified the digital signature. To avoid using the plaintext until the signature is verified, this example buffers the plaintext, and writes it to disk only when decryption and verification are complete.

```python
import filecmp
import os
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
import aws_encryption_sdk
from aws_encryption_sdk.identifiers import CommitmentPolicy, EncryptionKeyType, WrappingAlgorithm
from aws_encryption_sdk.internal.crypto.wrapping_keys import WrappingKey
from aws_encryption_sdk.key_providers.raw import RawMasterKeyProvider
```

Examples
class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
    """Randomly generates and provides 4096-bit RSA keys consistently per unique key id."""

    provider_id = "static-random"

    def __init__(self, **kwargs):  # pylint: disable=unused-argument
        """Initialize empty map of keys."""
        self._static_keys = {}

    def _get_raw_key(self, key_id):
        """Retrieves a static, randomly generated, RSA key for the specified key id.

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

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

    :param str key_arn: Amazon Resource Name (ARN) of the &KMS; key
    :param str source_plaintext_filename: Path to the plaintext file to encrypt and decrypt
    :param botocore_session: Botocore session to use for AWS KMS operations
    :returns: The decrypted content of the plaintext file
    :rtype: bytes
    """
```python
:param str source_plaintext_filename: Filename of file to encrypt
:param botocore_session: existing botocore session instance
:type botocore_session: botocore.session.Session

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

# Set up an encryption client with an explicit commitment policy. Note that if you
do not explicitly choose a
# commitment policy, REQUIRE_ENCRYPT_REQUIRE_DECRYPT is used by default.
client = aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

# Create an AWS KMS master key provider
kms_kwars = dict(key_ids=[key_arn])
if botocore_session is not None:
    kms_kwars["botocore_session"] = botocore_session
kms_master_key_provider = aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(**kms_kwars)

# 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)

# Add the static master key provider to the AWS KMS master key provider
# The resulting master key provider uses AWS KMS master keys to generate (and
# encrypt)
# data keys and static master keys to create an additional encrypted copy of each
data key.
kms_master_key_provider.add_master_key_provider(static_master_key_provider)

# Encrypt plaintext with both AWS KMS and static master keys
with open(source_plaintext_filename, "rb") as plaintext,
        open(ciphertext_filename, "wb") as ciphertext:
    with client.stream(source=plaintext, mode="e",
                      key_provider=kms_master_key_provider) as encryptor:
        for chunk in encryptor:
            ciphertext.write(chunk)

# Decrypt the ciphertext with only the AWS KMS master key
```
# Buffer the data in memory before writing to disk. This ensures verification of the
digital signature before returning plaintext.
with open(ciphertext_filename, "rb") as ciphertext,
open(cycled_kms_plaintext_filename, "wb") as plaintext:
    with client.stream(
        source=ciphertext, mode="d",
        key_provider=aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(**kms_kwargs)
    ) as kms_decryptor:
        plaintext.write(kms_decryptor.read())

# Decrypt the ciphertext with only the static master key
# Buffer the data in memory before writing to disk to ensure verification of the
signature before returning plaintext.
with open(ciphertext_filename, "rb") as ciphertext,
open(cycled_static_plaintext_filename, "wb") as plaintext:
    with client.stream(source=ciphertext, mode="d",
        key_provider=static_master_key_provider) as static_decryptor:
        plaintext.write(static_decryptor.read())

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

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

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

This very basic example creates a function that encrypts a fixed string. It lets you specify an AWS KMS key, the required cache size (capacity), and a maximum age value. For a more complex, real-world example of data key caching, see Data key caching example code.

Although it is optional, this example also uses an encryption context 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.

```
# 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
from aws_encryption_sdk import CommitmentPolicy

def encrypt_with_caching(kms_key_arn, max_age_in_cache, cache_capacity):
    """Encrypts a string using an &KMS; key and data key caching.

    :param str kms_key_arn: Amazon Resource Name (ARN) of the &KMS; 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
    """
```
```python
# 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"}

# Set up an encryption client with an explicit commitment policy. Note that if you
do not explicitly choose a
# commitment policy, REQUIRE_ENCRYPT_REQUIRE_DECRYPT is used by default.
client = aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

# Create a master key provider for the &KMS; key
key_provider = aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_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 = client.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.

Note

Versions of the AWS Encryption CLI earlier than 4.0.0 are in the end-of-support phase. You can safely update from version 2.1.x and later to the latest version of the AWS Encryption CLI without any code or data changes. However, new security features introduced in version 2.1.x are not backward-compatible. To update from version 1.7.x or earlier, you must first update to the latest 1.x version of the AWS Encryption CLI. For details, see Migrating your AWS Encryption SDK.

New security features were originally released in AWS Encryption CLI versions 1.7.x and 2.0.x. However, AWS Encryption CLI version 1.8.x replaces version 1.7.x and AWS Encryption CLI 2.1.x replaces 2.0.x. For details, see the relevant security advisory in the aws-encryption-sdk-cli repository on GitHub.

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, key commitment, and signing.

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

All language-specific implementations of the AWS Encryption SDK, including the AWS Encryption CLI, are interoperable. For example, you can encrypt data with the AWS Encryption SDK for Java 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
- How to use the AWS Encryption CLI
- Examples of the AWS Encryption CLI
- AWS Encryption SDK CLI syntax and parameter reference
- Versions of the AWS Encryption CLI

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
- Installing and updating the AWS Encryption CLI

Installing the prerequisites

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

Install the following prerequisites before you install the AWS Encryption CLI,

Python

Python 3.6 or later is required by the AWS Encryption CLI versions 4.1.0 and later.
Earlier versions of the AWS Encryption CLI support Python 2.7 and 3.4 and later, but we recommend that you use the latest version of the AWS Encryption CLI.

Python is included in most Linux and macOS installations, but you need to upgrade to Python 3.6 or later. We recommend that you use the latest version of Python. On Windows, you have to install Python; it is not installed by default. To download and install Python, see Python downloads.

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

```bash
python
```

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

```bash
python -V
```

On Windows, after you install Python, 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.

```powershell
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.

On Linux installations, versions of `pip` earlier than 8.1 can't build the `cryptography` library that the AWS Encryption CLI requires. If you choose not to update your `pip` version, you can install the build tools separately. For more information, see [Building cryptography on Linux](#).
AWS Command Line Interface

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

To use AWS KMS keys 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 and updating the AWS Encryption CLI

Install the latest version of the AWS Encryption CLI. When you use pip to install the AWS Encryption CLI, it automatically installs the libraries that the CLI needs, including the AWS Encryption SDK for Python, the Python cryptography library, and the AWS SDK for Python (Boto3).

Note

Versions of the AWS Encryption CLI earlier than 4.0.0 are in the end-of-support phase. You can safely update from version 2.1.x and later to the latest version of the AWS Encryption CLI without any code or data changes. However, new security features introduced in version 2.1.x are not backward-compatible. To update from version 1.7.x or earlier, you must first update to the latest 1.x version of the AWS Encryption CLI. For details, see Migrating your AWS Encryption SDK.

New security features were originally released in AWS Encryption CLI versions 1.7.x and 2.0.x. However, AWS Encryption CLI version 1.8.x replaces version 1.7.x and AWS Encryption CLI 2.1.x replaces 2.0.x. For details, see the relevant security advisory in the aws-encryption-sdk-cli repository on GitHub.

To install the latest version of the AWS Encryption CLI

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

To upgrade to the latest version of the AWS Encryption CLI

```
pip install --upgrade aws-encryption-sdk-cli
```
To find the version numbers of your AWS Encryption CLI and AWS Encryption SDK

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

The output lists the version numbers of both libraries.

```
aws-encryption-sdk-cli/2.1.0 aws-encryption-sdk/2.0.0
```

To upgrade to the latest version of the AWS Encryption CLI

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

Installing the AWS Encryption CLI also installs the latest version of the AWS SDK for Python (Boto3), if it's not already installed. If Boto3 is installed, the installer verifies the Boto3 version and updates it if required.

To find your installed version of Boto3

```
pip show boto3
```

To update to the latest version of Boto3

```
pip install --upgrade boto3
```

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 CLI

This topic explains how to use the parameters in the AWS Encryption CLI. For examples, see `Examples of the AWS Encryption CLI`. For complete documentation, see `Read the Docs`. The syntax shown in these examples is for AWS Encryption CLI version 2.1.x and later.
For an example showing how to use the security feature that limits encrypted data keys, see [Limiting encrypted data keys](#).

For an example showing how to use AWS KMS multi-Region keys, see [Using multi-Region AWS KMS keys](#).

**Topics**

- [How to encrypt and decrypt data](#)
- [How to specify wrapping keys](#)
- [How to provide input](#)
- [How to specify the output location](#)
- [How to use an encryption context](#)
- [How to specify a commitment policy](#)
- [How to store parameters in a configuration file](#)

### 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 wrapping key (or master key), such as an AWS KMS key in AWS Key Management Service (AWS KMS). If you are using a custom master key provider, you also need to specify the provider. You also specify output locations for the encrypted message and for metadata about the encryption operation. An encryption context is optional, but recommended.

In version 1.8.x, the --commitment-policy parameter is required when you use the --wrapping-keys parameter; otherwise it's not valid. Beginning in version 2.1.x, the --commitment-policy parameter is optional, but recommended.

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

The AWS Encryption CLI encrypts your data under a unique data key. Then it encrypts the data key under the wrapping keys you specify. It returns an encrypted message 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. You also specify the wrapping keys that the AWS Encryption CLI can use to decrypt the message, or tell the AWS Encryption CLI it can use any wrapping keys that encrypted the message.
Beginning in version 1.8.x, the --wrapping-keys parameter is optional when decrypting, but recommended. Beginning in version 2.1.x, the --wrapping-keys parameter is required when encrypting and decrypting.

When decrypting, you can use the `key` attribute of the --wrapping-keys parameter to specify the wrapping keys that decrypt your data. Specifying an AWS KMS wrapping key when decrypting is optional, but it's a best practice that prevents you from using a key you didn't intend to use. If you're using a custom master key provider, you must specify the provider and wrapping key.

If you don't use the `key` attribute, you must set the `discovery` attribute of the --wrapping-keys parameter to `true`, which lets the AWS Encryption CLI decrypt using any wrapping key that encrypted the message.

As a best practice, use the --max-encrypted-data-keys parameter to avoid decrypting a malformed message with an excessive number of encrypted data keys. Specify the expected number of encrypted data keys (one for each wrapping key used in encryption) or a reasonable maximum (such as 5). For details, see Limiting encrypted data keys.

The `--buffer` parameter returns plaintext only after all input is processed, including verifying the digital signature if one is present.

The `--decrypt-unsigned` parameter decrypts ciphertext and ensures that messages are unsigned before decryption. Use this parameter if you used the --algorithm parameter and selected an algorithm suite without digital signing to encrypt data. If the ciphertext is signed, decryption fails.

You can use --decrypt or --decrypt-unsigned for decryption but not both.

```bash
aws-encryption-cli --decrypt --input myEncryptedMessage \
   --wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
   --output myPlaintextData \
   --metadata-output ~/metadata \
   --max-encrypted-data-keys 1 \
   --buffer \
   --encryption-context purpose=test \
   --commitment-policy require-encrypt-require-decrypt
```
The AWS Encryption CLI uses the wrapping 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 wrapping keys

When you encrypt data in the AWS Encryption CLI, you need to specify at least one wrapping key (or master key). You can use AWS KMS keys in AWS Key Management Service (AWS KMS), wrapping keys from a custom master key provider, or both. The custom master key provider can be any compatible Python master key provider.

To specify wrapping keys in versions 1.8.x and later, use the `--wrapping-keys` parameter ( `-w`). The value of this parameter is a collection of attributes 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 `--wrapping-keys` parameter with a key attribute. Beginning in version 2.1.x, the `--wrapping-keys` parameter is also required in decrypt commands. When decrypting, the `--wrapping-keys` parameter must have a key attribute or a discovery attribute with a value of true (but not both). Other attributes are optional.

- **Custom master key provider.** You must specify a `--wrapping-keys` parameter in every command. The parameter value must have key and provider attributes.

You can include multiple `--wrapping-keys` parameters and multiple key attributes in the same command.

### Wrapping key parameter attributes

The value of the `--wrapping-keys` parameter consists of the following attributes and their values. A `--wrapping-keys` parameter (or `--master-keys` parameter) is required in all encrypt commands. Beginning in version 2.1.x, the `--wrapping-keys` parameter is also required when decrypting.

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

Use the `key` attribute to identify a wrapping key. When encrypting, the value can be any key identifier that the master key provider recognizes.

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

In an encrypt command, you must include at least one `key` attribute and value. To encrypt your data key under multiple wrapping keys, use multiple `key` attributes.

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

In encrypt commands that use AWS KMS keys, 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 an AWS KMS key, see [Key Identifiers](https://docs.aws.amazon.com/kms/latest/APIReference/API_KmsKey.html) in the *AWS Key Management Service Developer Guide*.

```
```

In decrypt commands that use a custom master key provider, `key` and `provider` attributes are required.

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

In decrypt commands that use AWS KMS, you can use the `key` attribute to specify the AWS KMS keys to use for decrypting, or the `discovery` attribute with a value of `true`, which lets the AWS Encryption CLI use any AWS KMS key that was used to encrypt the message. If you specify an AWS KMS key, it must be one of the wrapping keys used to encrypt the message.

Specifying the wrapping key is an [AWS Encryption SDK best practice](https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/best-practices.html). It assures that you use the AWS KMS key you intend to use.

In a decrypt command, the value of the `key` attribute must be a `key ARN`.

```
\ AWS KMS key
aws-encryption-cli --decrypt --wrapping-keys key=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```
Discovery: Use any AWS KMS key when decrypting

If you don't need to limit the AWS KMS keys to use when decrypting, you can use the discovery attribute with a value of true. A value of true allows the AWS Encryption CLI to decrypt using any AWS KMS key that encrypted the message. If you don't specify a discovery attribute, discovery is false (default). The discovery attribute is valid only in decrypt commands and only when the message was encrypted with AWS KMS keys.

The discovery attribute with a value of true is an alternative to using the key attribute to specify AWS KMS keys. When decrypting a message encrypted with AWS KMS keys, each --wrapping-keys parameter must have a key attribute or a discovery attribute with a value of true, but not both.

When discovery is true, it's a best practice to use the discovery-partition and discovery-account attributes to limit the AWS KMS keys used to those in the AWS accounts you specify. In the following example, the discovery attributes allow the AWS Encryption CLI to use any AWS KMS key in the specified AWS accounts.

```
aws-encryption-cli --decrypt --wrapping-keys \
    discovery=true \
    discovery-partition=aws \
    discovery-account=111122223333 \
    discovery-account=444455556666
```

Provider: Specify the master key provider

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

```
--wrapping-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 CLI repository.

Region: Specify an AWS Region

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

```
--encrypt --wrapping-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.

```
--wrapping-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 wrapping keys**

You can specify multiple wrapping keys (or **master keys**) in each command.

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

There are two ways to specify multiple wrapping keys:

- Include multiple **key** attributes in the **--wrapping-keys** parameter value.

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

--wrapping-keys key=$key_oregon key=$key_ohio
```
• Include multiple `--wrapping-keys` parameters in the same command. Use this syntax when the attribute values that you specify do not apply to all of the wrapping keys in the command.

```
--wrapping-keys region=us-east-2 key=alias/test_key \
--wrapping-keys region=us-west-1 key=alias/test_key
```

The **discovery** attribute with a value of `true` lets the AWS Encryption CLI use any AWS KMS key that encrypted the message. If you use multiple `--wrapping-keys` parameters in the same command, using `discovery=true` in any `--wrapping-keys` parameter effectively overrides the limits of the **key** attribute in other `--wrapping-keys` parameters.

For example, in the following command, the **key** attribute in the first `--wrapping-keys` parameter limits the AWS Encryption CLI to the specified AWS KMS key. However, the **discovery** attribute in the second `--wrapping-keys` parameter lets the AWS Encryption CLI use any AWS KMS key in the specified accounts to decrypt the message.

```
aws-encryption-cli --decrypt \
    --wrapping-keys key=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab \
    --wrapping-keys discovery=true \
        discovery-partition=aws \
        discovery-account=111122223333 \
        discovery-account=444455556666
```

**How to provide input**

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

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

You can provide input in any of the following ways:

• Use a file.

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

• Use a file name pattern.
--input testdir/*.xml

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

--input testdir --recursive

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

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

How to specify the output location

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

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

Note

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

You can the output location in several ways.

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

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

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

```
--output Test
```

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

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

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

```
--output -
```

How to use an encryption context

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

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

In an encrypt command

The encryption context that you specify in an encrypt command, along with any additional pairs that the CMM adds, is cryptographically bound to the encrypted data. It is also included (in plaintext) in the encrypted message that the command returns. If you are using an AWS KMS
key, 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 specify a commitment policy**

To set the commitment policy for the command, use the `--commitment-policy` parameter. This parameter is introduced in version 1.8.x. It is valid in encrypt and decrypt commands. The commitment policy you set is valid only for the command in which it appears. If you do not set a commitment policy for a command, the AWS Encryption CLI uses the default value.

For example, the following parameter value sets the commitment policy to `require-encrypt-allow-decrypt`, which always encrypts with key commitment, but will decrypt a ciphertext that was encrypted with or without key commitment.
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 an 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, key.conf, specifies two AWS KMS keys in different Regions.

```bash
--wrapping-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 key.conf file in an encrypt command.

Bash

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

PowerShell

```
PS C:\> aws-encryption-cli -e `@key.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.

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

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

Bash

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

PowerShell

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

Next: Try the AWS Encryption CLI examples

Examples of the AWS Encryption CLI

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 CLI. For a quick reference, see AWS Encryption SDK CLI syntax and parameter reference.

⚠️ Note

The following examples use the syntax for AWS Encryption CLI version 2.1.x.
For an example showing how to use the security feature that limits encrypted data keys, see
Limiting encrypted data keys.

For an example showing how to use AWS KMS multi-Region keys, see Using multi-Region AWS KMS keys.

Topics
• Encrypting a file
• Decrypting a file
• Encrypting all files in a directory
• Decrypting all files in a directory
• Encrypting and decrypting on the command line
• Using multiple master keys
• Encrypting and decrypting in scripts
• Using data key caching

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, encrypts the file contents under the data key, and then writes the encrypted message to a new file.

The first command saves the key ARN of an AWS KMS key in the $keyArn variable. When encrypting with an AWS KMS key, you can identify it by using a key ID, key ARN, alias name, or alias ARN. For details about the key identifiers for an AWS KMS 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 `--wrapping-keys` parameter, and its required `key` attribute, tell the command to use the AWS KMS 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.

This command also uses the `--commitment-policy` parameter to set the commitment policy explicitly. In version 1.8.x, this parameter is required when you use the `--wrapping-keys` parameter. Beginning in version 2.1.x, the `--commitment-policy` parameter is optional, but recommended.

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

Bash

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

$ aws-encryption-cli --encrypt
   --input hello.txt
   --wrapping-keys key=$keyArn
   --metadata-output ~/metadata
   --encryption-context purpose=test
   --commitment-policy require-encrypt-require-decrypt
   --output .
```

PowerShell

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

PS C:\> aws-encryption-cli --encrypt
   --input Hello.txt
   --wrapping-keys key=$keyArn
   --metadata-output $home\Metadata.txt
   --commitment-policy require-encrypt-require-decrypt
   --encryption-context purpose=test
```
When the encrypt command succeeds, it does not return any output. To determine whether the command succeeded, check the Boolean value in the `$?` 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**

```bash
$ echo $?
0
```

**PowerShell**

```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** that includes the ciphertext of the `hello.txt` file, an encrypted copy of the data key, and additional metadata, including the encryption context.

**Bash**

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

**PowerShell**

```powershell
PS C:\> dir
Directory: C:\TestCLI

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
```

Examples
Decrypting a file

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

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

The --wrapping-keys parameter with a key attribute specifies the wrapping key used to decrypt the encrypted message. In decrypt commands with AWS KMS keys, the value of the key attribute must be a key ARN. The --wrapping-keys parameter is required in a decrypt command. If you are using AWS KMS keys, you can use the key attribute to specify AWS KMS keys for decrypting or the discovery attribute with a value of true (but not both). If you are using a custom master key provider, the key and provider attributes are required.

The --commitment-policy parameter is optional beginning in version 2.1.x, but it is recommended. Using it explicitly makes your intent clear, even if you specify the default value, require-encrypt-require-decrypt.

The --encryption-context parameter is optional in the decrypt command, even when an encryption context 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.

As a best practice, use the --max-encrypted-data-keys parameter to avoid decrypting a malformed message with an excessive number of encrypted data keys. Specify the expected number of encrypted data keys (one for each wrapping key used in encryption) or a reasonable maximum (such as 5). For details, see Limiting encrypted data keys.

The --buffer returns plaintext only after all input is processed, including verifying the digital signature if one is present.
Bash

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

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

$ aws-encryption-cli --decrypt
  --input hello.txt.encrypted
  --wrapping-keys key=$keyArn
  --commitment-policy require-encrypt-require-decrypt
  --encryption-context purpose=test
  --metadata-output ~/metadata
  --max-encrypted-data-keys 1
  --buffer
  --output .
```

PowerShell

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

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

PS C:\> aws-encryption-cli --decrypt
  --input Hello.txt.encrypted
  --wrapping-keys key=$keyArn
  --commitment-policy require-encrypt-require-decrypt
  --encryption-context purpose=test
  --metadata-output $home\Metadata.txt
  --max-encrypted-data-keys 1
  --buffer
  --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
```
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 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.
The first command saves the Amazon Resource Name (ARN) of an AWS KMS key in the $keyArn 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 --wrapping-keys parameter, and its required key attribute, specify the wrapping key to use. The encrypt command includes an encryption context, 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.

The --commitment-policy parameter is optional beginning in version 2.1.x, but it is recommended. If the command or script fails because it cannot decrypt a ciphertext, the explicit commitment policy setting might help you to detect the problem quickly.

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

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

Bash

```
# To run this example, replace the fictitious key ARN with a valid master key identifier.
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```
$ aws-encryption-cli --encrypt \
  --input testdir --recursive\n  --wrapping-keys key=$keyArn \n  --encryption-context dept=IT \n  --commitment-policy require-encrypt-require-decrypt \n  --metadata-output ~/metadata \n  --output testenc

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

PowerShell

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

PS C:\> aws-encryption-cli --encrypt `n  --input .\TestDir --recursive `n  --wrapping-keys key=$keyArn `n  --encryption-context dept=IT `n  --commitment-policy require-encrypt-require-decrypt `n  --metadata-output .\Metadata\Metadata.txt `n  --output .\TestEnc

PS C:\> dir .\TestEnc

Directory: C:\TestEnc

<table>
<thead>
<tr>
<th>Mode</th>
<th>LastWriteTime</th>
<th>Length</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>----</td>
<td>--------------</td>
<td>--------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>-a---</td>
<td>9/17/2017 2:32 PM</td>
<td>2713</td>
<td>cool-new-thing.py.encrypted</td>
</tr>
<tr>
<td>-a---</td>
<td>9/17/2017 2:32 PM</td>
<td>620</td>
<td>Hello.txt.encrypted</td>
</tr>
<tr>
<td>-a---</td>
<td>9/17/2017 2:32 PM</td>
<td>585</td>
<td>Employees.csv.encrypted</td>
</tr>
</tbody>
</table>

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

PowerShell

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

Directory: C:\TestEnc

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

This decrypt command decrypts all of the files in the TestEnc directory and writes the plaintext files to the TestDec directory. The `--wrapping-keys` parameter with a `key` attribute and a `key ARN` value tells the AWS Encryption CLI which AWS KMS keys to use to decrypt the files. 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**

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

$ aws-encryption-cli --decrypt
   --input testenc --recursive
   --wrapping-keys key=$keyArn
   --encryption-context dept=IT
   --commitment-policy require-encrypt-require-decrypt
   --metadata-output ~/metadata
   --max-encrypted-data-keys 1
   --buffer
   --output testdec --interactive

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

**PowerShell**

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

PS C:\> aws-encryption-cli --decrypt
   --input C:\TestEnc --recursive
   --wrapping-keys key=$keyArn
   --encryption-context dept=IT
   --commitment-policy require-encrypt-require-decrypt
   --metadata-output $home\Metadata.txt
   --max-encrypted-data-keys 1
   --buffer
   --output C:\TestDec --interactive

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

**Examples**
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 key ARN of an AWS KMS key in the $keyArn variable.

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

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

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

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

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

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

Bash

$ encrypted=$(echo 'Hello World' | aws-encryption-cli --encrypt -S --input - --output - -- encode --wrapping-keys key= $keyArn )

PowerShell

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

• 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.)

This example uses the discovery attribute of the --wrapping-keys parameter to allow the AWS Encryption CLI to use any AWS KMS key to decrypt the data. It doesn't specify a commitment policy, so it uses the default value for version 2.1.x and later, require-encrypt-require-decrypt.
Because the output was encrypted and then encoded, the decrypt command uses the \texttt{--decode} parameter to decode Base64-encoded input before decrypting it. You can also use the \texttt{--decode} parameter to decode Base64-encoded input before encrypting it.

Again, the command omits the encryption context and suppresses the metadata (\texttt{-S}).

\textbf{Bash}

\begin{verbatim}
$ echo $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=true --input - --output - --decode --buffer -S
Hello World
\end{verbatim}

\textbf{PowerShell}

\begin{verbatim}
PS C:\> $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=$true --input - --output - --decode --buffer -S
Hello World
\end{verbatim}

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

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

\textbf{Bash}

\begin{verbatim}
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ echo 'Hello World' |
  aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --output - --encode -S |
  aws-encryption-cli --decrypt --wrapping-keys discovery=true --input - --output - --decode -S
Hello World
\end{verbatim}
PowerShell

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

PS C:\> 'Hello World' |
    aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --output - --encode -S | 
    aws-encryption-cli --decrypt --wrapping-keys discovery=$true --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 wrapping 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 --wrapping-keys parameter with three key attributes. You can also use multiple --wrapping-keys parameters in the same command.
To encrypt the log file, the AWS Encryption CLI asks the first wrapping key in the list, \$key1, to generate the data key that it uses to encrypt the data. Then, it uses each of the other wrapping keys to encrypt a plaintext copy of the same data key. The encrypted message in the output file includes all three of the encrypted data keys.

Bash

```bash
$ key1=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$ key2=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
$ key3=arn:aws:kms:ap-southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d

$ aws-encryption-cli --encrypt --input /logs/finance.log \
    --output /archive --suffix \ 
    --encryption-context class=log \ 
    --metadata-output ~/metadata \ 
    --wrapping-keys key=$key1 key=$key2 key=$key3
```

PowerShell

```powershell
PS C:\> $key1 = 'arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
PS C:\> $key2 = 'arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef'
PS C:\> $key3 = '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 $home\Metadata.txt `\ 
    --wrapping-keys key=$key1 key=$key2 key=$key3
```

This command decrypts the encrypted copy of the Finance.log file and writes it to a Finance.log.clear file in the Finance directory. To decrypt data encrypted under three AWS KMS keys, you can specify the same three AWS KMS keys or any subset of them. This example specifies only one of the AWS KMS keys.
To tell the AWS Encryption CLI which AWS KMS keys to use to decrypt your data, use the key attribute of the --wrapping-keys parameter. When decrypting with AWS KMS keys, the value of the key attribute must be a key ARN.

You must have permission to call the Decrypt API on the AWS KMS keys you specify. For more information, see Authentication and Access Control for AWS KMS.

As a best practice, this examples use the --max-encrypted-data-keys parameter to avoid decrypting a malformed message with an excessive number of encrypted data keys. Even though this example uses only one wrapping key for decryption, the encrypted message has three (3) encrypted data keys; one for each of the three wrapping keys used when encrypting. Specify the expected number of encrypted data keys or a reasonable maximum value, such as 5. If you specify a maximum value less than 3, the command fails. For details, see Limiting encrypted data keys.

Bash

```
$ aws-encryption-cli --decrypt --input /archive/finance.log \
   --wrapping-keys key=$key1 \
   --output /finance --suffix '.clear' \
   --metadata-output ~/metadata \
   --max-encrypted-data-keys 3 \
   --buffer \
   --encryption-context class=log
```

PowerShell

```
PS C:\> aws-encryption-cli --decrypt  \
   --input D:\Archive\Finance.log  \
   --wrapping-keys key=$key1  \
   --output D:\Finance --suffix '.clear'  \
   --metadata-output .\Metadata\Metadata.txt  \
   --max-encrypted-data-keys 3  \
   --buffer  \
   --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.

Bash

```bash
# Continue running even if an operation fails.
set +e

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

compress(){
    gzip -qf $1
}

encrypt(){
    # -e encrypt
    # -i input
    # -o output
    # --metadata-output unique file for metadata
    # -m masterKey read from environment variable
    # -c 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 ${1} ${s3bucket}/${s3folder}  
}

# Validate all required arguments are present.
if [ "${dir}" ] && [ "${encryptionContext}" ] && [ "${s3bucket}" ] &&  
[ "${s3folder}" ] && [ "${masterKey}" ]; then

# Is $dir a valid directory?
  test -d "${dir}"
  if [ $? -ne 0 ]; then
    echo "Input is not a directory; exiting"
    exit 1
  fi

# Iterate over all the files in the directory, except *gz and *encrypted (in case of  
a re-run).
  for f in $(find ${dir} -type f \( -name "*" ! -name \*.gz ! -name \*encrypted \) );
    do
      echo "Working on $f"
      compress ${f}
      encrypt ${f}.gz
      rm -f ${f}.gz
      s3put ${f}.gz.encrypted
    done;
  else
    echo "Arguments: <Directory> <encryption context> <s3://bucketname> <s3 folder>"
    echo " and ENV var \$masterKey must be set"
    exit 255
  fi

PowerShell

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

[Parameter()]  
[Switch]  
$Recurse,  

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

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

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

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

[Parameter()]  
[String]  
$EncryptionContext,  

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

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

[Parameter()]  
[String]  
$S3BucketFolder  
)

BEGIN {}  
PROCESS {

Examples
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 $EncryptDirectory -m key=$wrappingKeyID provider=$masterKeyProvider -c $EncryptionContext --metadata-output $MetadataDirectory -v) 2>&1

        # Check error status
        if ($? -eq $false)
        {
            # Write the error
            $err
        }
        elseif (Test-Path "$EncryptDirectory\$fileName.zip.encrypted")
        {
            # Step 3: Write to S3 bucket
            if ($S3BucketFolder)
            {

Examples

245
Using data key caching

This example uses data key caching 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 AWS KMS key in a keyARN 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
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

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

PowerShell

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

PS C:\> aws-encryption-cli --encrypt
--input C:\Windows\Logs --recursive
--output $home\Archive --suffix '.archive'
--wrapping-keys key=$keyARN
--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.

```powershell
PS C:\> Measure-Command {aws-encryption-cli --encrypt`
```
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 $home\Archive --suffix '.archive'
  --wrapping-keys key=$keyARN 
  --encryption-context class=log 
  --suppress-metadata 
  --caching capacity=1 max_age=10
  --max_messages_encrypted=10}
```

Days       : 0
Hours      : 0
Minutes    : 0
Seconds    : 11
Milliseconds : 813
Ticks      : 118132640
TotalDays  : 0.000136727592592593
TotalHours : 0.00328146222222222

Examples
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.

```plaintext
PS C:\> Measure-Command {aws-encryption-cli --encrypt
    --input C:\Windows\Logs --recursive
    --output $home\Archive --suffix '.archive'
    --wrapping-keys key=$keyARN
    --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 wrapping keys and other parameters, see [How to use the AWS Encryption CLI](#). For examples, see [Examples of the AWS Encryption CLI](#). 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. They represent recommended syntax in AWS Encryption CLI version 2.1.x and later.

New security features were originally released in AWS Encryption CLI versions 1.7.x and 2.0.x. However, AWS Encryption CLI version 1.8.x replaces version 1.7.x and AWS Encryption CLI 2.1.x replaces 2.0.x. For details, see the relevant security advisory in the aws-encryption-sdk-cli repository on GitHub.

⚠️ Note

Unless noted in the parameter description, each parameter or attribute can be used only once in each command.

If you use an attribute that a parameter does not support, the AWS Encryption CLI ignores that unsupported attribute without a warning or error.

Get help

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

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

Get the version

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

aws-encryption-cli --version

Encrypt data

The following syntax diagram shows the parameters that an encrypt command uses.
aws-encryption-cli --encrypt
  --input <input> [--recursive] [--decode]
  --output <output> [--interactive] [--no-overwrite] [--suffix
[<suffix>]] [--encode]
    --wrapping-keys  [--wrapping-keys] ...
      key=<keyID> [key=<keyID>] ...
      [provider=<provider-name>] [region=<aws-region>]
[profile=<aws-profile>]
    --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata]
    [--commitment-policy <commitment-policy>]
    [--encryption-context <encryption_context> [<encryption_context>
...]]
    [--max-encrypted-data-keys <integer>]
    [--algorithm <algorithm_suite>]
    [--caching <attributes>]
    [--frame-length <length>]
    [-v | -vv | -vvv | -vvvv]
    [--quiet]

Decrypt data

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

In version 1.8.x, the --wrapping-keys parameter is optional when decrypting, but recommended. Beginning in version 2.1.x, the --wrapping-keys parameter is required when encrypting and decrypting. For AWS KMS keys, you can use the key attribute to specify wrapping keys (best practice) or set the discovery attribute to true, which doesn't limit the wrapping keys that the AWS Encryption CLI can use.
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.

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, or --decrypt-unsigned parameter.

--decrypt (-d)

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

--decrypt-unsigned [Introduced in versions 1.9.x and 2.2.x]

The --decrypt-unsigned parameter decrypts ciphertext and ensures that messages are unsigned before decryption. Use this parameter if you used the --algorithm parameter and
selected an algorithm suite without digital signing to encrypt data. If the ciphertext is signed, decryption fails.

You can use `--decrypt` or `--decrypt-unsigned` for decryption but not both.

`--wrapping-keys (-w)` [Introduced in version 1.8.x]

Specifies the wrapping keys (or master keys) used in encryption and decryption operations. You can use multiple `--wrapping-keys` parameters in each command.

Beginning in version 2.1.x, the `--wrapping-keys` parameter is required in encrypt and decrypt commands. In version 1.8.x, encrypt commands require either a `--wrapping-keys` or `--master-keys` parameter. In version 1.8.x decrypt commands, a `--wrapping-keys` parameter is optional but recommended.

When using a custom master key provider, encrypt and decrypt commands require key and provider attributes. When using AWS KMS keys, encrypt commands require a key attribute. Decrypt commands require a key attribute or a discovery attribute with a value of `true` (but not both). Using the key attribute when decrypting is an AWS Encryption SDK best practice. It is particularly important if you're decrypting batches of unfamiliar messages, such as those in an Amazon S3 bucket or an Amazon SQS queue.

For an example showing how to use AWS KMS multi-Region keys as wrapping keys, see Using multi-Region AWS KMS keys.

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

**key**

Identifies the wrapping key used in the operation. The format is a key=ID pair. You can specify multiple key attributes in each `--wrapping-keys` parameter value.

- **Encrypt commands**: All encrypt commands require the key attribute. When you use an AWS KMS key 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 descriptions of the AWS KMS key identifiers, see Key identifiers in the AWS Key Management Service Developer Guide.

- **Decrypt commands**: When decrypting with AWS KMS keys, the `--wrapping-keys` parameter requires a key attribute with a key ARN value or a discovery attribute with a value of `true` (but not both). Using the key attribute is an AWS Encryption SDK best practice. When decrypting with a custom master key provider, the key attribute is required.
**Note**

To specify an AWS KMS wrapping key in a decrypt command, the value of the `key` attribute must be a key ARN. If you use a key ID, alias name, or alias ARN, the AWS Encryption CLI does not recognize the wrapping key.

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

**discovery**

Allows the AWS Encryption CLI to use any AWS KMS key to decrypt the message. The `discovery` value can be `true` or `false`. The default value is `false`. The `discovery` attribute is valid only in decrypt commands and only when the master key provider is AWS KMS.

When decrypting with AWS KMS keys, the `--wrapping-keys` parameter requires a `key` attribute or a `discovery` attribute with a value of `true` (but not both). If you use the `key` attribute, you can use a `discovery` attribute with a value of `false` to explicitly reject discovery.

- **False (default)** — When the `discovery` attribute isn't specified or its value is `false`, the AWS Encryption CLI decrypts the message using only the AWS KMS keys specified by the `key` attribute of the `--wrapping-keys` parameter. If you don't specify a `key` attribute when discovery is `false`, the decrypt command fails. This value supports an AWS Encryption CLI best practice.

- **True** — When the value of the `discovery` attribute is `true`, the AWS Encryption CLI gets the AWS KMS keys from metadata in the encrypted message, and uses those AWS KMS keys to decrypt the message. The `discovery` attribute with a value of `true` behaves like versions of the AWS Encryption CLI before version 1.8.x that didn't permit you to specify a wrapping key when decrypting. However, your intent to use any AWS KMS key is explicit. If you specify a `key` attribute when discovery is `true`, the decrypt command fails.

The `true` value might cause the AWS Encryption CLI to use AWS KMS keys in different AWS accounts and Regions, or attempt to use AWS KMS keys that the user isn't authorized to use.
When `discovery` is `true`, it's a best practice to use the `discovery-partition` and `discovery-account` attributes to limit the AWS KMS keys used to those in the AWS accounts you specify.

**discovery-account**

Limits the AWS KMS keys used for decrypting to those in the specified AWS account. The only valid value for this attribute is an [AWS account ID](https://aws.amazon.com/account/id).

This attribute is optional and valid only in decrypt commands with AWS KMS keys where the `discovery` attribute is set to `true` and the `discovery-partition` attribute is specified.

Each `discovery-account` attribute takes just one AWS account ID, but you can specify multiple `discovery-account` attributes in the same `--wrapping-keys` parameter. All accounts specified in a given `--wrapping-keys` parameter must be in the specified AWS partition.

**discovery-partition**

Specifies the AWS partition for the accounts in the `discovery-account` attribute. Its value must be an AWS partition, such as `aws`, `aws-cn`, or `aws-gov-cloud`. For information, see [Amazon Resource Names](https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html) in the *AWS General Reference*.

This attribute is required when you use the `discovery-account` attribute. You can specify only one `discovery-partition` attribute in each `--wrapping-keys` parameter. To specify AWS accounts in multiple partitions, use an additional `--wrapping-keys` parameter.

**provider**

Identifies the [master key provider](https://aws.amazon.com/kms/). 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 key. This attribute is valid only for AWS KMS keys. 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](https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-profiles.html). This attribute is valid only for AWS KMS keys. 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.

**--commitment-policy**

Specifies the [commitment policy](#) for encrypt and decrypt commands. The commitment policy determines whether your message is encrypted and decrypted with the [key commitment](#) security feature.

The **--commitment-policy** parameter is introduced in version 1.8.x. It is valid in encrypt and decrypt commands.

**In version 1.8.x**, the AWS Encryption CLI uses the *forbid-encrypt-allow-decrypt* commitment policy for all encrypt and decrypt operations. When you use the **--wrapping-keys** parameter in an encrypt or decrypt command, a **--commitment-policy** parameter with the *forbid-encrypt-allow-decrypt* value is required. If you don't use the **--wrapping-keys** parameter, the **--commitment-policy** parameter is invalid. Setting a commitment policy explicitly prevents your commitment policy from changing automatically to *require-encrypt-require-decrypt* when you upgrade to version 2.1.x

Beginning in **version 2.1.x**, all commitment policy values are supported. The **--commitment-policy** parameter is optional and the default value is *require-encrypt-require-decrypt*.

This parameter has the following values:

- *forbid-encrypt-allow-decrypt* — Cannot encrypt with key commitment. It can decrypt ciphertexts encrypted with or without key commitment.

  In version 1.8.x, this is the only valid value. The AWS Encryption CLI uses the *forbid-encrypt-allow-decrypt* commitment policy for all encrypt and decrypt operations.
AWS Encryption SDK

• **require-encrypt-allow-decrypt** — Encrypts only with key commitment. Decrypts with and without key commitment. This value is introduced in version 2.1.x.

• **require-encrypt-require-decrypt** (default) — Encrypts and decrypts only with key commitment. This value is introduced in version 2.1.x. It is the default value in versions 2.1.x and later. With this value, the AWS Encryption CLI will not decrypt any ciphertext that was encrypted with earlier versions of the AWS Encryption SDK.

For detailed information about setting your commitment policy, see [Migrating your AWS Encryption SDK](#).

**--encryption-context (-c)**

Specifies an encryption context 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"**.

**--buffer (-b) [Introducted in versions 1.9.x and 2.2.x]**

Returns plaintext only after all input is processed, including verifying the digital signature if one is present.

**--max-encrypted-data-keys [Introducted in versions 1.9.x and 2.2.x]**

Specifies the maximum number of encrypted data keys in an encrypted message. This parameter is optional.

Valid values are 1 – 65,535. If you omit this parameter, the AWS Encryption CLI does not enforce any maximum. An encrypted message can hold up to 65,535 (2^16 - 1) encrypted data keys.

You can use this parameter in encrypt commands to prevent a malformed message. You can use it in decrypt commands to detect malicious messages and avoid decrypting messages with numerous encrypted data keys that you can't decrypt. For details and an example, see [Limiting encrypted data keys](#).
--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 vs 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.

--master-keys (-m) [Deprecated]

Note

The --master-keys parameter is deprecated in 1.8.x and removed in version 2.1.x. Instead, use the --wrapping-keys parameter.

Specifies the master keys 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 (non-AWS KMS) 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 wrapping key used in the operation. The format is a key=ID pair. The key attribute is required in all encrypt commands.

When you use an AWS KMS key 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 AWS KMS key identifiers, see Key identifiers in the AWS Key Management Service Developer Guide.
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 key.

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**. 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 key. This attribute is valid only for AWS KMS keys. 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 keys. 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.

**Advanced parameters**

**--algorithm**

Specifies an alternate **algorithm suite**. This parameter is optional and valid only in encrypt commands.

If you omit this parameter, the AWS Encryption CLI uses one of the default algorithm suites for the AWS Encryption SDK introduced in version 1.8.x. Both default algorithms use AES-GCM with an **HKDF**, an ECDSA signature, and a 256-bit encryption key. One uses key commitment; one does not. The choice of default algorithm suite is determined by the **commitment policy** for the command.

The default algorithm suites are 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. This parameter is optional and valid only in encrypt commands.

Enter a value in bytes. Valid values are 0 and 1 – 2^31 - 1. A value of 0 indicates nonframed data. The default is 4096 (bytes).

Note

Whenever possible, use framed data. The AWS Encryption SDK supports nonframed data only for legacy use. Some language implementations of the AWS Encryption SDK can still generate nonframed ciphertext. All supported language implementations can decrypt framed and nonframed ciphertext.

--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.

Enter a value in bytes. If you omit this parameter, the AWS Encryption SDK does not limit the frame size when decrypting.

--caching

Enables the data key caching 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 documentation before using this feature.

The --caching parameter has the following attributes.

capacity (required)

Determines the maximum number of entries in the cache.

The minimum value is 1. There is no maximum value.

max_age (required)

Determine how long cache entries are used, in seconds, beginning when they are added to the cache.
Enter a value greater than 0. There is no maximum value.

**max_messages_encrypted (optional)**

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

Valid values are 1 – 2^32. The default value is 2^32 (messages).

**max_bytes_encrypted (optional)**

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

Valid values are 0 and 1 – 2^63 - 1. The default value is 2^63 - 1 (messages). A value of 0 lets you use data key caching only when you are encrypting empty message strings.

**Versions of the AWS Encryption CLI**

We recommend that you use the latest version of the AWS Encryption CLI.

⚠️ **Note**

Versions of the AWS Encryption CLI earlier than 4.0.0 are in the end-of-support phase. You can safely update from version 2.1.x and later to the latest version of the AWS Encryption CLI without any code or data changes. However, new security features introduced in version 2.1.x are not backward-compatible. To update from version 1.7.x or earlier, you must first update to the latest 1.x version of the AWS Encryption CLI. For details, see Migrating your AWS Encryption SDK. New security features were originally released in AWS Encryption CLI versions 1.7.x and 2.0.x. However, AWS Encryption CLI version 1.8.x replaces version 1.7.x and AWS Encryption CLI 2.1.x replaces 2.0.x. For details, see the relevant security advisory in the aws-encryption-sdk-cli repository on GitHub.

For information about significant versions of the AWS Encryption SDK, see Versions of the AWS Encryption SDK.

**Which version do I use?**

If you're new to the AWS Encryption CLI, use the latest version.
To decrypt data encrypted by a version of the AWS Encryption SDK earlier than version 1.7.x, migrate first to the latest version of the AWS Encryption CLI. Make all recommended changes before updating to version 2.1.x or later. For details, see Migrating your AWS Encryption SDK.

Learn more

- For detailed information about the changes and guidance for migrating to these new versions, see Migrating your AWS Encryption SDK.
- For descriptions of the new AWS Encryption CLI parameters and attributes, see AWS Encryption SDK CLI syntax and parameter reference.

The following lists describe the change to the AWS Encryption CLI in versions 1.8.x and 2.1.x.

**Version 1.8.x changes to the AWS Encryption CLI**

- Deprecates the --master-keys parameter. Instead, use the --wrapping-keys parameter.
- Adds the --wrapping-keys (-w) parameter. It supports all attributes of the --master-keys parameter. It also adds the following optional attributes, which are valid only when decrypting with AWS KMS keys.
  - discovery
  - discovery-partition
  - discovery-account

For custom master key providers, --encrypt and --decrypt commands require either a --wrapping-keys parameter or a --master-keys parameter (but not both). Also, an --encrypt command with AWS KMS keys requires either a --wrapping-keys parameter or a --master-keys parameter (but not both).

In a --decrypt command with AWS KMS keys, the --wrapping-keys parameter is optional, but recommended, because it is required in version 2.1.x. If you use it, you must specify either the key attribute or the discovery attribute with a value of true (but not both).

- Adds the --commitment-policy parameter. The only valid value is forbid-encrypt-allow-decrypt. The forbid-encrypt-allow-decrypt commitment policy is used in all encrypt and decrypt commands.

In version 1.8.x, when you use the --wrapping-keys parameter, a --commitment-policy parameter with the forbid-encrypt-allow-decrypt value is required. Setting the value
explicitly prevents your commitment policy from changing automatically to require-encrypt-require-decrypt when you upgrade to version 2.1.x.

### Version 2.1.x changes to the AWS Encryption CLI

- Removes the --master-keys parameter. Instead, use the --wrapping-keys parameter.
- The --wrapping-keys parameter is required in all encrypt and decrypt commands. You must specify either a key attribute or a discovery attribute with a value of true (but not both).
- The --commitment-policy parameter supports the following values. For details, see Setting your commitment policy.
  - forbid-encrypt-allow-decrypt
  - require-encrypt-allow-decrypt
  - require-encrypt-require-decrypt (Default)
- The --commitment-policy parameter is optional in version 2.1.x. The default value is require-encrypt-require-decrypt.

### Version 1.9.x and 2.2.x changes to the AWS Encryption CLI

- Adds the --decrypt-unsigned parameter. For details, see Version 2.2.x.
- Adds the --buffer parameter. For details, see Version 2.2.x.
- Adds the --max-encrypted-data-keys parameter. For details, see Limiting encrypted data keys.

### Version 3.0.x changes to the AWS Encryption CLI

- Adds support for AWS KMS multi-Region keys. For details, see Using multi-Region AWS KMS keys.
Data key caching

Data key caching stores data keys and related cryptographic material 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 and a caching cryptographic materials manager (caching CMM) that interacts with the cache and enforces security thresholds 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 to ensure that you use the minimum amount of caching required to meet your cost and performance goals.

Data key caching is not supported by the AWS Encryption SDK for .NET. Version 3.x of the AWS Encryption SDK for Java deprecates the caching CMM. However, version 4.x of the AWS Encryption SDK for .NET and version 3.x of the AWS Encryption SDK for Java support the AWS KMS Hierarchical keyring, an alternative cryptographic materials caching solution.

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.
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.

The examples in this section show how to use version 2.0.x and later of the AWS Encryption SDK. For examples that use earlier versions, find your release in the Releases list of the GitHub repository for your programming language.

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

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java
- JavaScript Browser: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

The AWS Encryption SDK for .NET does not support data key caching.

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.** In these examples, we use the local cache that the AWS Encryption SDK provides. We limit the cache to 10 data keys.

**C**

```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**

The following example uses version 2.x of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java deprecates the data key caching CMM. With version 3.x, you can also use the [AWS KMS Hierarchical keyring](https://aws.amazon.com/kms/), an alternative cryptographic materials caching solution.

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

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

**JavaScript Browser**

```javascript
const capacity = 10
const cache = getLocalCryptographicMaterialsCache(capacity)
```

**JavaScript Node.js**

```javascript
const capacity = 10
const cache = getLocalCryptographicMaterialsCache(capacity)
```

**Python**

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

cache = aws_encryption_sdk.LocalCryptoMaterialsCache(MAX_CACHE_SIZE)

- Create a **master key provider** (Java and Python) or a **keyring** (C and JavaScript). These examples use an AWS Key Management Service (AWS KMS) master key provider or a compatible **AWS KMS keyring**.

### C

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

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

### Java

The following example uses version 2.x of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java deprecates the data key caching CMM. With version 3.x, you can also use the **AWS KMS Hierarchical keyring**, an alternative cryptographic materials caching solution.

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

MasterKeyProvider<KmsMasterKey> keyProvider =
    KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn);
```

### 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 key (generatorKeyId).

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

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

/* Create an AWS KMS keyring
 * You must configure the AWS KMS keyring with at least one AWS KMS key
 * The input is the Amazon Resource Name (ARN)
 */

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

**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 key

key_provider = aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_arn])
```
• Create a caching cryptographic materials manager (caching CMM).

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

C

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”.

```c
// 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

The following example uses version 2.x of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java does not support data key caching, but it does support the AWS KMS Hierarchical keyring, an alternative cryptographic materials caching solution.

```java
/*

Using data key caching: Step-by-step
```
* Security thresholds
  * Max entry age is required.
  * Max messages (and max bytes) per entry are optional
*/
int MAX_ENTRY_AGE(SECONDS) = 60;
int MAX_ENTRY_MSGS = 10;

//Create a caching CMM
CryptoMaterialsManager cachingCmm =
    CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
    .withCache(cache)
    .withMaxAge(MAX_ENTRY_AGE(SECONDS),
    .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

/*
 * 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 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.
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_2(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

The following example uses version 2.x of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java deprecates the data key caching CMM. With version 3.x, you can also use the AWS KMS Hierarchical keyring, an alternative cryptographic materials caching solution.

```java
// When the call to encryptData specifies a caching CMM,
// the encryption operation uses the data key cache
final AwsCrypto encryptionSdk = AwsCrypto.standard();
return encryptionSdk.encryptData(cachingCmm, plaintext_source).getResult();
```
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
# Set up an encryption client
client = aws_encryption_sdk.EncryptionSDKClient()

# When the call to encrypt specifies a caching CMM,
# the encryption operation uses the data key cache
encrypted_message, header = client.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 into test code that you can run.

The example creates a local cache and a master key provider or keyring for an AWS KMS key. Then, it uses the local cache and master key provider or keyring to create a caching CMM with appropriate security thresholds. In Java and Python, the encryption request specifies the caching CMM, the plaintext data to encrypt, and an encryption context. 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 key. Be sure that you have permission to use the AWS KMS key to generate a data key.

For more detailed, real-world examples of creating and using a data key cache, see Data key caching example code.

C

```c
/*
 * 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.
 */

#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_key_arn,
    int max_entry_age,
    int cache_capacity) {  
    const uint64_t MAX_ENTRY_MSGS = 100;

    struct aws_allocator *allocator = aws_default_allocator();

    // Load error strings for debugging
    aws_cryptosdk_load_error_strings();

    // Create a keyring
```
struct aws_cryptosdk_keyring *kms_keyring =
Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_key_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(
    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_2(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

The following example uses version 2.x of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java deprecates the data key caching CMM. With version 3.x, you can also use the AWS KMS Hierarchical keyring, an alternative cryptographic materials caching solution.

// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.examples;

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.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import java.nio.charset.StandardCharsets;
import java.util.Collections;
import java.util.Map;
import java.util.concurrent.TimeUnit;

/**
 * <p>
 * Encrypts a string using an &KMS; key and data key caching
 * */

* Arguments:
  * KMS Key ARN: To find the Amazon Resource Name of your KMS key, see 'Find the key ID and ARN' at https://docs.aws.amazon.com/kms/latest/deerguide/find-cmk-id-arn.html
  * Max entry age: Maximum time (in seconds) that a cached entry can be used
  * Cache capacity: Maximum number of entries in the cache

```
public class SimpleDataKeyCachingExample {

  /*
   * Security thresholds
   * Max entry age is required.
   * Max messages (and max bytes) per data key are optional
   */
  private static final int MAX_ENTRY_MSGS = 100;

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

    // Encryption context
    // Most encrypted data should have an associated encryption context
    // to protect integrity. This sample uses placeholder values.
    // For more information see:
    final Map<String, String> encryptionContext = Collections.singletonMap("purpose", "test");

    // Create a master key provider
    MasterKeyProvider<KmsMasterKey> keyProvider = KmsMasterKeyProvider.builder()
                    .buildStrict(kmsKeyArn);

    // 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 = AwsCrypto.standard();
    return encryptionSdk.encryptData(cachingCmm, myData, encryptionContext).getResult();

JavaScript Browser

    // Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
    // SPDX-License-Identifier: Apache-2.0

    /* This is a simple example of using a caching CMM with a KMS keyring
     * to encrypt and decrypt using the AWS Encryption SDK for Javascript in a browser.
     */

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

    /* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
     * policy,
     * which enforces that this client only encrypts using committing algorithm suites
     * and enforces that this client
     * will only decrypt encrypted messages
     * that were created with a committing algorithm suite.
     * This is the default commitment policy
     * if you build the client with `buildClient()`.

Data key caching example: Encrypt a string
const { encrypt, decrypt } = buildClient(
    CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* This is injected by webpack.
 * The webpack.DefinePlugin or @aws-sdk/karma-credential-loader will replace the values when bundling.
 * The credential values are pulled from @aws-sdk/credential-provider-node
 * Use any method you like to get credentials into the browser.
 * See kms.webpack.config
 */
declare const credentials: {
    accessKeyId: string
    secretAccessKey: string
    sessionToken: string
}

/* This is done to facilitate testing. */
export async function testCachingCMMExample() {
    /* This example uses an &KMS; keyring. The generator key in a &KMS; keyring generates and encrypts the data key.
     * The caller needs kms:GenerateDataKey permission on the &KMS; key in generatorKeyId.
     */
    const generatorKeyId =

    /* Adding additional KMS keys that can decrypt.
     * The caller must have kms:Encrypt permission for every &KMS; key in keyIds.
     * You might list several keys in different AWS Regions.
     * This allows you to decrypt the data in any of the represented Regions.
     * In this example, the generator key
     * and the additional key are actually the same &KMS; key.
     * In `generatorId`, this &KMS; key is identified by its alias ARN.
     * In `keyIds`, this &KMS; key is identified by its key ARN.
     * In practice, you would specify different &KMS; keys,
     * or omit the `keyIds` parameter.
     * This is *only* to demonstrate how the &KMS; key ARNs are configured.
     */
    const keyIds = [
        'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f",
    ]
/* Need a client provider that will inject correct credentials. 
   * The credentials here are injected by webpack from your environment bundle is created
   * The credential values are pulled using @aws-sdk/credential-provider-node.
   * See kms.webpack.config
   * You should inject your credential into the browser in a secure manner
   * that works with your application.
   */
const { accessKeyId, secretAccessKey, sessionToken } = credentials

/* getClient takes a KMS client constructor
   * and optional configuration values.
   * The credentials can be injected here,
   * because browsers do not have a standard credential discovery process the way Node.js does.
   */
const clientProvider = getClient(KMS, {
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken,
  },
});

/* You must configure the KMS keyring with your KMS keys */
const keyring = new KmsKeyringBrowser({
  clientProvider,
  generatorKeyId,
  keyIds,
});

/* Create a cache to hold the data keys (and related cryptographic material).
   * This example uses the local cache provided by the Encryption SDK.
   * The `capacity` value represents the maximum number of entries
   * that the cache can hold.
   * To make room for an additional entry,
   * the cache evicts the oldest cached entry.
   * Both encrypt and decrypt requests count independently towards this threshold.
   * Entries that exceed any cache threshold are actively removed from the cache.
   * By default, the SDK checks one item in the cache every 60 seconds (60,000 milliseconds).
   * To change this frequency, pass in a `proactiveFrequency` value
   * as the second parameter. This value is in milliseconds.
   */
const capacity = 100
const cache = getLocalCryptographicMaterialsCache(capacity)

/* The partition name lets multiple caching CMMs share the same local
cryptographic cache.
* By default, the entries for each CMM are cached separately. However, if you
want these CMMs to share the cache,
* use the same partition name for both caching CMMs.
* If you don't supply a partition name, the Encryption SDK generates a random
name for each caching CMM.
* As a result, sharing elements in the cache MUST be an intentional operation.
*/
const partition = 'local partition name'

/* maxAge is the time in milliseconds that an entry will be cached.
* Elements are actively removed from the cache.
*/
const maxAge = 1000 * 60

/* The maximum number of bytes that will be encrypted under a single data key.
* This value is optional,
* but you should configure the lowest practical value.
*/
const maxBytesEncrypted = 100

/* The maximum number of messages that will be encrypted under a single data key.
* This value is optional,
* but you should configure the lowest practical value.
*/
const maxMessagesEncrypted = 10

const cachingCMM = new WebCryptoCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  partition,
  maxAge,
  maxBytesEncrypted,
  maxMessagesEncrypted,
})

/* Encryption context is a *very* powerful tool for controlling
* and managing access.
* When you pass an encryption context to the encrypt function,
* the encryption context is cryptographically bound to the ciphertext.*/
* If you don't pass in the same encryption context when decrypting,
  * the decrypt function fails.
* The encryption context is ***not*** secret!
* Encrypted data is opaque.
* You can use an encryption context to assert things about the encrypted data.
  * The encryption context helps you to determine
  * whether the ciphertext you retrieved is the ciphertext you expect to decrypt.
  * For example, if you are only expecting data from 'us-west-2',
  * the appearance of a different AWS Region in the encryption context can indicate malicious interference.
* See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/concepts.html#encryption-context

* Also, cached data keys are reused ***only*** when the encryption contexts passed into the functions are an exact case-sensitive match.
* See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-caching-details.html#caching-encryption-context

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

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

const { result } = await encrypt(cachingCMM, plainText, { encryptionContext })
```

/* Find data to encrypt. */
const plainText = new Uint8Array([1, 2, 3, 4, 5])

/* Encrypt the data.  
* The caching CMM only reuses data keys
* when it know the length (or an estimate) of the plaintext.
* However, in the browser, 
* you must provide all of the plaintext to the encrypt function.
* Therefore, the encrypt function in the browser knows the length of the plaintext
* and does not accept a plaintextLength option.
*/
const { result } = await encrypt(cachingCMM, plainText, { encryptionContext })

/* Log the plain text 
* only for testing and to show that it works. 
*/
console.log('plainText:', plainText)
document.write('</br>plainText:' + plainText + '</br>')
/* Log the base64-encoded result
 * so that you can try decrypting it with another AWS Encryption SDK implementation.
 */
const resultBase64 = toBase64(result)
console.log(resultBase64)
document.write(resultBase64)

/* Decrypt the data.
 * NOTE: This decrypt request will not use the data key
 * that was cached during the encrypt operation.
 * Data keys for encrypt and decrypt operations are cached separately.
 */
const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

/* Grab the encryption context so you can verify it. */
const { encryptionContext: decryptedContext } = messageHeader

/* Verify the encryption context.
 * If you use an algorithm suite with signing,
 * the Encryption SDK adds a name-value pair to the encryption context that contains the public key.
 * Because the encryption context might contain additional key-value pairs,
 * do not include a test that requires that all key-value pairs match.
 * Instead, verify that the key-value pairs that you supplied to the `encrypt` function are included in the encryption context that the `decrypt` function returns.
 */
Object.entries(encryptionContext).forEach(([key, value]) => {
  if (decryptedContext[key] !== value)
    throw new Error('Encryption Context does not match expected values')
})

/* Log the clear message
 * only for testing and to show that it works.
 */
document.write('</br>Decrypted:' + plaintext)
console.log(plaintext)

/* Return the values to make testing easy. */
return { plainText, plaintext }
}
import { KmsKeyringNode, buildClient, CommitmentPolicy, NodeCachingMaterialsManager, getLocalCryptographicMaterialsCache, } from '@aws-crypto/client-node

/* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment policy,
 * which enforces that this client only encrypts using committing algorithm suites
 * and enforces that this client
 * will only decrypt encrypted messages
 * that were created with a committing algorithm suite.
 * This is the default commitment policy
 * if you build the client with `buildClient()`.
 */

const { encrypt, decrypt } = buildClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

export async function cachingCMMNodeSimpleTest() {
  /* An &KMS; key is required to generate the data key.
   * You need kms:GenerateDataKey permission on the &KMS; key in generatorKeyId.
   */

  const generatorKeyId =

  /* Adding alternate &KMS; keys that can decrypt.
   * Access to kms:Encrypt is required for every &KMS; key in keyIds.
   * You might list several keys in different AWS Regions.
   * This allows you to decrypt the data in any of the represented Regions.
   * In this example, the generator key
   * and the additional key are actually the same &KMS; key.
   * In `generatorId`, this &KMS; key is identified by its alias ARN.
   * In `keyIds`, this &KMS; key is identified by its key ARN.
   * In practice, you would specify different &KMS; keys,
   * or omit the `keyIds` parameter.
   * This is *only* to demonstrate how the &KMS; key ARNs are configured.
   */
const keyIds = [
  'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f',
]

/* The KMS keyring must be configured with the desired KMS keys
 * This example passes the keyring to the caching CMM
 * instead of using it directly.
 */
const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })

/* Create a cache to hold the data keys (and related cryptographic material).
 * This example uses the local cache provided by the Encryption SDK.
 * The `capacity` value represents the maximum number of entries
 * that the cache can hold.
 * To make room for an additional entry,
 * the cache evicts the oldest cached entry.
 * Both encrypt and decrypt requests count independently towards this threshold.
 * Entries that exceed any cache threshold are actively removed from the cache.
 * By default, the SDK checks one item in the cache every 60 seconds (60,000 milliseconds).
 * To change this frequency, pass in a `proactiveFrequency` value
 * as the second parameter. This value is in milliseconds.
 */
const capacity = 100
const cache = getLocalCryptographicMaterialsCache(capacity)

/* The partition name lets multiple caching CMMs share the same local cryptographic cache.
 * By default, the entries for each CMM are cached separately. However, if you want these CMMs to share the cache,
 * use the same partition name for both caching CMMs.
 * If you don't supply a partition name, the Encryption SDK generates a random name for each caching CMM.
 * As a result, sharing elements in the cache MUST be an intentional operation.
 */
const partition = 'local partition name'

/* maxAge is the time in milliseconds that an entry will be cached.
 * Elements are actively removed from the cache.
 */
const maxAge = 1000 * 60

/* The maximum amount of bytes that will be encrypted under a single data key. */
* This value is optional, * but you should configure the lowest value possible. */
const maxBytesEncrypted = 100

/* The maximum number of messages that will be encrypted under a single data key. * This value is optional, * but you should configure the lowest value possible. */
const maxMessagesEncrypted = 10

const cachingCMM = new NodeCachingMaterialsManager({
    backingMaterials: keyring,
    cache,
    partition,
    maxAge,
    maxBytesEncrypted,
    maxMessagesEncrypted,
})

/* Encryption context is a *very* powerful tool for controlling * and managing access. * When you pass an encryption context to the encrypt function, * the encryption context is cryptographically bound to the ciphertext. * If you don’t pass in the same encryption context when decrypting, * the decrypt function fails. * The encryption context is ***not*** secret! * Encrypted data is opaque. * You can use an encryption context to assert things about the encrypted data. * The encryption context helps you to determine * whether the ciphertext you retrieved is the ciphertext you expect to decrypt. * For example, if you are are only expecting data from 'us-west-2', * the appearance of a different AWS Region in the encryption context can indicate malicious interference. * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/concepts.html#encryption-context
 *
* Also, cached data keys are reused ***only*** when the encryption contexts passed into the functions are an exact case-sensitive match. * 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',
}

/* Find data to encrypt. A simple string. */
const cleartext = 'asdf'

/* Encrypt the data. 
 * The caching CMM only reuses data keys 
 * when it know the length (or an estimate) of the plaintext. 
 * If you do not know the length, 
 * because the data is a stream 
 * provide an estimate of the largest expected value. 
 * 
 * If your estimate is smaller than the actual plaintext length 
 * the AWS Encryption SDK will throw an exception. 
 * 
 * If the plaintext is not a stream, 
 * the AWS Encryption SDK uses the actual plaintext length 
 * instead of any length you provide. 
 */
const { result } = await encrypt(cachingCMM, cleartext,
    encryptionContext,
    plaintextLength: 4,
)

/* Decrypt the data. 
 * NOTE: This decrypt request will not use the data key 
 * that was cached during the encrypt operation. 
 * Data keys for encrypt and decrypt operations are cached separately. 
 */
const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

/* Grab the encryption context so you can verify it. */
const { encryptionContext: decryptedContext } = messageHeader

/* Verify the encryption context. 
 * If you use an algorithm suite with signing, 
 * the Encryption SDK adds a name-value pair to the encryption context that 
 * contains the public key. 
 * Because the encryption context might contain additional key-value pairs, 
 * do not include a test that requires that all key-value pairs match. 

* Instead, verify that the key-value pairs that you supplied to the `encrypt` function are included in the encryption context that the `decrypt` function returns.

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

/* Return the values so the code can be tested. */
return { plaintext, result, cleartext, messageHeader }
}
```

---

**Python**

```python
# Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License"). You
# may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# or in the "license" file accompanying this file. This file is
# distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF
# ANY KIND, either express or implied. See the License for the specific
# language governing permissions and limitations under the License.

"""Example of encryption with data key caching.""
import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy

def encrypt_with_caching(kms_key_arn, max_age_in_cache, cache_capacity):
    """Encrypts a string using an &KMS; key and data key caching.

    :param str kms_key_arn: Amazon Resource Name (ARN) of the &KMS; 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"
```
Setting cache security thresholds

When you implement data key caching, you need to configure the security thresholds that the caching CMM 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 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.

All language implementations of the AWS Encryption SDK define the maximum age in seconds, except for the AWS Encryption SDK for JavaScript, which uses milliseconds.

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](#)
- [Creating a cryptographic materials cache](#)
- [Creating a caching cryptographic materials manager](#)
- [What is in a data key cache entry?](#)
- [Encryption context: How to select cache entries](#)
- [Is my application using cached data keys?](#)

**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. 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 (caching CMM). The caching CMM is a specialized cryptographic materials manager (CMM) that interacts with a cache and an underlying CMM. (When you specify a master key provider 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**
- **Encryption context** (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 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 (C and JavaScript) or master key provider (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, which it returns to the user.

Encrypted 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) 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, 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, 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 `getLocalCryptoMaterialsCache` 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 (caching CMM). The caching CMM gets data keys from the cache, puts data keys in the cache, and enforces security thresholds 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 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), or master key provider or keyring.

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, 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 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
- Message signing key (if one is used)
- Algorithm suite
- 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 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 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** 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** with a **local cache** 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. 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 and Python. The sample also includes an 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>Requests per second per client</th>
<th>Clients per region</th>
<th>Average requests per second per region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate data key (us-west-2)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Encrypt data key (eu-central-1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total (per region)</td>
<td>1</td>
<td>500</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No cache</td>
<td>500</td>
</tr>
<tr>
<td>Local cache</td>
<td>5</td>
</tr>
</tbody>
</table>

### Consumer requests

<table>
<thead>
<tr>
<th>Requests per second per client</th>
<th>Client per region</th>
<th>Average requests per second per region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrypt data key</td>
<td>500</td>
<td>1,000</td>
</tr>
<tr>
<td>Producers</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No cache</td>
<td>2</td>
</tr>
<tr>
<td>Local cache</td>
<td>10</td>
</tr>
</tbody>
</table>

Data key caching example code

This code sample creates a simple implementation of data key caching with a local cache in Java and Python. 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 details about the implementation of data key caching in each language, see the Javadoc and Python documentation for the AWS Encryption SDK.

Data key caching is available for all programming languages that the AWS Encryption SDK supports.

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

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java
- JavaScript Browser: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

**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 (caching CMM) and associates it with a local cache and an underlying AWS KMS master key provider. The caching CMM caches the data keys (and related cryptographic materials) 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) or master key provider, the encryption will use data key caching.

**Java**

The following example uses version 2.x of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java deprecates the data key caching CMM. With version 3.x, you can also use the AWS KMS Hierarchical keyring, an alternative cryptographic materials caching solution.

```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
 */
```

Example code
import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import com.amazonaws.encryptionsdk.multi.MultipleProviderFactory;
import com.amazonaws.util.json.Jackson;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kms.KmsClient;

/**
 * Pushes data to Kinesis Streams in multiple Regions.
 */
public class MultiRegionRecordPusher {

    private static final long MAX_ENTRY_AGE_MILLISECONDS = 300000;
    private static final long MAX_ENTRYUSES = 100;
    private static final int MAX_CACHE_ENTRIES = 100;
    private final String streamName_;
    private final ArrayList<KinesisClient> kinesisClients_;
private final CachingCryptoMaterialsManager cachingMaterialsManager_;  
private final 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_ = AwsCrypto.builder()  
.withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)  
.build();  
kinesisClients_ = new ArrayList<>();  

AwsCredentialsProvider credentialsProvider =  
DefaultCredentialsProvider.builder().build();  

// Build KmsMasterKey and AmazonKinesisClient objects for each target region  
List<KmsMasterKey> masterKeys = new ArrayList<>();  
for (Region region : regions) {  
kinesisClients_.add(KinesisClient.builder()  
.withCredentialsProvider(credentialsProvider)  
.region(region)  
.build());  

KmsMasterKey regionMasterKey = KmsMasterKeyProvider.builder()  
.defaultRegion(region)  
.builderSupplier(() ->  
KmsClient.builder().credentialsProvider(credentialsProvider))  
.buildStrict(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<>();
    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 (KinesisClient regionalKinesisClient : kinesisClients_) {
        regionalKinesisClient.putRecord(builder ->
            builder.streamName(streamName_)
                .data(SdkBytes.fromByteArray(encryptedData))
                .partitionKey(partitionKey);
        }
    }
}

Python

"""
Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
import json
import uuid

from aws_encryption_sdk import EncryptionSDKClient, StrictAwsKmsMasterKeyProvider,
CachingCryptoMaterialsManager, LocalCryptoMaterialsCache, CommitmentPolicy
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 EncryptionSDKClient
        _client = EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

        # Set up KMSMasterKeyProvider with cache
        _key_provider = StrictAwsKmsMasterKeyProvider(kms_alias_name)

        # 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,
Example code

```python
client = boto3.client('kms', region_name=region),
key_id=kms_alias_name
)
_key_provider.add_master_key_provider(regional_master_key)

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

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

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

encryption_context = {'stream': self._stream_name}

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

# Encrypt data
encrypted_data, _header = _client.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 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 decrypt method.

The Java code builds a master key provider in **strict mode** with a specified AWS KMS key. Strict mode isn't required when decrypting, but it's a **best practice**. The Python code uses **discovery mode**, which lets the AWS Encryption SDK use any wrapping key that encrypted a data key to decrypt it.

Java

The following example uses version 2.x of the AWS Encryption SDK for Java. Version 3.x of the AWS Encryption SDK for Java deprecates the data key caching CMM. With version 3.x, you can also use the **AWS KMS Hierarchical keyring**, an alternative cryptographic materials caching solution.

This code creates a master key provider for decrypting in strict mode. The AWS Encryption SDK can use only the AWS KMS keys you specify to decrypt your message.

```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.
 */
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
```
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRecord;
import com.amazonaws.util.BinaryUtils;
import java.io.UnsupportedEncodingException;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;

/**
 * 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 final CachingCryptoMaterialsManager cachingMaterialsManager_;
    private final AwsCrypto crypto_;
    private final DynamoDbTable<Item> table_;

    /**
     * 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.
     */
    public LambdaDecryptAndWrite() {
        String kmsKeyArn = System.getenv("CMK_ARN");
        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()
            .withMasterKeyProvider(KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn))
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .build();

        crypto_ = AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
String tableName = System.getenv("TABLE_NAME");
DynamoDbEnhancedClient dynamodb = DynamoDbEnhancedClient.builder().build();
table_ = dynamodb.table(tableName, TableSchema.fromClass(Item.class));
}

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

    // Decrypt and unpack record
    CryptoResult<byte[], ?> plaintextResult =
    crypto_.decryptData(cachingMaterialsManager_,
                        ciphertext);

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

    // Write record to DynamoDB
    String jsonItem = new String(plaintextResult.getResult(),
                                 StandardCharsets.UTF_8);
    System.out.println(jsonItem);
    table_.putItem(Item.fromJSON(jsonItem));
}

private static class Item {
    static Item fromJSON(String jsonText) {
        // Parse JSON and create new Item
        return new Item();
    }
}
Python

This Python code decrypts with a master key provider in discovery mode. It lets the AWS Encryption SDK use any wrapping key that encrypted a data key to decrypt it. Strict mode, in which you specify the wrapping keys that can be used for decryption, is a best practice.

```python
import base64
import json
import logging
import os
from aws_encryption_sdk import EncryptionSDKClient,
    DiscoveryAwsKmsMasterKeyProvider, CachingCryptoMaterialsManager,
    LocalCryptoMaterialsCache, CommitmentPolicy
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 encryption_sdk_client
```
Example code

```python
encryption_sdk_client = EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

global materials_manager
key_provider = DiscoveryAwsKmsMasterKeyProvider()
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 = encryption_sdk_client.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!')
```

Example code
Data key caching example: AWS CloudFormation template

This AWS CloudFormation template sets up all the necessary AWS resources to reproduce the data key caching example.

JSON

```
{
    "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 key 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": {
"Type": "String",
"Description": "Name to use for Kinesis Stream"
}
"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": {
"Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${PythonLambdaOutputTable}"
}
},
{
"Effect": "Allow",
"Action": [
"dynamodb:PutItem"
],
"Resource": {
"Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${PythonLambdaOutputTable}"
}
},
{
"Effect": "Allow",
"Action": [
"kinesis:GetRecords",
"kinesis:GetShardIterator",
"kinesis:DescribeStream",
"kinesis:ListStreams"
]
],
  "Resource": {
    "Fn::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": {
      "Fn::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": {
      "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:${AWS::AccountId}:stream/${InputStream}"
    },
    "FunctionName": {
      "Ref": "PythonLambdaFunction"
    },
    "StartingPosition": "TRIM_HORIZON"
  }
},
"JavaLambdaOutputTable": {
  "Type": "AWS::DynamoDB::Table",
  "Properties": {
    "AttributeDefinitions": [
      {
        "AttributeName": "id",
        "AttributeType": "S"
      }
    ],
    "KeySchema": [
      {
        "AttributeName": "id",
        "KeyType": "HASH"
      }
    ],
    "ProvisionedThroughput": {
      "ReadCapacityUnits": 1,
      "WriteCapacityUnits": 1
    }
  }
},
"JavaLambdaRole": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "lambda.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
},
"JavaSecurityGroup": {
  "Type": "AWS::EC2::SecurityGroup",
  "Properties": {
    "SecurityGroupIngress": [
      {
        "IpProtocol": "tcp",
        "FromPort": 8080,
        "ToPort": 8080,
        "CidrIp": "0.0.0.0/0"
      }
    ],
    "GroupName": "SampleSecurityGroup",
    "GroupDescription": "Sample security group for sample application"
  }
}
"Statement": [
    {
        "Effect": "Allow",
        "Principal": {
            "Service": "lambda.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }
],
"ManagedPolicyArns": [
    "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
],
"Policies": [
    {
        "PolicyName": "JavaLambdaAccess",
        "PolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Action": [
                        "dynamodb:DescribeTable",
                        "dynamodb:BatchWriteItem"
                    ],
                    "Resource": {
                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${JavaLambdaOutputTable}"
                    }
                },
                {
                    "Effect": "Allow",
                    "Action": ["dynamodb:PutItem"],
                    "Resource": {
                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${JavaLambdaOutputTable}"*
                    }
                },
                {
                    "Effect": "Allow",
                    "Action": ["dynamodb:GetItem"]
                }
            ]
        }
    }
]
"kinesis:GetRecords",
"kinesis:GetShardIterator",
"kinesis:DescribeStream",
"kinesis:ListStreams"
],
"Resource": {
  "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:${AWS::AccountId}:stream/${InputStream}"
}
}],
"JavaLambdaFunction": {
  "Type": "AWS::Lambda::Function",
  "Properties": {
    "Description": "Java consumer",
    "Runtime": "java8",
    "MemorySize": 512,
    "Timeout": 90,
    "Role": {
      "Fn::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": {
  "Fn::GetAtt": [
    "RegionKinesisCMK",
    "Arn"
  ]
}
}
,
"JavaLambdaSourceMapping": {
  "Type": "AWS::Lambda::EventSourceMapping",
  "Properties": {
    "BatchSize": 1,
    "Enabled": true,
    "EventSourceArn": {
      "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:${AWS::AccountId}:stream/${InputStream}"}
  },
  "FunctionName": {
    "Ref": "JavaLambdaFunction"
  },
  "StartingPosition": "TRIM_HORIZON"
}
},
"RegionKinesisCMK": {
  "Type": "AWS::KMS::Key",
  "Properties": {
    "Description": "Used to encrypt data passing through Kinesis Stream in this region",
    "Enabled": true,
    "KeyPolicy": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "AWS": {
              "Fn::Sub": "arn:aws:iam:${AWS::AccountId}:root"
            }
          }
        }
      ]
    }
  }
}
"Action": [
    "kms:Encrypt",
    "kms:GenerateDataKey",
    "kms:CreateAlias",
    "kms:DeleteAlias",
    "kms:DescribeKey",
    "kms:DisableKey",
    "kms:EnableKey",
    "kms:PutKeyPolicy",
    "kms:ScheduleKeyDeletion",
    "kms:UpdateAlias",
    "kms:UpdateKeyDescription"
],
"Resource": "*"
},
{
    "Effect": "Allow",
    "Principal": {
        "AWS": [
            {"Fn::GetAtt": [
                "PythonLambdaRole",
                "Arn"
            ],
            {"Fn::GetAtt": [
                "JavaLambdaRole",
                "Arn"
            ]}
        ]
    },
    "Action": "kms:Decrypt",
    "Resource": "*"
}
},
"RegionKinesisCMKAlias": {
    "Type": "AWS::KMS::Alias",
    "Properties": {
        "AliasName": {
            "Fn::GetAtt": [
                "PythonLambdaRole",
                "Arn"
            ]
        }
    }
}
"Fn::Sub": "alias/${KeyAliasSuffix}"
],
"TargetKeyId": {
  "Ref": "RegionKinesisCMK"
}
}
}
}

**YAML**

```yaml
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:${AWS::Region}:arn:aws:lambda:${AWS::Account}:role/${PythonLambdaOutputTable}
            - Effect: Allow
              Action:
                - dynamodb:PutItem
Resource: !Sub arn:aws:dynamodb:${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
Versions of the AWS Encryption SDK

The AWS Encryption SDK language implementations use semantic versioning to make it easier for you to identify the magnitude of changes in each release. A change in the major version number, such as 1.x.x to 2.x.x, indicates a breaking change that is likely to require code changes and a planned deployment. Breaking changes in a new version might not impact every use case, review the release notes to see if you're impacted. A change in a minor version, such as x.1.x to x.2.x, is always backward compatible, but might include deprecated elements.

Whenever possible, use the latest version of the AWS Encryption SDK in your chosen programming language. The maintenance and support policy for each version differs between programming language implementations. For details about the supported versions in your preferred programming language, see the SUPPORT_POLICY.rst file in its GitHub repository.

When upgrades include new features that require special configuration to avoid encryption or decryption error, we provide an intermediate version and detailed instructions for using it. For example, versions 1.7.x and 1.8.x are designed to be transitional versions that help you upgrade from versions earlier than 1.7.x to versions 2.0.x and later. For details, see Migrating your AWS Encryption SDK.

Note

The x in a version number represents any patch of the major and minor version. For example, version 1.7.x represents all versions that begin with 1.7, including 1.7.1 and 1.7.9. New security features were originally released in AWS Encryption CLI versions 1.7.x and 2.0.x. However, AWS Encryption CLI version 1.8.x replaces version 1.7.x and AWS Encryption CLI 2.1.x replaces 2.0.x. For details, see the relevant security advisory in the aws-encryption-sdk-cli repository on GitHub.

The following tables provide an overview of the major differences between supported versions of the AWS Encryption SDK for each programming language.

C

For a detailed description of all changes, see the CHANGELOG.md in the aws-encryption-sdk-c repository on GitHub.
### Major version Details SDK major version life-cycle phase

1.x

- 1.0 Initial release.
- 1.7 Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see version 1.7.x.

2.x

- 2.0 Updates to the AWS Encryption SDK. For more information, see version 2.0.x.
- 2.2 Improvements to the message decryption process.
- 2.3 Adds support for AWS KMS multi-Region keys.

**C# / .NET**

For a detailed description of all changes, see the CHANGELOG.md in the aws-encryption-sdk-net repository on GitHub.

<table>
<thead>
<tr>
<th>Major version</th>
<th>Details</th>
<th>SDK major version life-cycle phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.x</td>
<td>3.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
Version 3.x of the AWS Encryption SDK for .NET will enter maintenance mode on May, 13 2024.

4.x 4.0 Adds support for the AWS KMS Hierarchical keyring, the required encryption context CMM, and asymmetric RSA AWS KMS keyrings.

**Command line interface (CLI)**

For a detailed description of all changes, see Versions of the AWS Encryption CLI and the CHANGELOG.rst in the aws-encryption-sdk-cli repository on GitHub.

<table>
<thead>
<tr>
<th>Major version</th>
<th>Details</th>
<th>SDK major version life-cycle phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>1.0 Initial release.</td>
<td>End-of-Support phase</td>
</tr>
<tr>
<td></td>
<td>1.7 Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see version 1.7.x.</td>
<td></td>
</tr>
<tr>
<td>2.x</td>
<td>2.0 Updates to the AWS Encryption SDK. For more information, see version 2.0.x.</td>
<td>End-of-Support phase</td>
</tr>
</tbody>
</table>
## 2.1
Removes the `--discovery` parameter and replaces it with the `discovery` attribute of the `--wrapping-keys` parameter.

Version 2.1.0 of the AWS Encryption CLI is equivalent to version 2.0 in other programming languages.

## 2.2
Improvements to the message decryption process.

## 3.0
3.0
Adds support for AWS KMS multi-Region keys. **End-of-Support phase**

## 4.0
4.0
The AWS Encryption CLI no longer supports Python 2 or Python 3.4. As of major version 4.x of the AWS Encryption CLI, only Python 3.5 or later is supported. **General Availability (GA)**
### 4.1

The AWS Encryption CLI no longer supports Python 3.5. As of version 4.1.x of the AWS Encryption CLI, only Python 3.6 or later is supported.

### 4.2

The AWS Encryption CLI no longer supports Python 3.6. As of version 4.2.x of the AWS Encryption CLI, only Python 3.7 or later is supported.

### Java

For a detailed description of all changes, see the [CHANGELOG.rst](https://aws-encryption-sdk-java.github.io/aws-encryption-sdk-java) in the [aws-encryption-sdk-java](https://github.com/aws-encryption-sdk-java/aws-encryption-sdk-java) repository on GitHub.

<table>
<thead>
<tr>
<th>Major version</th>
<th>Details</th>
<th>SDK major version life-cycle phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>1.0</td>
<td>Initial release.</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>Adds support for cryptographic materials manager and data key caching. Moved to deterministic IV generation.</td>
</tr>
<tr>
<td></td>
<td>1.6.1</td>
<td>Deprecates AwsCrypto encryptString() and</td>
</tr>
</tbody>
</table>
1.7 Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see version 1.7.x.

2.0 Updates to the AWS Encryption SDK. For more information, see version 2.0.x.

2.2 Improvements to the message decryption process.

2.3 Adds support for AWS KMS multi-Region keys.

2.4 Adds support for AWS SDK for Java 2.x.
Integrates the AWS Encryption SDK for Java with the material providers library.

Adds support for symmetric and asymmetric RSA AWS KMS keyrings, AWS KMS Hierarchical keyrings, Raw AES keyrings, Raw RSA keyrings, Multi-keyrings, and the required encryption context CMM.

JavaScript

For a detailed description of all changes, see the CHANGELOG.md in the aws-encryption-sdk-javascript repository on GitHub.

<table>
<thead>
<tr>
<th>Major version</th>
<th>Details</th>
<th>SDK major version life-cycle phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>1.0</td>
<td>Initial release.</td>
</tr>
<tr>
<td></td>
<td>1.7</td>
<td>Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see version 1.7.x.</td>
</tr>
</tbody>
</table>
2.x 2.0  Updates to the AWS Encryption SDK. For more information, see version 2.0.x.

2.2  Improvements to the message decryption process.

2.3  Adds support for AWS KMS multi-Region keys.

3.x 3.0  Removes CI coverage for Node 10. Upgrades dependencies to no longer support Node 8 and Node 10.

4.x 4.0  Requires version 3 of the AWS Encryption SDK for JavaScript's kms-client to use the AWS KMS keyring.

Python

For a detailed description of all changes, see the CHANGELOG.rst in the aws-encryption-sdk-python repository on GitHub.

<table>
<thead>
<tr>
<th>Major version</th>
<th>Details</th>
<th>SDK major version life-cycle phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>1.0</td>
<td>End-of-Support phase</td>
</tr>
<tr>
<td>2.x</td>
<td>2.0</td>
<td>End-of-Support phase</td>
</tr>
<tr>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.x</td>
<td>3.0</td>
<td>Maintenance</td>
</tr>
<tr>
<td>4.x</td>
<td>4.0</td>
<td>General Availability (GA)</td>
</tr>
</tbody>
</table>

Python
<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3</td>
<td>Adds support for cryptographic materials manager and data key caching. Moved to deterministic IV generation.</td>
</tr>
<tr>
<td>1.7</td>
<td>Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see version 1.7.x.</td>
</tr>
<tr>
<td>2.x</td>
<td>Updates to the AWS Encryption SDK. For more information, see version 2.0.x.</td>
</tr>
<tr>
<td>2.0</td>
<td>End-of-Support phase</td>
</tr>
<tr>
<td>2.2</td>
<td>Improvements to the message decryption process.</td>
</tr>
<tr>
<td>2.3</td>
<td>Adds support for AWS KMS multi-Region keys.</td>
</tr>
</tbody>
</table>
The AWS Encryption SDK for Python no longer supports Python 2 or Python 3.4. As of major version 3.x of the AWS Encryption SDK for Python, only Python 3.5 or later is supported.

Version details

The following list describes the major differences between supported versions of the AWS Encryption SDK.

Topics

• Versions earlier than 1.7.x
• Version 1.7.x
• Version 2.0.x
• Version 2.2.x
• Version 2.3.x

Versions earlier than 1.7.x

Note

All 1.x.x versions of the AWS Encryption SDK are in the end-of-support phase. Upgrade to the latest available version of the AWS Encryption SDK for your programming language as soon as is practical. To upgrade from an AWS Encryption SDK version earlier than 1.7.x, you must first upgrade to 1.7.x. For details, see Migrating your AWS Encryption SDK.

Versions of the AWS Encryption SDK earlier than 1.7.x provide important security features, including encryption with the Advanced Encryption Standard algorithm in Galois/Counter Mode...
(AES-GCM), an HMAC-based extract-and-expand key derivation function (HKDF), signing, and a 256-bit encryption key. However, these versions don't support best practices that we recommend, including key commitment.

Version 1.7.x

Note

All 1.x.x versions of the AWS Encryption SDK are in the end-of-support phase.

Version 1.7.x is designed to help users of earlier versions of the AWS Encryption SDK to upgrade to versions 2.0.x and later. If you are new to the AWS Encryption SDK, you can skip this version and begin with the latest available version in your programming language.

Version 1.7.x is fully backward compatible; it does not introduce any breaking changes or change the behavior of the AWS Encryption SDK. It's also forwards compatible; it allows you to update your code so it's compatible with version 2.0.x. It includes new features, but doesn't fully enable them. And it requires configuration values that prevent you from immediately adopting all new features until you are ready.

Version 1.7.x includes the following changes:

AWS KMS master key provider updates (required)

Version 1.7.x introduces new constructors to the AWS Encryption SDK for Java and AWS Encryption SDK for Python that explicitly create AWS KMS master key providers in either strict or discovery mode. This version adds similar changes to the AWS Encryption SDK command-line interface (CLI). For details, see Updating AWS KMS master key providers.

- In strict mode, AWS KMS master key providers require a list of wrapping keys, and they encrypt and decrypt with only the wrapping keys you specify. This is an AWS Encryption SDK best practice that assures that you are using the wrapping keys you intend to use.
- In discovery mode, AWS KMS master key providers do not take any wrapping keys. You cannot use them for encrypting. When decrypting, they can use any wrapping key to decrypt an encrypted data key. However, you can limit the wrapping keys used for decryption to those in particular AWS accounts. Account filtering is optional, but it's a best practice that we recommend.
The constructors that create earlier versions of AWS KMS master key providers are deprecated in version 1.7.x and removed in version 2.0.x. These constructors instantiate master key providers that encrypt using the wrapping keys you specify. However, they decrypt encrypted data keys using the wrapping key that encrypted them, without regard to the specified wrapping keys. Users might unintentionally decrypt messages with wrapping keys they don't intend to use, including AWS KMS keys in other AWS accounts and Regions.

There are no changes to constructors for AWS KMS master keys. When encrypting and decrypting, AWS KMS master keys use only the AWS KMS key that you specify.

AWS KMS keyring updates (optional)

Version 1.7.x adds a new filter to the AWS Encryption SDK for C and AWS Encryption SDK for JavaScript implementations that limits AWS KMS discovery keyrings to particular AWS accounts. This new account filter is optional, but it's a best practice that we recommend. For details, see Updating AWS KMS keyrings.

There are no changes to constructors for AWS KMS keyrings. Standard AWS KMS keyrings behave like master key providers in strict mode. AWS KMS discovery keyrings are created explicitly in discovery mode.

Passing a key ID to AWS KMS Decrypt

Beginning in version 1.7.x, when decrypting encrypted data keys, the AWS Encryption SDK always specifies an AWS KMS key in its calls to the AWS KMS Decrypt operation. The AWS Encryption SDK gets the key ID value for the AWS KMS key from the metadata in each encrypted data key. This feature doesn't require any code changes.

Specifying the key ID of the AWS KMS key is not required to decrypt ciphertext that was encrypted under a symmetric encryption KMS key, but it is an AWS KMS best practice. Like specifying wrapping keys in your key provider, this practice assures that AWS KMS only decrypts using the wrapping key you intend to use.

Decrypt ciphertext with key commitment

Version 1.7.x can decrypt ciphertext that was encrypted with or without key commitment. However, it cannot encrypt ciphertext with key commitment. This property allows you to fully deploy applications that can decrypt ciphertext encrypted with key commitment before they ever encounter any such ciphertext. Because this version decrypts messages that are encrypted without key commitment, you don't need to re-encrypt any ciphertext.
To implement this behavior, version 1.7.x includes a new commitment policy configuration setting that determines whether the AWS Encryption SDK can encrypt or decrypt with key commitment. In version 1.7.x, the only valid value for the commitment policy, ForbiddenEncryptAllowDecrypt, is used in all encrypt and decrypt operations. This value prevents the AWS Encryption SDK from encrypting with either of the new algorithm suites that include key commitment. It allows the AWS Encryption SDK to decrypt ciphertext with and without key commitment.

Although there is only one valid commitment policy value in version 1.7.x, we require that you can set this value explicitly when you use the new APIs introduced in this release. Setting the value explicitly prevents your commitment policy from changing automatically to require-encrypt-require-decrypt when you upgrade to version 2.1.x. Instead, you can migrate your commitment policy in stages.

Algorithm suites with key commitment

Version 1.7.x includes two new algorithm suites that support key commitment. One includes signing; the other does not. Like earlier supported algorithm suites, both of these new algorithm suites include encryption with AES-GCM, a 256-bit encryption key, and an HMAC-based extract-and-expand key derivation function (HKDF).

However, the default algorithm suite used for encryption does not change. These algorithm suites are added to version 1.7.x to prepare your application to use them in versions 2.0.x and later.

CMM implementation changes

Version 1.7.x introduces changes to the Default cryptographic materials manager (CMM) interface to support key commitment. This change affects you only if you have written a custom CMM. For details, see the API documentation or GitHub repository for your programming language.

Version 2.0.x

Version 2.0.x supports new security features offered in the AWS Encryption SDK, including specified wrapping keys and key commitment. To support these features, version 2.0.x includes breaking changes for earlier versions of the AWS Encryption SDK. You can prepare for these changes by deploying version 1.7.x. Version 2.0.x includes all of the new features introduced in version 1.7.x with the following additions and changes.
Note

Version 2.x.x of the AWS Encryption SDK for Python, AWS Encryption SDK for JavaScript, and the AWS Encryption CLI are in the end-of-support phase. For information about support and maintenance of this AWS Encryption SDK version in your preferred programming language, see the SUPPORT_POLICY.rst file in its GitHub repository.

AWS KMS master key providers

The original AWS KMS master key provider constructors that were deprecated in version 1.7.x are removed in version 2.0.x. You must explicitly construct AWS KMS master key providers in strict mode or discovery mode.

Encrypt and decrypt ciphertext with key commitment

Version 2.0.x can encrypt and decrypt ciphertext with or without key commitment. Its behavior is determined by the commitment policy setting. By default, it always encrypts with key commitment and only decrypts ciphertext encrypted with key commitment. Unless you change the commitment policy, the AWS Encryption SDK will not decrypt ciphertexts encrypted by any earlier version of the AWS Encryption SDK, including version 1.7.x.

Important

By default, version 2.0.x will not decrypt any ciphertext that was encrypted without key commitment. If your application might encounter a ciphertext that was encrypted without key commitment, set a commitment policy value with AllowDecrypt.

In version 2.0.x, the commitment policy setting has three valid values:

- ForbidEncryptAllowDecrypt — The AWS Encryption SDK cannot encrypt with key commitment. It can decrypt ciphertexts encrypted with or without key commitment.
- RequireEncryptAllowDecrypt — The AWS Encryption SDK must encrypt with key commitment. It can decrypt ciphertexts encrypted with or without key commitment.
- RequireEncryptRequireDecrypt (default) — The AWS Encryption SDK must encrypt with key commitment. It only decrypts ciphertexts with key commitment.
If you are migrating from an earlier version of the AWS Encryption SDK to version 2.0.x, set the commitment policy to a value that assures that you can decrypt all existing ciphertexts that your application might encounter. You are likely to adjust this setting over time.

Version 2.2.x

Adds support for digital signatures and limiting encrypted data keys.

Note

Version 2.x.x of the AWS Encryption SDK for Python, AWS Encryption SDK for JavaScript, and the AWS Encryption CLI are in the end-of-support phase. For information about support and maintenance of this AWS Encryption SDK version in your preferred programming language, see the SUPPORT_POLICY.rst file in its GitHub repository.

Digital signatures

To improve handling of digital signatures when decrypting, the AWS Encryption SDK includes the following features:

- *Non-streaming mode* — returns plaintext only after processing all input, including verifying the digital signature if present. This feature prevents you from using plaintext before verifying the digital signature. Use this feature whenever you decrypt data encrypted with digital signatures (the default algorithm suite). For example, because the AWS Encryption CLI always processes data in streaming mode, use the `-b` parameter when decrypting ciphertext with digital signatures.

- *Unsigned-only decryption mode* — this feature only decrypts unsigned ciphertext. If decryption encounters a digital signature in the ciphertext, the operation fails. Use this feature to avoid unintentionally processing plaintext from signed messages before verifying the signature.

Limiting encrypted data keys

You can limit the number of encrypted data keys in an encrypted message. This feature can help you detect a misconfigured master key provider or keyring when encrypting, or identify a malicious ciphertext when decrypting.
You should limit encrypted data keys when you decrypt messages from an untrusted source. It prevents unnecessary, expensive, and potentially exhaustive calls to your key infrastructure.

**Version 2.3.x**

Adds support for AWS KMS multi-Region keys. For details, see [Using multi-Region AWS KMS keys](#).

---

**Note**

The AWS Encryption CLI supports multi-Region keys beginning in version 3.0.x. Version 2.x.x of the AWS Encryption SDK for Python, AWS Encryption SDK for JavaScript, and the AWS Encryption CLI are in the [end-of-support phase](#). For information about [support and maintenance](#) of this AWS Encryption SDK version in your preferred programming language, see the SUPPORT_POLICY.rst file in its [GitHub repository](#).
Migrating your AWS Encryption SDK

The AWS Encryption SDK supports multiple interoperable [programming language implementations](#), each of which is developed in an open-source repository on GitHub. As a **best practice**, we recommend that you use the latest version of the AWS Encryption SDK for each language.

You can safely upgrade from version 2.0.x or later of AWS Encryption SDK to the latest version. However, the 2.0.x version of the AWS Encryption SDK introduces significant new security features, some of which are breaking changes. To upgrade from versions earlier than 1.7.x to versions 2.0.x and later, you must first upgrade to the latest 1.x version. The topics in this section are designed to help you understand the changes, select the correct version for your application, and migrate safely and successfully to the newest versions of the AWS Encryption SDK.

For information about significant versions of the AWS Encryption SDK, see [Versions of the AWS Encryption SDK](#).

---

⚠️ **Important**

Do not upgrade directly from a version earlier than 1.7.x to version 2.0.x or later without first upgrading to the latest 1.x version. If you upgrade directly to version 2.0.x or later and enable all new features immediately, the AWS Encryption SDK won't be able to decrypt ciphertext encrypted under older versions of the AWS Encryption SDK.

---

🔍 **Note**

The earliest version of the AWS Encryption SDK for .NET is version 3.0.x. All versions of the AWS Encryption SDK for .NET support the security best practices introduced in 2.0.x of the AWS Encryption SDK. You can safely upgrade to the latest version without any code or data changes.

AWS Encryption CLI: When reading this migration guide, use the 1.7.x migration instructions for AWS Encryption CLI 1.8.x and use the 2.0.x migration instructions for AWS Encryption CLI 2.1.x. For details, see [Versions of the AWS Encryption CLI](#).

New security features were originally released in AWS Encryption CLI versions 1.7.x and 2.0.x. However, AWS Encryption CLI version 1.8.x replaces version 1.7.x and AWS
New users

If you're new to the AWS Encryption SDK, install the latest version of the AWS Encryption SDK for your programming language. The default values enable all security features of the AWS Encryption SDK, including encryption with signing, key derivation, and key commitment of the AWS Encryption SDK.

Current users

We recommend that you upgrade from your current version to the latest available version as soon as possible. All 1.x versions of the AWS Encryption SDK are in the end-of-support phase, as are later versions in some programming languages. For details about the support and maintenance status of the AWS Encryption SDK in your programming language, see Support and maintenance.

AWS Encryption SDK versions 2.0.x and later provide new security features to help protect your data. However, AWS Encryption SDK version 2.0.x includes breaking changes that are not backwards compatible. To assure a safe transition, begin by migrating from your current version to the latest 1.x in your programming language. When your latest 1.x version is fully deployed and operating successfully, you can safely migrate to versions 2.0.x and later. This two-step process is critical especially for distributed applications.

For more information about the AWS Encryption SDK security features that underlie these changes, see Improved client-side encryption: Explicit KeyIds and key commitment in the AWS Security Blog.

Looking for help with using the AWS Encryption SDK for Java with the AWS SDK for Java 2.x? See Prerequisites.

Topics

- How to migrate and deploy the AWS Encryption SDK
- Updating AWS KMS master key providers
- Updating AWS KMS keyrings
- Setting your commitment policy
How to migrate and deploy the AWS Encryption SDK

When migrating from an AWS Encryption SDK version earlier than 1.7.x to version 2.0.x or later, you must transition safely to encrypting with key commitment. Otherwise, your application will encounter ciphertexts that it cannot decrypt. If you are using AWS KMS master key providers, you must update to new constructors that create master key providers in strict mode or discovery mode.

**Note**

This topic is designed for users migrating from earlier versions of the AWS Encryption SDK to version 2.0.x or later. If you are new to the AWS Encryption SDK, you can begin using the latest available version immediately with the default settings.

To avoid a critical situation in which you cannot decrypt ciphertext that you need to read, we recommend that you migrate and deploy in multiple distinct stages. Verify that each stage is complete and fully deployed before starting the next stage. This is particularly important for distributed applications with multiple hosts.

**Stage 1: Update your application to the latest 1.x version**

Update to the latest 1.x version for your programming language. Test carefully, deploy your changes, and confirm that the update has propagated to all destination hosts before starting stage 2.

**Important**

Verify that your latest 1.x version is version 1.7.x or later of the AWS Encryption SDK.

The latest 1.x versions of the AWS Encryption SDK are backward compatible with legacy versions of the AWS Encryption SDK and forward compatible with versions 2.0.x and later. They include the new features that are present in version 2.0.x, but include safe defaults designed for this migration. They allow you to upgrade your AWS KMS master key providers, if necessary, and to fully deploy with algorithm suites that can decrypt ciphertext with key commitment.
• Replace deprecated elements, including constructors for legacy AWS KMS master key providers. In Python, be sure to turn on deprecation warnings. Code elements that are deprecated in the latest 1.x versions are removed from versions 2.0.x and later.

• Explicitly set your commitment policy to ForbidEncryptAllowDecrypt. Although this is the only valid value in the latest 1.x versions, this setting is required when you use the APIs introduced in this release. It prevents your application from rejecting ciphertext encrypted without key commitment when you migrate to version 2.0.x and later. For details, see the section called “Setting your commitment policy”.

• If you use AWS KMS master key providers, you must update your legacy master key providers to master key providers that support strict mode and discovery mode. This update is required for the AWS Encryption SDK for Java, AWS Encryption SDK for Python, and the AWS Encryption CLI. If you use master key providers in discovery mode, we recommend that you implement the discovery filter that limits the wrapping keys used to those in particular AWS accounts. This update is optional, but it's a best practice that we recommend. For details, see Updating AWS KMS master key providers.

• If you use AWS KMS discovery keyrings, we recommend that you include a discovery filter that limits the wrapping keys used in decryption to those in particular AWS accounts. This update is optional, but it's a best practice that we recommend. For details, see Updating AWS KMS keyrings.

Stage 2: Update your application to the latest version

After deploying the latest 1.x version successfully to all hosts, you can upgrade to versions 2.0.x and later. Version 2.0.x includes breaking changes for all earlier versions of the AWS Encryption SDK. However, if you make the code changes recommended in Stage 1, you can avoid errors when you migrate to the latest version.

Before you update to the latest version, verify that your commitment policy is consistently set to ForbidEncryptAllowDecrypt. Then, depending on your data configuration, you can migrate at your own pace to RequireEncryptAllowDecrypt and then to the default setting, RequireEncryptRequireDecrypt. We recommend a series of transition steps like the following pattern.

1. Begin with your commitment policy set to ForbidEncryptAllowDecrypt. The AWS Encryption SDK can decrypt messages with key commitment, but it doesn't yet encrypt with key commitment.
2. When you are ready, update your commitment policy to `RequireEncryptAllowDecrypt`. The AWS Encryption SDK begins to encrypt your data with key commitment. It can decrypt ciphertext with and without key commitment.

Before updating your commitment policy to `RequireEncryptAllowDecrypt`, verify that your latest 1.x version is deployed to all hosts, including hosts of any applications that decrypt the ciphertext you produce. Versions of the AWS Encryption SDK prior to version 1.7.x cannot decrypt messages encrypted with key commitment.

This is also a good time to add metrics to your application to measure whether you are still processing ciphertext without key commitment. This will help you determine when it's safe to update your commitment policy setting to `RequireEncryptRequireDecrypt`. For some applications, such as those that encrypt messages in an Amazon SQS queue, this might mean waiting long enough that all ciphertext encrypted under old versions have been re-encrypted or deleted. For other applications, such as encrypted S3 objects, you might need to download, re-encrypt, and re-upload all objects.

3. When you are certain that you don't have any messages encrypted without key commitment, you can update your commitment policy to `RequireEncryptRequireDecrypt`. This value assures that your data is always encrypted and decrypted with key commitment. This setting is the default, so you aren't required to set it explicitly, but we recommend it. An explicit setting will aid debugging and any potential rollbacks that might be required if your application encounters ciphertext encrypted without key commitment.

### Updating AWS KMS master key providers

To migrate to the latest 1.x version of the AWS Encryption SDK, and then to version 2.0.x or later, you must replace legacy AWS KMS master key providers with master key providers created explicitly in strict mode or discovery mode. Legacy master key providers are deprecated in version 1.7.x and removed in version 2.0.x. This change is required for applications and scripts that use the AWS Encryption SDK for Java, AWS Encryption SDK for Python, and the AWS Encryption CLI. The examples in this section will show you how to update your code.

> **Note**
> In Python, turn on deprecation warnings. This will help you identify the parts of your code that you need to update.
If you are using an AWS KMS master key (not a master key provider), you can skip this step. AWS KMS master keys are not deprecated or removed. They encrypt and decrypt only with the wrapping keys that you specify.

The examples in this section focus on the elements of your code that you need to change. For a complete example of the updated code, see the Examples section of the GitHub repository for your programming language. Also, these examples typically use key ARNs to represent AWS KMS keys. When you create a master key provider for encrypting, you can use any valid AWS KMS key identifier to represent an AWS KMS key. When you create a master key provider for decrypting, you must use a key ARN.

Learn more about migration

For all AWS Encryption SDK users, learn about setting your commitment policy in the section called “Setting your commitment policy”.

For AWS Encryption SDK for C and AWS Encryption SDK for JavaScript users, learn about an optional update to keyrings in Updating AWS KMS keyrings.

Topics

- Migrating to strict mode
- Migrating to discovery mode

Migrating to strict mode

After updating to the latest 1.x version of the AWS Encryption SDK, replace your legacy master key providers with master key providers in strict mode. In strict mode, you must specify the wrapping keys to use when encrypting and decrypting. The AWS Encryption SDK uses only the wrapping keys you specify. Deprecated master key providers can decrypt data using any AWS KMS key that encrypted a data key, including AWS KMS keys in different AWS accounts and Regions.

Master key providers in strict mode are introduced in the AWS Encryption SDK version 1.7.x. They replace legacy master key providers, which are deprecated in 1.7.x and removed in 2.0.x. Using master key providers in strict mode is an AWS Encryption SDK best practice.

The following code creates a master key provider in strict mode that you can use for encrypting and decrypting.
Java

This example represents code in an application that uses the version 1.6.2 or earlier of the AWS Encryption SDK for Java.

This code uses the KmsMasterKeyProvider.builder() method to instantiate an AWS KMS master key provider that uses one AWS KMS key as a wrapping key.

```java
// Create a master key provider
// Replace the example key ARN with a valid one
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .withKeysForEncryption(awsKmsKey)
    .build();
```

This example represents code in an application that uses version 1.7.x or later of the AWS Encryption SDK for Java. For a complete example, see BasicEncryptionExample.java.

The Builder.build() and Builder.withKeysForEncryption() methods used in the previous example are deprecated in version 1.7.x and are removed from version 2.0.x.

To update to a strict mode master key provider, this code replaces calls to deprecated methods with a call to the new Builder.buildStrict() method. This example specifies one AWS KMS key as the wrapping key, but the Builder.buildStrict() method can take a list of multiple AWS KMS keys.

```java
// Create a master key provider in strict mode
// Replace the example key ARN with a valid one from your AWS account.
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);
```

Python

This example represents code in an application that uses version 1.4.1 of the AWS Encryption SDK for Python. This code uses KMSMasterKeyProvider, which is deprecated in version 1.7.x and removed from version 2.0.x. When decrypting, it uses any AWS KMS key that encrypted a data key without regard to the AWS KMS keys you specify.

Migrating to strict mode
Note that KMSMasterKey is not deprecated or removed. When encrypting and decrypting, it uses only the AWS KMS key you specify.

```python
# Create a master key provider
# Replace the example key ARN with a valid one
key_1 = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"

aws_kms_master_key_provider = KMSMasterKeyProvider(  
    key_ids=[key_1, key_2]
)
```

This example represents code in an application that uses version 1.7.x of the AWS Encryption SDK for Python. For a complete example, see `basic_encryption.py`.

To update to a strict mode master key provider, this code replaces the call to KMSMasterKeyProvider() with a call to StrictAwsKmsMasterKeyProvider().

```python
# Create a master key provider in strict mode
# Replace the example key ARNs with valid values from your AWS account
key_1 = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(  
    key_ids=[key_1, key_2]
)
```

AWS Encryption CLI

This example shows how to encrypt and decrypt using the AWS Encryption CLI version 1.1.7 or earlier.

In version 1.1.7 and earlier, when encrypting, you specify one or more master keys (or wrapping keys), such as an AWS KMS key. When decrypting, you can't specify any wrapping keys unless you are using a custom master key provider. The AWS Encryption CLI can use any wrapping key that encrypted a data key.

```bash
\ Replace the example key ARN with a valid one
```
This example shows how to encrypt and decrypt using the AWS Encryption CLI version 1.7.x or later. For complete examples, see [Examples of the AWS Encryption CLI](#).

The `--master-keys` parameter is deprecated in version 1.7.x and removed in version 2.0.x. It's replaced by `--wrapping-keys` parameter, which is required in encrypt and decrypt commands. This parameter supports strict mode and discovery mode. Strict mode is an AWS Encryption SDK best practice that assures that you use the wrapping key that you intend.

To upgrade to strict mode, use the `key` attribute of the `--wrapping-keys` parameter to specify a wrapping key when encrypting and decrypting.

Migrating to strict mode
Migrating to discovery mode

Beginning in version 1.7.x, it's an AWS Encryption SDK best practice to use strict mode for AWS KMS master key providers, that is, to specify wrapping keys when encrypting and decrypting. You must always specify wrapping keys when encrypting. But there are situations in which specifying the key ARNs of AWS KMS keys for decrypting is impractical. For example, if you're using aliases to identify AWS KMS keys when encrypting, you lose the benefit of aliases if you have to list key ARNs when decrypting. Also, because master key providers in discovery mode behave like the original master key providers, you might use them temporarily as part of your migration strategy, and then upgrade to master key providers in strict mode later.

In cases like this, you can use master key providers in discovery mode. These master key providers don't let you specify wrapping keys, so you cannot use them for encrypting. When decrypting, they can use any wrapping key that encrypted a data key. But unlike legacy master key providers, which behave the same way, you create them in discovery mode explicitly. When using master key providers in discovery mode, you can limit the wrapping keys that can be used to those in particular AWS accounts. This discovery filter is optional, but it's a best practice that we recommend. For information about AWS partitions and accounts, see Amazon Resource Names in the AWS General Reference.

The following examples create an AWS KMS master key provider in strict mode for encrypting and an AWS KMS master key provider in discovery mode for decrypting. The master key provider in discovery mode uses a discovery filter to limit the wrapping keys used for decrypting to the aws partition and to particular example AWS accounts. Although the account filter is not necessary in this very simple example, it's a best practice that is very beneficial when one application encrypts data and a different application decrypts the data.

Java

This example represents code in an application that uses version 1.7.x or later of the AWS Encryption SDK for Java. For a complete example, see DiscoveryDecryptionExample.java.

To instantiate a master key provider in strict mode for encrypting, this example uses the Builder.buildStrict() method. To instantiate a master key provider in
discovery mode for decrypting, it uses the `Builder.buildDiscovery()` method. The `Builder.buildDiscovery()` method takes a `DiscoveryFilter` that limits the AWS Encryption SDK to AWS KMS keys in the specified AWS partition and accounts.

```java
// Create a master key provider in strict mode for encrypting
// Replace the example alias ARN with a valid one from your AWS account.

KmsMasterKeyProvider encryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

// Create a master key provider in discovery mode for decrypting
// Replace the example account IDs with valid values.
DiscoveryFilter accounts = new DiscoveryFilter("aws", Arrays.asList("111122223333", "444455556666"));

KmsMasterKeyProvider decryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildDiscovery(accounts);
```

### Python

This example represents code in an application that uses version 1.7.x or later of the AWS Encryption SDK for Python. For a complete example, see `discovery_kms_provider.py`.

To create a master key provider in strict mode for encrypting, this example uses `StrictAwsKmsMasterKeyProvider`. To create a master key provider in discovery mode for decrypting, it uses `DiscoveryAwsKmsMasterKeyProvider` with a `DiscoveryFilter` that limits the AWS Encryption SDK to AWS KMS keys in the specified AWS partition and accounts.

```python
# Create a master key provider in strict mode
# Replace the example key ARN and alias ARNs with valid values from your AWS account.
key_1 = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias"
key_2 = "arn:aws:kms:us-west-2:444455556666:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[key_1, key_2])

# Create a master key provider in discovery mode for decrypting
# Replace the example account IDs with valid values
```
accounts = DiscoveryFilter(
    partition="aws",
    account_ids=['111122223333', '444455556666']
)
aws_kms_master_key_provider = DiscoveryAwsKmsMasterKeyProvider(
    discovery_filter=accounts
)

AWS Encryption CLI

This example shows how to encrypt and decrypt using the AWS Encryption CLI version 1.7.x or later. Beginning in version 1.7.x, the --wrapping-keys parameter is required when encrypting and decrypting. The --wrapping-keys parameter supports strict mode and discovery mode. For complete examples, see the section called "Examples".

When encrypting, this example specifies a wrapping key, which is required. When decrypting, it explicitly chooses discovery mode by using the discovery attribute of the --wrapping-keys parameter with a value of true.

To limit the wrapping keys that the AWS Encryption SDK can use in discovery mode to those in particular AWS accounts, this example uses the discovery-partition and discovery-account attributes of the --wrapping-keys parameter. These optional attributes are valid only when the discovery attribute is set to true. You must use the discovery-partition and discovery-account attributes together; neither is valid alone.

```bash
# Replace the example key ARN with a valid value

# Encrypt your plaintext data
$ aws-encryption-cli --encrypt
    --input hello.txt
    --wrapping-keys key=$keyAlias
    --metadata-output ~/metadata
    --encryption-context purpose=test
    --output .

# Decrypt your ciphertext
# Replace the example account IDs with valid values
$ aws-encryption-cli --decrypt
    --input hello.txt.encrypted
    --wrapping-keys discovery=true
    discovery-partition=aws
```

Migrating to discovery mode
Updating AWS KMS keyrings

The AWS KMS keyrings in the AWS Encryption SDK for C, the AWS Encryption SDK for .NET, and the AWS Encryption SDK for JavaScript support best practices by allowing you to specify wrapping keys when encrypting and decrypting. If you create an AWS KMS discovery keyring, you do so explicitly.

Note

The earliest version of the AWS Encryption SDK for .NET is version 3.0.x. All versions of the AWS Encryption SDK for .NET support the security best practices introduced in 2.0.x of the AWS Encryption SDK. You can safely upgrade to the latest version without any code or data changes.

When you update to the latest 1.x version of the AWS Encryption SDK, you can use a discovery filter to limit the wrapping keys that an AWS KMS discovery keyring or AWS KMS regional discovery keyring uses when decrypting to those in particular AWS accounts. Filtering a discovery keyring is an AWS Encryption SDK best practice.

The examples in this section will show you how to add the discovery filter to an AWS KMS regional discovery keyring.

Learn more about migration

For all AWS Encryption SDK users, learn about setting your commitment policy in the section called “Setting your commitment policy”.

For AWS Encryption SDK for Java, AWS Encryption SDK for Python, and AWS Encryption CLI users, learn about a required update to master key providers in the section called “Updating AWS KMS master key providers”.

```
discovery-account=111122223333 \
  discovery-account=444455556666 \
  --encryption-context purpose=test \
  --metadata-output ~/metadata \
  --output .
```
You might have code like the following in your application. This example creates an AWS KMS regional discovery keyring that can only use wrapping keys in the US West (Oregon) (us-west-2) Region. This example represents code in AWS Encryption SDK versions earlier than 1.7.x. However, it is still valid in versions 1.7.x and later.

C

```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

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

Beginning in version 1.7.x, you can add a discovery filter to any AWS KMS discovery keyring. This discovery filter limits the AWS KMS keys that the AWS Encryption SDK can use for decryption to those in the specified partition and accounts. Before using this code, change the partition, if necessary, and replace the example account IDs with valid ones.

C

For a complete example, see `kms_discovery.cpp`.

```c
std::shared_ptr<KmsKeyring::DiscoveryFilter> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
        .AddAccount("111122223333")
        .AddAccount("444455556666")
    .Build());
```
Setting your commitment policy

**Key commitment** assures that your encrypted data always decrypts to the same plaintext. To provide this security property, beginning in version 1.7.x, the AWS Encryption SDK uses new **algorithm suites** with key commitment. To determine whether your data is encrypted and decrypted with key commitment, use the **commitment policy** configuration setting. Encrypting and decrypting data with key commitment is an **AWS Encryption SDK best practice**.

Setting a commitment policy is an important part of the second step in the migration process — migrating from the latest 1.x versions of the AWS Encryption SDK to versions 2.0.x and later.
After setting and changing your commitment policy, be sure to test your application thoroughly before deploying it in production. For migration guidance, see How to migrate and deploy the AWS Encryption SDK.

The commitment policy setting has three valid values in versions 2.0.x and later. In the latest 1.x versions (beginning with version 1.7.x), only ForbidEncryptAllowDecrypt is valid.

- **ForbidEncryptAllowDecrypt** — The AWS Encryption SDK cannot encrypt with key commitment. It can decrypt ciphertexts encrypted with or without key commitment.

  In the latest 1.x versions, this is the only valid value. It ensures that you don't encrypt with key commitment until you are fully prepared to decrypt with key commitment. Setting the value explicitly prevents your commitment policy from changing automatically to require-encrypt-require-decrypt when you upgrade to versions 2.0.x or later. Instead, you can migrate your commitment policy in stages.

- **RequireEncryptAllowDecrypt** — The AWS Encryption SDK always encrypts with key commitment. It can decrypt ciphertexts encrypted with or without key commitment. This value is added in version 2.0.x.

- **RequireEncryptRequireDecrypt** — The AWS Encryption SDK always encrypts and decrypts with key commitment. This value is added in version 2.0.x. It is the default value in versions 2.0.x and later.

In the latest 1.x versions, the only valid commitment policy value is ForbidEncryptAllowDecrypt. After you migrate to version 2.0.x or later, you can change your commitment policy in stages as you are ready. Don't update your commitment policy to RequireEncryptRequireDecrypt until you are certain that you don't have any messages encrypted without key commitment.

These examples show you how to set your commitment policy in the latest 1.x versions and in versions 2.0.x and later. The technique depends on your programming language.

**Learn more about migration**

For AWS Encryption SDK for Java, AWS Encryption SDK for Python, and the AWS Encryption CLI, learn about required changes to master key providers in the section called “Updating AWS KMS master key providers”.

For AWS Encryption SDK for C and AWS Encryption SDK for JavaScript, learn about an optional update to keyrings in Updating AWS KMS keyrings.
How to set your commitment policy

The technique that you use to set your commitment policy differs slightly with each language implementation. These examples show you how to do it. Before changing your commitment policy, review the multi-stage approach in How to migrate and deploy.

C

Beginning in version 1.7.x of the AWS Encryption SDK for C, you use the aws_cryptosdk_session_set_commitment_policy function to set the commitment policy on your encrypt and decrypt sessions. The commitment policy that you set applies to all encrypt and decrypt operations called on that session.

The aws_cryptosdk_session_new_from_keyring and aws_cryptosdk_session_new_from_cmm functions are deprecated in version 1.7.x and removed in version 2.0.x. These functions are replaced by aws_cryptosdk_session_new_from_keyring_2 and aws_cryptosdk_session_new_from_cmm_2 functions that return a session.

When you use the aws_cryptosdk_session_new_from_keyring_2 and aws_cryptosdk_session_new_from_cmm_2 in the latest 1.x versions, you are required to call the aws_cryptosdk_session_set_commitment_policy function with the COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT commitment policy value. In versions 2.0.x and later, calling this function is optional and it takes all valid values. The default commitment policy for versions 2.0.x and later is COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT.

For a complete example, see string.cpp.

```c
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Create an AWS KMS keyring */
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);

/* Create an encrypt session with a CommitmentPolicy setting */
struct aws_cryptosdk_session *encrypt_session =
    aws_cryptosdk_session_new_from_keyring_2(
```
**aws_cryptosdk_keyring_release(kms_keyring);**

**aws_cryptosdk_session_set_commitment_policy(encrypt_session,**

  COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);**

...  

/* Encrypt your data */

size_t plaintext_consumed_output;  
aws_cryptosdk_session_process(encrypt_session,**
  ciphertext_output,**
  ciphertext_buf_sz_output,**
  ciphertext_len_output,**
  plaintext_input,**
  plaintext_len_input,**
  &plaintext_consumed_output)**

...  

/* Create a decrypt session with a CommitmentPolicy setting */

struct aws_cryptosdk_keyring *kms_keyring =
  Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);  
struct aws_cryptosdk_session *decrypt_session =
  *aws_cryptosdk_session_new_from_keyring_2(**
    alloc, AWS_CRYPTOSDK_DECRYPT, kms_keyring);**
aws_cryptosdk_keyring_release(kms_keyring);  
aws_cryptosdk_session_set_commitment_policy(decrypt_session,**
  COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);**

/ * Decrypt your ciphertext */

size_t ciphertext_consumed_output;  
aws_cryptosdk_session_process(decrypt_session,**
  plaintext_output,**
  plaintext_buf_sz_output,**
  plaintext_len_output,**
  ciphertext_input,**
  ciphertext_len_input,**
  &ciphertext_consumed_output)**

**C#/.NET**

The require-encrypt-require-decrypt value is the default commitment policy in all versions of the AWS Encryption SDK for .NET. You can set it explicitly as a best practice,
but it's not required. However, if you are using the AWS Encryption SDK for .NET to decrypt ciphertext that was encrypted by another language implementation of the AWS Encryption SDK without key commitment, you need to change the commitment policy value to REQUIRE_ENCRYPT_ALLOW_DECRYPT or FORBID_ENCRYPT_ALLOW_DECRYPT. Otherwise, attempts to decrypt the ciphertext will fail.

In the AWS Encryption SDK for .NET, you set the commitment policy on an instance of the AWS Encryption SDK. Instantiate an AwsEncryptionSdkConfig object with a CommitmentPolicy parameter, and use the configuration object to create the AWS Encryption SDK instance. Then, call the Encrypt() and Decrypt() methods of the configured AWS Encryption SDK instance.

This example sets the commitment policy to require-encrypt-allow-decrypt.

```csharp
// Instantiate the material providers
var materialProviders = 
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig
{ 
    CommitmentPolicy = CommitmentPolicy.REQUIRE_ENCRYPT.Allow_DECRYPT
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

string keyArn = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

var encryptionContext = new Dictionary<string, string>()
{
    {"purpose", "test"} encryption
};

var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var keyring = materialProviders.CreateAwsKmsKeyring(createKeyringInput);

// Encrypt your plaintext data
var encryptInput = new EncryptInput

AWS Encryption SDK

```javascript
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);

// Decrypt your ciphertext
var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = keyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

AWS Encryption CLI

To set a commitment policy in the AWS Encryption CLI, use the `--commitment-policy` parameter. This parameter is introduced in version 1.8.x.

In the latest 1.x version, when you use the `--wrapping-keys` parameter in an `--encrypt` or `--decrypt` command, a `--commitment-policy` parameter with the `forbid-encrypt-allow-decrypt` value is required. Otherwise, the `--commitment-policy` parameter is invalid.

In versions 2.1.x and later, the `--commitment-policy` parameter is optional and defaults to the `require-encrypt-require-decrypt` value, which won't encrypt or decrypt any ciphertext encrypted without key commitment. However, we recommend that you set the commitment policy explicitly in all encrypt and decrypt calls to help with maintenance and troubleshooting.

This example sets the commitment policy. It also uses the `--wrapping-keys` parameter that replaces the `--master-keys` parameter beginning in version 1.8.x. For details, see the section called “Updating AWS KMS master key providers”. For complete examples, see Examples of the AWS Encryption CLI.

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

\ Encrypt your plaintext data - no change to algorithm suite used
$ aws-encryption-cli --encrypt
    --input hello.txt

How to set your commitment policy
Java

Beginning in version 1.7.x of the AWS Encryption SDK for Java, you set the commitment policy on your instance of AwsCrypto, the object that represents the AWS Encryption SDK client. This commitment policy setting applies to all encrypt and decrypt operations called on that client.

The AwsCrypto() constructor is deprecated in the latest 1.x versions of the AWS Encryption SDK for Java and is removed in version 2.0.x. It's replaced by a new Builder class, a Builder.withCommitmentPolicy() method, and the CommitmentPolicy enumerated type.

In the latest 1.x versions, the Builder class requires the Builder.withCommitmentPolicy() method and the CommitmentPolicy.ForbidEncryptAllowDecrypt argument. Beginning in version 2.0.x, the Builder.withCommitmentPolicy() method is optional; the default value is CommitmentPolicy.RequireEncryptRequireDecrypt.

For a complete example, see SetCommitmentPolicyExample.java.

```
// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.ForbidEncryptAllowDecrypt)
    .build();

// Create a master key provider in strict mode
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
```
KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
  .buildStrict(awsKmsKey);

// Encrypt your plaintext data
CryptoResult<byte[], KmsMasterKey> encryptResult = crypto.encryptData(
  masterKeyProvider,
  sourcePlaintext,
  encryptionContext);
byte[] ciphertext = encryptResult.getResult();

// Decrypt your ciphertext
CryptoResult<byte[], KmsMasterKey> decryptResult = crypto.decryptData(
  masterKeyProvider,
  ciphertext);
byte[] decrypted = decryptResult.getResult();

### JavaScript

Beginning in version 1.7.x of the AWS Encryption SDK for JavaScript, you can set the commitment policy when you call the new buildClient function that instantiates an AWS Encryption SDK client. The buildClient function takes an enumerated value that represents your commitment policy. It returns updated encrypt and decrypt functions that enforce your commitment policy when you encrypt and decrypt.

In the latest 1.x versions, the buildClient function requires the CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT argument. Beginning in version 2.0.x, the commitment policy argument is optional and the default value is CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT.

The code for Node.js and the browser are identical for this purpose, except that browser needs a statement to set credentials.

The following example encrypts data with an AWS KMS keyring. The new buildClient function sets the commitment policy to FORBID_ENCRYPT_ALLOW_DECRYPT, the default value in the latest 1.x versions. The upgraded encrypt and decrypt functions that buildClient returns enforce the commitment policy you set.

```javascript
import { buildClient } from '@aws-crypto/client-node'
const { encrypt, decrypt } = 
  buildClient(CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT)
```
// Create an AWS KMS keyring
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias'
const keyIds = ['arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })

// Encrypt your plaintext data
const { ciphertext } = await encrypt(keyring, plaintext, { encryptionContext: context })

// Decrypt your ciphertext
const { decrypted, messageHeader } = await decrypt(keyring, ciphertext)

Python

Beginning in version 1.7.x of the AWS Encryption SDK for Python, you set the commitment policy on your instance of EncryptionSDKClient, a new object that represents the AWS Encryption SDK client. The commitment policy that you set applies to all encrypt and decrypt calls that use that instance of the client.

In the latest 1.x versions, the EncryptionSDKClient constructor requires the CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT enumerated value. Beginning in version 2.0.x, the commitment policy argument is optional and the default value is CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT.

This example uses the new EncryptionSDKClient constructor and sets the commitment policy to the 1.7.x default value. The constructor instantiates a client that represents the AWS Encryption SDK. When you call the encrypt, decrypt, or stream methods on this client, they enforce the commitment policy that you set. This example also uses the new constructor for the StrictAwsKmsMasterKeyProvider class, which specifies AWS KMS keys when encrypting and decrypting.

For a complete example, see set_commitment.py.

# Instantiate the client
client =
  aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT)

// Create a master key provider in strict mode
aws_kms_key = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
aws_kms_strict_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[aws_kms_key]
)

# Encrypt your plaintext data
ciphertext, encrypt_header = client.encrypt(
    source=source_plaintext,
    encryption_context=encryption_context,
    master_key_provider=aws_kms_strict_master_key_provider
)

# Decrypt your ciphertext
decrypted, decrypt_header = client.decrypt(
    source=ciphertext,
    master_key_provider=aws_kms_strict_master_key_provider
)

Troubleshooting migration to the latest versions

Before updating your application to version 2.0.x or later of the AWS Encryption SDK, update to the latest 1.x version of the AWS Encryption SDK and deploy it completely. That will help you avoid most errors you might encounter when updating to versions 2.0.x and later. For detailed guidance, including examples, see Migrating your AWS Encryption SDK.

⚠️ Important
Verify that your latest 1.x version is version 1.7.x or later of the AWS Encryption SDK.

🔍 Note
AWS Encryption CLI: References in this guide to version 1.7.x of the AWS Encryption SDK apply to version 1.8.x of the AWS Encryption CLI. References in this guide to version 2.0.x of the AWS Encryption SDK apply to 2.1.x of the AWS Encryption CLI.

New security features were originally released in AWS Encryption CLI versions 1.7.x and 2.0.x. However, AWS Encryption CLI version 1.8.x replaces version 1.7.x and AWS Encryption CLI 2.1.x replaces 2.0.x. For details, see the relevant security advisory in the aws-encryption-sdk-cli repository on GitHub.
This topic is designed to help you recognize and resolve the most common errors you might encounter.

**Topics**

- Deprecated or removed objects
- Configuration conflict: Commitment policy and algorithm suite
- Configuration conflict: Commitment policy and ciphertext
- Key commitment validation failed
- Other encryption failures
- Other decryption failures
- Rollback considerations

### Deprecated or removed objects

Version 2.0.x includes several breaking changes, including removing legacy constructors, methods, functions, and classes that were deprecated in version 1.7.x. To avoid compiler errors, import errors, syntax errors, and symbol not found errors (depending on your programming language), upgrade first to the latest 1.x version of the AWS Encryption SDK for your programming language. (This must be version 1.7.x or later.) While using the latest 1.x version, you can begin using the replacement elements before the original symbols are removed.

If you need to upgrade to version 2.0.x or later immediately, consult the changelog for your programming language, and replace the legacy symbols with the symbols the changelog recommends.

### Configuration conflict: Commitment policy and algorithm suite

If you specify an algorithm suite that conflicts with your commitment policy, the call to encrypt fails with a `Configuration conflict` error.

To avoid this type of error, don't specify an algorithm suite. By default, the AWS Encryption SDK chooses the most secure algorithm that is compatible with your commitment policy. However, if you must specify an algorithm suite, such as one without signing, be sure to choose an algorithm suite that is compatible with your commitment policy.
### Commitment policy

<table>
<thead>
<tr>
<th>Commitment policy</th>
<th>Compatible algorithm suites</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForbidEncryptAllowDecrypt</td>
<td>Any algorithm suite <em>without</em> key commitment, such as:</td>
</tr>
<tr>
<td></td>
<td>AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384 (03 78) (with signing)</td>
</tr>
<tr>
<td></td>
<td>AES_256_GCM_IV12_TAG16_HKDF_SHA256 (01 78) (without signing)</td>
</tr>
<tr>
<td>RequireEncryptAllowDecrypt</td>
<td>Any algorithm suite <em>with</em> key commitment, such as:</td>
</tr>
<tr>
<td>RequireEncryptRequireDecrypt</td>
<td>AES_256_GCM_HKDF_SHA512_COMMIT_KEY_ECDSA_P384 (05 78) (with signing)</td>
</tr>
<tr>
<td></td>
<td>AES_256_GCM_HKDF_SHA512_COMMIT_KEY (04 78) (without signing)</td>
</tr>
</tbody>
</table>

If you encounter this error when you have not specified an algorithm suite, the conflicting algorithm suite might have been chosen by your [cryptographic materials manager](https://example.com) (CMM). The Default CMM won’t select a conflicting algorithm suite, but a custom CMM might. For help, consult the documentation for your custom CMM.

### Configuration conflict: Commitment policy and ciphertext

The `RequireEncryptRequireDecrypt` commitment policy does not permit the AWS Encryption SDK to decrypt a message that was encrypted without *key commitment*. If you ask the AWS Encryption SDK to decrypt a message without key commitment, it returns a *Configuration conflict* error.

To avoid this error, before setting the `RequireEncryptRequireDecrypt` commitment policy, be sure that all ciphertexts encrypted without key commitment are decrypted and re-encrypted with key commitment, or handled by a different application. If you encounter this error, you can return an error for the conflicting ciphertext or change your commitment policy temporarily to `RequireEncryptAllowDecrypt`. 
If you are encountering this error because you upgraded to version 2.0.x or later from a version earlier than 1.7.x without first upgrading to the latest 1.x version (version 1.7.x or later), consider rolling back to the latest 1.x version and deploying that version to all hosts before upgrading to version 2.0.x or later. For help, see How to migrate and deploy the AWS Encryption SDK.

Key commitment validation failed

When you decrypt messages that are encrypted with key commitment, you might get a Key commitment validation failed error message. This indicates that the decrypt call failed because a data key in an encrypted message is not identical to the unique data key for the message. By validating the data key during decryption, key commitment protects you from decrypting a message that might result in more than one plaintext.

This error indicates that the encrypted message that you were trying to decrypt was not returned by the AWS Encryption SDK. It might be a manually crafted message or the result of data corruption. If you encounter this error, your application can reject the message and continue, or stop processing new messages.

Other encryption failures

Encryption can fail for multiple reasons. You cannot use an AWS KMS discovery keyring or a master key provider in discovery mode to encrypt a message.

Be sure that you specify a keyring or master key provider with wrapping keys that you have permission to use for encryption. For help with permissions on AWS KMS keys, see Viewing a key policy and Determining access to an AWS KMS key in the AWS Key Management Service Developer Guide.

Other decryption failures

If your attempt to decrypt an encrypted message fails, it means that the AWS Encryption SDK could not (or would not) decrypt any of the encrypted data keys in the message.

If you used a keyring or master key provider that specifies wrapping keys, the AWS Encryption SDK uses only the wrapping keys you specify. Verify that you are using the wrapping keys that you intend and that you have kms:Decrypt permission on at least one of the wrapping keys. If you are using AWS KMS keys, as a fallback, you can try decrypting the message with an AWS KMS discovery keyring or a master key provider in discovery mode. If the operation succeeds, before returning the plaintext, verify that the key used to decrypt the message is one that you trust.
Rollback considerations

If your application is failing to encrypt or decrypt data, you can usually resolve the problem by updating the code symbols, keyrings, master key providers, or commitment policy. However, in some cases, you might decide that it's best to roll back your application to a previous version of the AWS Encryption SDK.

If you must roll back, do so with caution. Versions of the AWS Encryption SDK prior to 1.7.x cannot decrypt ciphertext encrypted with key commitment.

- Rolling back from the latest 1.x version to a previous version of the AWS Encryption SDK is generally safe. You might have to undo changes you made to your code to use symbols and objects that are not supported in previous versions.
- Once you have begun encrypting with key commitment (setting your commitment policy to RequireEncryptAllowDecrypt) in version 2.0.x or later, you can roll back to version 1.7.x, but not to any earlier version. Versions of the AWS Encryption SDK prior to 1.7.x cannot decrypt ciphertext encrypted with key commitment.

If you accidentally enable encrypting with key commitment before all hosts can decrypt with key commitment, it might be best to continue with the roll out rather than to roll back. If messages are transient or can be safely dropped, then you might consider a rollback with loss of messages. If a rollback is required, you might consider writing a tool that decrypts and re-encrypts all messages.
Frequently asked questions

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

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

The AWS SDKs provide libraries for interacting with Amazon Web Services (AWS), including AWS Key Management Service (AWS KMS). Some of the language implementations of the AWS Encryption SDK, such as the AWS Encryption SDK for .NET, always require the AWS SDK in the same programming language. Other language implementations require the corresponding AWS SDK only when you use AWS KMS keys in your keyrings or master key providers. For details, see the topic about your programming language in AWS Encryption SDK programming languages.

You can use the AWS SDKs to interact with AWS KMS, including encrypting and decrypting small amounts of data (up to 4,096 bytes with a symmetric encryption key) and generating data keys for client-side encryption. However, when you generate a data key, you must manage the entire encryption and decryption process, including encrypting your data with the data key outside of AWS KMS, safely discarding the plaintext data key, storing the encrypted data key, and then decrypting the data key and decrypting your data. The AWS Encryption SDK handles this process for you.

The AWS Encryption SDK provides a library that encrypts and decrypts data using industry standards and best practices. It generates the data key, encrypts it under the wrapping keys you
specify, and returns an *encrypted message*, a portable data object that includes the encrypted data and the encrypted data keys you need to decrypt it. When it's time to decrypt, you pass in the encrypted message and at least one of the wrapping keys (optional), and the AWS Encryption SDK returns your plaintext data.

You can use AWS KMS keys as wrapping keys in the AWS Encryption SDK, but it is not required. You can use encryption keys that you generate and those from your key manager or on-premises hardware security module. You can use the AWS Encryption SDK even if you don't have an AWS account.

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

The [Amazon S3 encryption client](https://aws.amazon.com/s3/) 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) symmetric algorithm in Galois/Counter Mode (GCM), known as AES-GCM, to encrypt your data. It lets you choose from several symmetric and asymmetric algorithms to encrypt the data keys that encrypt your data.

For AES-GCM, the default algorithm suite is AES-GCM with a 256-bit key, key derivation (HKDF), digital signatures, and key commitment. AWS Encryption SDK also supports 192-bit, and 128-bit encryption keys and encryption algorithms without digital signatures and key commitment.

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](#).

For implementation details about the supported algorithms, see [Algorithms reference](#).
How is the initialization vector (IV) generated and where is it stored?

The AWS Encryption SDK uses a deterministic method to construct a different IV value for each frame. This procedure guarantees that IVs are never repeated within a message. (Prior to version 1.3.0 of the AWS Encryption SDK for Java and the AWS Encryption SDK for Python, the AWS Encryption SDK randomly generated a unique IV value for each frame.)

The IV is stored in the encrypted message that the AWS Encryption SDK returns. For more information, see the AWS Encryption SDK message format reference.

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

The method depends on the keyring or master key provider you use.

The AWS KMS keyrings and master key providers in the AWS Encryption SDK use the AWS KMS GenerateDataKey API operation to generate each data key and encrypt it under its wrapping key. To encrypt copies of the data key under additional KMS keys, they use the AWS KMS Encrypt operation. To decrypt the data keys, they use the AWS KMS Decrypt operation. For details, see AWS KMS keyring in the AWS Encryption SDK Specification in GitHub.

Other keyrings generate the data key, encrypt, and decrypt using best practice methods for each programming language. For details, see the specification of the keyring or master key provider in the Framework section of the AWS Encryption SDK Specification in GitHub.

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 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, 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. The message header contains the encrypted data keys and information about how the message body is formed. The message body contains the encrypted data. If the algorithm suite includes a digital signature, the message format includes a footer that contains the signature. For more information, see AWS Encryption SDK message format reference.
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 wrapping keys or master keys
- The frame size (when \textit{framed data} is used)

When you use the AWS Encryption SDK with its default configuration (one AWS KMS key as the wrapping key (or master key), no AAD, nonframed data, and an encryption algorithm with signing), 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 \textit{AWS Encryption SDK message format reference}.

Can I use my own master key provider?

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

Can I encrypt data under more than one wrapping key?

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

To encrypt data under multiple wrapping keys, create a keyring or master key provider with multiple wrapping keys. When working with keyrings, you can create a \textit{single keyring with multiple wrapping keys} or a \textit{multi-keyring}.

When you encrypt data with multiple wrapping keys, the AWS Encryption SDK uses one wrapping key to generate a plaintext data key. The data key is unique and mathematically unrelated to the wrapping key. The operation returns the plaintext data key and a copy of the data key encrypted by the wrapping key. Then the encryption method, encrypts the data key with the other wrapping keys. The resulting \textit{encrypted message} includes the encrypted data and one encrypted data key for each wrapping key.
The encrypted message can be decrypted by using any one of the wrapping keys used in the encryption operation. The AWS Encryption SDK uses a wrapping key to decrypt an encrypted data key. Then, it uses the plaintext data key to decrypt the data.

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

Most programming language implementations of the AWS Encryption SDK can encrypt raw bytes (byte arrays), I/O streams (byte streams), and strings. The AWS Encryption SDK for .NET does not support I/O streams. We provide example code for each of the supported [programming languages](#).

**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](#) that supports streaming.

The AWS Encryption SDK for .NET does not support I/O streams.
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.

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in GitHub.

The AWS Encryption SDK uses the supported algorithms 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
- AWS Encryption SDK message format examples
- Body additional authenticated data (AAD) reference for the AWS Encryption SDK
- AWS Encryption SDK algorithms reference
- AWS Encryption SDK initialization vector reference
- AWS KMS Hierarchical keyring technical details

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.
For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the [AWS Encryption SDK Specification](#) in GitHub.

The encryption operations in the AWS Encryption SDK return a single data structure or **encrypted 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).

The [algorithms suites](#) supported by the AWS Encryption SDK use one of two message format versions. Algorithm suites without *key commitment* use message format version 1. Algorithm suites with key commitment use message format version 2.

**Topics**

- Header structure
- Body structure
- Footer structure

**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 in message format versions 1 and 2. The bytes are appended in the order shown.

The **Not present** value indicates that the field doesn't exist in that version of the message format. **Bold text** indicates values that are different in each version.

<table>
<thead>
<tr>
<th>Field</th>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example field</td>
<td>Present</td>
<td>Not present</td>
</tr>
</tbody>
</table>

**Note**

You might need to scroll horizontally or vertically to see all of the data in this table.
## Header Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Message format version 1</th>
<th>Message format version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length (bytes)</td>
<td>Length (bytes)</td>
</tr>
<tr>
<td><strong>Version</strong></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>1</td>
<td>Not present</td>
</tr>
<tr>
<td><strong>Algorithm ID</strong></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Message ID</strong></td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td><strong>AAD Length</strong></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>When the <em>encryption context</em> is empty, the value of the 2-byte AAD Length field is 0.</td>
<td>When the <em>encryption context</em> is empty, the value of the 2-byte AAD Length field is 0.</td>
</tr>
<tr>
<td><strong>AAD</strong></td>
<td>Variable. The length of this field appears in the previous 2 bytes (AAD Length field).</td>
<td>Variable. The length of this field appears in the previous 2 bytes (AAD Length field).</td>
</tr>
<tr>
<td></td>
<td>When the <em>encryption context</em> is empty, there is no AAD field in the header.</td>
<td>When the <em>encryption context</em> is empty, there is no AAD field in the header.</td>
</tr>
<tr>
<td><strong>Encrypted Data Key Count</strong></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Encrypted Data Key(s)</strong></td>
<td>Variable. Determined by the number of encrypted data keys and the length of each.</td>
<td>Variable. Determined by the number of encrypted data keys and the length of each.</td>
</tr>
<tr>
<td><strong>Content Type</strong></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>4</td>
<td>Not present</td>
</tr>
<tr>
<td><strong>IV Length</strong></td>
<td>1</td>
<td>Not present</td>
</tr>
<tr>
<td><strong>Frame Length</strong></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Field</td>
<td>Message format version 1</td>
<td>Message format version 2</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Algorithm Suite Data</td>
<td>Not present</td>
<td>Variable. Determined by the algorithm that generated the message.</td>
</tr>
<tr>
<td>Header Authentication</td>
<td>Variable. Determined by the algorithm that generated the message.</td>
<td>Variable. Determined by the algorithm that generated the message.</td>
</tr>
</tbody>
</table>

**Version**

The version of this message format. The version is either 1 or 2 encoded as the byte 01 or 02 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.

This field is not present in message format version 2.

**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.

**Message ID**

A randomly generated 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.

This value is 128 bits in message format version 1 and 256 bits in version 2.
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 is empty, the value of the AAD Length field is 0.

AAD

The additional authenticated data. The AAD is an encoding of the encryption context, an array of key-value pairs where each key and value is a string of UTF-8 encoded characters. The encryption context is converted to a sequence of bytes and used for the AAD value. When the encryption context is empty, there is no AAD field in the header.

When the algorithms with signing 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</td>
<td>2</td>
</tr>
<tr>
<td>Key Length</td>
<td>2</td>
</tr>
<tr>
<td>Key</td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Key Length).</td>
</tr>
<tr>
<td>Value Length</td>
<td>2</td>
</tr>
<tr>
<td>Value</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. The maximum number of encrypted data keys in each message is $65,535 (2^{16} - 1)$.

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 (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Provider ID Length</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Key Provider ID</strong></td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Key Provider ID Length).</td>
</tr>
<tr>
<td><strong>Key Provider Information Length</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Key Provider Information</strong></td>
<td>Variable. Equal to the value specified in the previous 2 bytes (Key Provider Information Length).</td>
</tr>
<tr>
<td><strong>Encrypted Data Key Length</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Encrypted Data Key</strong></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 key.
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 data, either nonframed or framed.

Note

Whenever possible, use framed data. The AWS Encryption SDK supports nonframed data only for legacy use. Some language implementations of the AWS Encryption SDK can still generate nonframed ciphertext. All supported language implementations can decrypt framed and nonframed ciphertext.

Framed data is divided into equal-length parts; each part is encrypted separately. Framed content is type 2, encoded as the byte 02 in hexadecimal notation.

Nonframed data is not divided; it is a single encrypted blob. Non-framed content is type 1, encoded as the byte 01 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).

This field is not present in message format version 2.

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 that generated the message.

This field is not present in message format version 2, which only supports algorithm suites that use deterministic IV values in the message header.
Frame Length

The length of each frame of framed data. It is a 4-byte value interpreted as a 32-bit unsigned integer that specifies the number of bytes in each frame. When the data is nonframed, that is, when the value of the `Content Type` field is 1, this value must be 0.

**Note**

Whenever possible, use framed data. The AWS Encryption SDK supports nonframed data only for legacy use. Some language implementations of the AWS Encryption SDK can still generate nonframed ciphertext. All supported language implementations can decrypt framed and nonframed ciphertext.

Algorithm Suite Data

Supplementary data needed by the `algorithm` that generated the message. The length and contents are determined by the algorithm. Its length might be 0.

This field is not present in message format version 1.

Header Authentication

The header authentication is determined by the `algorithm` 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 version 1.0 (bytes)</th>
<th>Length in version 2.0 (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IV</strong></td>
<td>Variable. Determined by the IV bytes value of the <code>algorithm</code> that generated the message.</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Authentication Tag</strong></td>
<td>Variable. Determined by the authentication tag bytes value of the <code>algorithm</code> that generated the message.</td>
<td>Variable. Determined by the authentication tag bytes value of the <code>algorithm</code> that generated the message.</td>
</tr>
</tbody>
</table>
IV

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

This field is not present in the header of message format version 2. Message format version 2 only supports algorithm suites that use deterministic IV values in the message header.

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. The message body structure is the same in message format versions 1 and 2.

Topics

• Non-framed data
• Framed data

Non-framed data

Non-framed data is encrypted in a single blob with a unique IV and body AAD.

Note

Whenever possible, use framed data. The AWS Encryption SDK supports nonframed data only for legacy use. Some language implementations of the AWS Encryption SDK can still generate nonframed ciphertext. All supported language implementations can decrypt framed and nonframed ciphertext.

The following table describes the fields that form nonframed data. The bytes are appended in the order shown.
Non-Framed Body Structure

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

**IV**

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

**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.

**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.

**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.

**Note**

Whenever possible, use framed data. The AWS Encryption SDK supports nonframed data only for legacy use. Some language implementations of the AWS Encryption SDK can still generate nonframed ciphertext. All supported language implementations can decrypt framed and nonframed ciphertext.

The *frame length*, which is the length of the *encrypted content* 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><strong>Sequence Number</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>IV</strong></td>
<td>Variable. Equal to the value specified in the IV Length byte of the header.</td>
</tr>
<tr>
<td><strong>Encrypted Content</strong></td>
<td>Variable. Equal to the value specified in the Frame Length of the header.</td>
</tr>
<tr>
<td><strong>Authentication Tag</strong></td>
<td>Variable. Determined by the algorithm used, as specified in the Algorithm ID 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 used.

**Encrypted Content**

The encrypted content (ciphertext) for the frame, as returned by the encryption algorithm.
**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</td>
<td>4</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>4</td>
</tr>
<tr>
<td><strong>IV</strong></td>
<td>Variable. Equal to the value specified in the <strong>IV</strong> Length byte of the header.</td>
</tr>
<tr>
<td>Encrypted Content Length</td>
<td>4</td>
</tr>
<tr>
<td>Encrypted Content</td>
<td>Variable. Equal to the value specified in the previous 4 bytes (Encrypted Content Length).</td>
</tr>
<tr>
<td>Authentication Tag</td>
<td>Variable. Determined by the algorithm used, as specified in the <strong>Algorithm ID</strong> of the header.</td>
</tr>
</tbody>
</table>

**Sequence Number End**

An indicator for the final frame. The value is encoded as the 4 bytes FF FF FF FF in hexadecimal notation.

**Sequence Number**

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

Framed data must start at sequence number 1. Subsequent frames must be in order and must contain an increment of 1 of the previous frame. Otherwise, the decryption process stops and reports an error.

**IV**

The initialization vector (IV) for the frame. The SDK uses a deterministic method to construct a different IV for each frame in the message. The length of the IV length is specified by the **Algorithm suite**.
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.

Authentication Tag

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

Footer structure

When the algorithms with signing are used, the message format contains a footer. The message footer contains a digital 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. The message footer structure is the same in message format versions 1 and 2.

Footer Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature Length</td>
<td>2</td>
</tr>
<tr>
<td>Signature</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.
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.

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification 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

- Framed data (message format version 1)
- Framed data (message format version 2)
- Non-framed data (message format version 1)

Framed data (message format version 1)

The following example shows the message format for framed data in message format version 1.

```
+--------+
| Header |
+--------+
01       Version (1.0)
80       Type (128, customer authenticated encrypted data)
0378     Algorithm ID (see Algorithms reference)
6E7C0FBD 4DF4A999 717C22A2 DDFE1A27 Message ID (random 128-bit value)
008E     AAD Length (142)
0004     AAD Key-Value Pair Count (4)
0005     AAD Key-Value Pair 1, Key Length (5)
30746869 73 AAD Key-Value Pair 1, Key ("0This")
0002     AAD Key-Value Pair 1, Value Length (2)
```
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6973</td>
<td>AAD Key-Value Pair 1, Value (&quot;is&quot;)</td>
</tr>
<tr>
<td>0003</td>
<td>AAD Key-Value Pair 2, Key Length (3)</td>
</tr>
<tr>
<td>31616E</td>
<td>AAD Key-Value Pair 2, Key (&quot;1an&quot;)</td>
</tr>
<tr>
<td>000A</td>
<td>AAD Key-Value Pair 2, Value Length (10)</td>
</tr>
<tr>
<td>656E6372 79774690 6F6E</td>
<td>AAD Key-Value Pair 2, Value (&quot;encryption&quot;)</td>
</tr>
<tr>
<td>0008</td>
<td>AAD Key-Value Pair 3, Key Length (8)</td>
</tr>
<tr>
<td>32636F6E 74657874</td>
<td>AAD Key-Value Pair 3, Key (&quot;2context&quot;)</td>
</tr>
<tr>
<td>0007</td>
<td>AAD Key-Value Pair 3, Value Length (7)</td>
</tr>
<tr>
<td>657B616D 706C65</td>
<td>AAD Key-Value Pair 3, Value (&quot;example&quot;)</td>
</tr>
<tr>
<td>0015</td>
<td>AAD Key-Value Pair 4, Key Length (21)</td>
</tr>
<tr>
<td>6177732D 63727970 746F2D70 75626C69</td>
<td>AAD Key-Value Pair 4, Key (&quot;aws-crypto-public-key&quot;)</td>
</tr>
<tr>
<td>632D6B65 79</td>
<td>AAD Key-Value Pair 4, Value Length (68)</td>
</tr>
<tr>
<td>0044</td>
<td>AAD Key-Value Pair 4, Value (&quot;AjAsui2ot06LKwZXDJnU/Aqc2vD+00kp0Z1cc8Tg2qd7rs5aLTg71vfUEW/86+/5w==&quot;)</td>
</tr>
<tr>
<td>5844A6E 552F4171 63327644 2B304F6B</td>
<td>EncryptedDataKeyCount (2)</td>
</tr>
<tr>
<td>704F5A31 63633854 67327164 37727335</td>
<td>Encrypted Data Key 1, Key Provider ID Length (7)</td>
</tr>
<tr>
<td>614C5467 376C7666 5545572F 38562B2F</td>
<td>Encrypted Data Key 1, Key Provider Information Length (75)</td>
</tr>
<tr>
<td>35773D3D</td>
<td>Encrypted Data Key 1, Key Provider Information (&quot;arn:aws:kms:us-west-2:11112223333:key/715c0818-5825-4245-a755-138a6d9a11e6&quot;)</td>
</tr>
<tr>
<td>6573742D 323A3131 31313232 32323333</td>
<td>Encrypted Data Key 1, Key Provider Information Length (167)</td>
</tr>
<tr>
<td>33333A6B 65792F37 313556330 3831382D</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>35383235 2D343234 352D6137 35352D31</td>
<td>Encrypted Data Key 1, Encrypted Data Key Length (167)</td>
</tr>
<tr>
<td>333B6136 64396131 316536</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>004B</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>617732D</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>6B6D73</td>
<td>Encrypted Data Key 1, Key Provider Information Length (75)</td>
</tr>
<tr>
<td>0002</td>
<td>Encrypted Data Key 1, Key Provider Information (&quot;aws-kms&quot;)</td>
</tr>
<tr>
<td>0007</td>
<td>Encrypted Data Key 1, Key Provider Information Length (75)</td>
</tr>
<tr>
<td>0015</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>004B</td>
<td>Encrypted Data Key 1, Encrypted Data Key Length (167)</td>
</tr>
<tr>
<td>01010200 7857A1C1 F7370545 4ECA7C83</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>956C4702 23DCE8D7 16C59679 973E3CED</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>02AAEF29 7F000000 7E307C06 092A8648</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>86F70D01 0706A06F 306D0201 00306806</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>092A8648 86F70D01 0701301E 06096086</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>4B016503 04012E30 11040C3F F0C897B</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>7A12EB19 8BF2D802 0110803B 24003D1F</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
<tr>
<td>A5474FBC 392360B5 CB9997E0 6A17DE4C</td>
<td>Encrypted Data Key 1, Encrypted Data Key</td>
</tr>
</tbody>
</table>

Framed data (message format version 1)
| AESD67332 6BF86DAB 60D8CCB8 8295DBE9 |
| 4707E356 ADA3735A 7C52D77B B3135A47 |
| 9F224BF9 E67E87 |
| 0007 | Encrypted Data Key 2, Key Provider ID Length |
| (7) | |
| 6177732D 6B6D73 | Encrypted Data Key 2, Key Provider ID ("aws-kms") |
| 004E | Encrypted Data Key 2, Key Provider Information Length (78) |
| 61726E3A 6177733A 6B6D733A 63612D63 | Encrypted Data Key 2, Key Provider Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-be3435b423ff") |
| 656E7472 616C2D31 3A313313 31323233 333A6B65 792F3962 31333631 34622D61 6663632D 34366138 2D616134 372D6265 33343335 62343233 6666 | |
| 00A7 | Encrypted Data Key 2, Encrypted Data Key Length (167) |
| 01010200 78FAFFFB D6DE06AF AC72F79B | Encrypted Data Key 2, Encrypted Data Key |
| 0E57BD87 3F60F4E6 FD196144 5A002C94 | |
| 0B896144 0CA27950 CA571201 4A58029 | Content Type (2, framed data) |
| 02 | Reserved |
| 0C | IV Length (12) |
| 2 | Frame Length (256) |
| 4ECBD5C0 9899CA65 923D2347 | IV |
| 0B896144 0CA27950 CA571201 4A58029 | Authentication Tag |
| +--------+ | Body | |
| +--------+ | |
| 00000001 | Frame 1, Sequence Number (1) |
| 6BD3FE9C ADBCB213 5B89E8F1 | Frame 1, IV |
| 1F6471E0 A51AF310 10FA9EF6 F0C76EDF | Frame 1, Encrypted Content |
| F5AFA33C 7D2E8C6C 9C5D5175 A212AF8E | |
| FBD9A0C3 C6E3FB59 C125DBF2 89AC7939 | |
| BDEEE34A8 0F00F49E ACBBD8B2 1C785089 | |
Frame 1, Authentication Tag
Frame 2, Sequence Number (2)
Frame 2, IV
Frame 2, Encrypted Content
Frame 2, Authentication Tag
Final Frame, Sequence Number End
Final Frame, Sequence Number (3)
Final Frame, IV
Final Frame, Encrypted Content Length (142)
Final Frame, Encrypted Content
Framed data (message format version 2)

The following example shows the message format for framed data in message format version 2.
AAD Key-Value Pair 4, Value Length (68)
"QXRnM3JwOEVBNfNhNmaTk3MULTTk3NhM0M1LZE5vSmtwRHFPC0dIYkVaVDRqME5OM1FRStmbTFVY01WdThnPT0=
39373149 53353937 347a4e32 7959584e
6f4a6b70 44714f73 47486245 5a54346a
304e4e32 5164452b 666d3155 634d5675
38673d3d

Encrypted Data Key Count (1)

Encrypted Data Key 1, Key Provider ID Length (7)

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

Encrypted Data Key 1, Key Provider Information ("arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f")

Encrypted Data Key 1, Key Provider Information ("arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f")

Encrypted Data Key 1, Encrypted Data Key

Content Type (2, framed data)

Frame Length (4096)

Algorithm Suite Data (key commitment)

Authentication Tag

Final Frame, Sequence Number End

Final Frame, Sequence Number (1)
Non-framed data (message format version 1)

The following example shows the message format for nonframed data.

⚠️ Note

Whenever possible, use framed data. The AWS Encryption SDK supports nonframed data only for legacy use. Some language implementations of the AWS Encryption SDK can still generate nonframed ciphertext. All supported language implementations can decrypt framed and nonframed ciphertext.
Non-framed data (message format version 1)
| 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) | 004E |
| Encrypted Data Key 2, Key Provider Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-be3435b423ff") | 61726E3A 6177733A 6B6D733A 333A6B65 792F3962 31336361 34622D61 6663632D 34366138 2D616134 372D6265 33343335 62343323 6666 |
| Encrypted Data Key 2, Encrypted Data Key Length (167) | 00A7 |
| Encrypted Data Key 2, Encrypted Data Key | 01010200 78FAFFFB D6DE06AF AC72F79B 0E57BD87 3F60F4E6 FD196144 5A002DC9 AF787150 69000000 7E307C06 092A8648 86F70D01 0706A06F 306D0201 00306806 092A8648 86F70D01 0701301E 06096086 48016503 04012E30 11040CB2 A82000CC 76616EF2 A6B30D02 0110803B 8073D0F1 FDD01BD9 B0979082 099FDBFC F7B13548 3CC686D7 F3CF7C7A CCC52639 122A1495 71F18A46 80E2C43F A34C0E58 11D05114 2A363C2A E11397 |
| Content Type (1, nonframed data) | 01 |
| Reserved | 00000000 |
| IV Length (12) | 0C |
| Frame Length (0, nonframed data) | 00000000 |
| IV | 734C1BBE 032F7025 84CDA9D0 2C82BB23 4CBF4AAB 8F5C6002 622E886C |
| Authentication Tag | +------|
| Body | +------|
| Encrypted Data Key 2, Encrypted Data Key | D39DD3E5 915E0201 77A4AB11 |
| Encrypted Content Length (654) | 00000000 0000028E |
| Encrypted Content | E8B6F955 B5F22FE4 FD890224 4E1D5155 5871BA4C 93F78436 1085E4F8 D61ECE28 59455BD8 D76479DF C28D2E0B BDB3D5D3 E4159DFE C8A944B6 685643FC EA24122B |
Non-framed data (message format version 1)

Authentication Tag

Signature Length (103)

Signature
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.

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in GitHub.

You must provide additional authenticated data (AAD) to the AES-GCM algorithm for each cryptographic operation. This is true for both framed and nonframed body data. 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.

**Body AAD Structure**

<table>
<thead>
<tr>
<th>Field</th>
<th>Length, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message ID</td>
<td>16</td>
</tr>
<tr>
<td>Body AAD Content</td>
<td>Variable. See Body AAD Content in the following list.</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>4</td>
</tr>
<tr>
<td>Field</td>
<td>Length, in bytes</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Content Length</td>
<td>8</td>
</tr>
</tbody>
</table>

**Message ID**

The same *Message ID* value set in the message header.

**Body AAD Content**

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

For *nonframed data*, use the value AWSKMSEncryptionClient Single Block.

For regular frames in *framed data*, use the value AWSKMSEncryptionClient Frame.

For the final frame in *framed data*, use the value AWSKMSEncryptionClient Final Frame.

**Sequence Number**

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

For *framed data*, this is the frame sequence number.

For *nonframed data*, use the value 1, encoded as the 4 bytes `00 00 00 01` in hexadecimal notation.

**Content Length**

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

**AWS Encryption SDK 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](#).
For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in GitHub.

If you are building your own library that can read and write ciphertexts that are compatible with the AWS Encryption SDK, you'll need to understand how the AWS Encryption SDK implements the supported algorithm suites to encrypt raw data.

The AWS Encryption SDK supports the following algorithm suites. All AES-GCM algorithm suites have a 12-byte initialization vector and a 16-byte AES-GCM authentication tag. The default algorithm suite varies with the AWS Encryption SDK version and the selected key commitment policy. For details, see Commitment policy and algorithm suite.

### AWS Encryption SDK Algorithm Suites

<table>
<thead>
<tr>
<th>Algorithm ID</th>
<th>Message format version</th>
<th>Encryption algorithm</th>
<th>Data key length (bits)</th>
<th>Key derivation algorithm</th>
<th>Signature algorithm</th>
<th>Key commitment algorithm</th>
<th>Algorithm suite data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 78</td>
<td>0x02</td>
<td>AES-GCM</td>
<td>256</td>
<td>HKDF with SHA-512</td>
<td>ECDSA with P-384 and SHA-384</td>
<td>HKDF with SHA-512</td>
<td>32 (key commitment)</td>
</tr>
<tr>
<td>04 78</td>
<td>0x02</td>
<td>AES-GCM</td>
<td>256</td>
<td>HKDF with SHA-512</td>
<td>None</td>
<td>HKDF with SHA-512</td>
<td>32 (key commitment)</td>
</tr>
<tr>
<td>03 78</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>256</td>
<td>HKDF with SHA-384</td>
<td>ECDSA with P-384 and SHA-384</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>Algorithm ID</td>
<td>Message format version</td>
<td>Encryption algorithm</td>
<td>Data key length (bits)</td>
<td>Key derivation algorithm</td>
<td>Signature algorithm</td>
<td>Key commitment algorithm</td>
<td>Algorithm suite data length (bytes)</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
<td>----------------------</td>
<td>------------------------</td>
<td>--------------------------</td>
<td>----------------------</td>
<td>--------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>03 46</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>192</td>
<td>HKDF with SHA-384</td>
<td>ECDSA with P-384 and SHA-384</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>02 14</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>128</td>
<td>HKDF with SHA-256</td>
<td>ECDSA with P-256 and SHA-256</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>01 78</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>256</td>
<td>HKDF with SHA-256</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>01 46</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>192</td>
<td>HKDF with SHA-256</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>01 14</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>128</td>
<td>HKDF with SHA-256</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>00 78</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>256</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>00 46</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>192</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
</tr>
<tr>
<td>00 14</td>
<td>0x01</td>
<td>AES-GCM</td>
<td>128</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Algorithm ID

A 2-byte hexadecimal value that uniquely identifies an algorithm implementation. This value is stored in the message header of the ciphertext.

Message format version

The version of the message format. Algorithm suites with key commitment use message format version 2 (0x02). Algorithm suites without key commitment use message format version 1 (0x01).

Algorithm suite data length

The length in bytes of data specific to the algorithm suite. This field is supported only in message format version 2 (0x02). In message format version 2 (0x02), this data appears in the Algorithm suite data field of the message header. Algorithm suites that support key commitment use 32 bytes for the key commitment string. For more information, see Key commitment algorithm in this list.

Data key length

The length of the data key in bits. The AWS Encryption SDK supports 256-bit, 192-bit, and 128-bit keys. The data key is generated by a keyring or master key.

In 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.

Encryption algorithm

The name and mode of the encryption algorithm used. Algorithm suites in the AWS Encryption SDK use the Advanced Encryption Standard (AES) encryption algorithm with Galois/Counter Mode (GCM).

Key commitment algorithm

The algorithm used to calculate the key commitment string. The output is stored in the Algorithm suite data field of the message header and is used to validate the data key for key commitment.

For a technical explanation of adding key commitment to an algorithm suite, see Key Committing AEADS in Cryptology ePrint Archive.
Key derivation algorithm

The HMAC-based extract-and-expand key derivation function (HKDF) used to derive the data encryption key. The AWS Encryption SDK uses the HKDF defined in RFC 5869.

Algorithm suites without key commitment (algorithm ID 01xx – 03xx)

- The hash function used is either SHA-384 or SHA-256, depending on the algorithm suite.
- For the extract step:
  - No salt is used. Per the RFC, the salt is set to a string of zeros. The string length is equal to the length of the hash function output, which is 48 bytes for SHA-384 and 32 bytes for SHA-256.
  - The input keying material is the data key 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 and message ID (in that order).
  - The length of the output keying material is the Data key length. This output is used as the data encryption key in the encryption algorithm.

Algorithm suites with key commitment (algorithm ID 04xx and 05xx)

- The hash function used is SHA-512.
- For the extract step:
  - The salt is a 256-bit cryptographic random value. In message format version 2 (0x02), this value is stored in the MessageID field.
  - The initial keying material is the data key from the keyring or master key provider.
- For the expand step:
  - The input pseudorandom key is the output from the extract step.
  - The key label is the UTF-8-encoded bytes of the DERIVEKEY string in big endian byte order.
  - The input info is a concatenation of the algorithm ID and the key label (in that order).
  - The length of the output keying material is the Data key length. This output is used as the data encryption key in the encryption algorithm.

Message format version

The version of the message format used with the algorithm suite. For details, see Message format reference.
Signature algorithm

The signature algorithm that is used to generate a digital signature over the ciphertext header and body. The AWS Encryption SDK uses the Elliptic Curve Digital Signature Algorithm (ECDSA) with the following specifics:

- The elliptic curve used is either the P-384 or P-256 curve, as specified by the algorithm ID. These curves are defined in Digital Signature Standard (DSS) (FIPS PUB 186-4).
- The hash function used is SHA-384 (with the P-384 curve) or SHA-256 (with the P-256 curve).

AWS Encryption SDK initialization vector reference

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

To use the AWS Encryption SDK in one of the supported programming languages, see Programming languages.

For the specification that defines the elements of a proper AWS Encryption SDK implementation, see the AWS Encryption SDK Specification in GitHub.

The AWS Encryption SDK supplies the initialization vectors (IVs) that are required by all supported algorithm suites. 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, 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.

**AWS KMS Hierarchical keyring technical details**

The AWS KMS Hierarchical keyring uses a unique data key to encrypt each field and encrypts each data key with a unique wrapping key derived from an active branch key. It uses a key derivation in counter mode with a pseudorandom function with HMAC SHA-256 to derive the 32 byte wrapping key with the following inputs.

- A 16 byte random salt
- The active branch key
- The UTF-8 encoded value for the key provider identifier "aws-kms-hierarchy"

The Hierarchical keyring uses the derived wrapping key to encrypt a copy of the plaintext data key using AES-GCM-256 with a 16 byte authentication tag and the following inputs.

- The derived wrapping key is used as the AES-GCM cipher key
- The data key is used as the AES-GCM message
- A 12 byte random initialization vector (IV) is used as the AES-GCM IV
- Additional authenticated data (AAD) containing the following serialized values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Length in bytes</th>
<th>Interpreted as</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;aws-kms-hierarchy&quot;</td>
<td>17</td>
<td>UTF-8 encoded</td>
</tr>
<tr>
<td>The branch key identifier</td>
<td>Variable</td>
<td>UTF-8 encoded</td>
</tr>
<tr>
<td>The branch key version</td>
<td>16</td>
<td>UTF-8 encoded</td>
</tr>
<tr>
<td>Encryption context</td>
<td>Variable</td>
<td>UTF-8 encoded key value pairs</td>
</tr>
</tbody>
</table>
Document history for the AWS Encryption SDK Developer Guide

This topic describes significant updates to the *AWS Encryption SDK Developer Guide*.

Topics

- Recent updates
- Earlier updates

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, subscribe to the RSS feed.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AWS Encryption SDK for Java version 3.x</strong></td>
<td>Integrates the AWS Encryption SDK for Java with the material providers library. Adds support for keyrings and the required encryption context CMM.</td>
<td>December 6, 2023</td>
</tr>
<tr>
<td><strong>AWS Encryption SDK for .NET version 4.x</strong></td>
<td>Adds support for the AWS KMS Hierarchical keyring, the required encryption context CMM, and asymmetric RSA AWS KMS keyrings.</td>
<td>October 12, 2023</td>
</tr>
<tr>
<td>General availability</td>
<td>Introducing support for the AWS Encryption SDK for .NET.</td>
<td>May 17, 2022</td>
</tr>
<tr>
<td>Documentation change</td>
<td>Replace the AWS Key Management Service term</td>
<td>August 30, 2021</td>
</tr>
</tbody>
</table>
**customer master key (CMK)** with **AWS KMS key** and **KMS key**.

### General availability

**Added support for AWS Key Management Service. (AWS KMS) multi-Region keys.**
Multi-Region keys are AWS KMS keys in different AWS Regions that can be used interchangeably because they have the same key ID and key material.

**June 8, 2021**

### General availability

**Added and updated documentation about the improved message decryption process.**

**May 11, 2021**

### General availability

**Added and updated documentation for the general availability release of AWS Encryption CLI version 1.8.x to replace AWS Encryption CLI version 1.7.x, and AWS Encryption CLI 2.1.x to replace AWS Encryption CLI 2.0.x.**

**October 27, 2020**
<table>
<thead>
<tr>
<th>Release Type</th>
<th>Description</th>
<th>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 versions 1.7.x and 2.0.x, including a <a href="#">best practices guide</a>, a <a href="#">migration guide</a>, updated <a href="#">concepts</a>, updated <a href="#">programming language topics</a>, an updated <a href="#">algorithm suites reference</a>, an updated <a href="#">message format reference</a>, and a new <a href="#">message format example</a>.</td>
<td>September 24, 2020</td>
</tr>
<tr>
<td>General availability</td>
<td>Added and updated documentation for the general availability release of the <a href="#">AWS Encryption SDK for JavaScript</a>.</td>
<td>October 1, 2019</td>
</tr>
<tr>
<td>Preview release</td>
<td>Added and updated documentation of the public beta release of the <a href="#">AWS Encryption SDK for JavaScript</a>.</td>
<td>June 21, 2019</td>
</tr>
<tr>
<td>General availability</td>
<td>Added and updated documentation for the general availability release of the <a href="#">AWS Encryption SDK for C</a>.</td>
<td>May 16, 2019</td>
</tr>
<tr>
<td>Preview release</td>
<td>Added documentation of the preview release of the <a href="#">AWS Encryption SDK for C</a>.</td>
<td>February 5, 2019</td>
</tr>
</tbody>
</table>
New release

Added documentation of the command line interface for the AWS Encryption SDK.

November 20, 2017

Earlier updates

The following table describes significant changes to the AWS Encryption SDK Developer Guide before November 2017.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>New release</td>
<td>Added the Data key caching chapter for the new feature.</td>
<td>July 31, 2017</td>
</tr>
<tr>
<td></td>
<td>Added the section called “Initialization vector reference” 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 section called “Concepts” topic to explain concepts, including the new cryptographic materials manager.</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Expanded the Message format reference documentation into a new AWS Encryption SDK reference section.</td>
<td>March 21, 2017</td>
</tr>
<tr>
<td></td>
<td>Added a section about the AWS Encryption SDK Supported algorithm suites.</td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>New release</td>
<td>The AWS Encryption SDK now supports the <a href="https://www.python.org">Python</a> programming language, in addition to <a href="https://www.oracle.com/java/">Java</a>.</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>