AWS Encryption SDK

Developer Guide

AWS Encryption SDK: Developer Guide

Copyright © 2017 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
How the SDK Works 2
Symmetric Key Encryption 2
Envelope Encryption 3
Encryption Workflows 4
Getting Started 6
Supported Algorithms 7
Programming Languages 8
Java 8
Prerequisites 8
Installation 9
Example Code 9
Python 16
Prerequisites 16
Installation 16
Example Code 16
Frequently Asked Questions 23
Reference 26
Message Format Reference 26
Header Structure 27
Body Structure 31
Footer Structure 33
Algorithms Reference 34
Body AAD Reference 36
Message Format Examples 37
Non-Framed Data 37
Framed Data 39
Document History 43

What Is the AWS Encryption SDK?

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

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

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

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

A Default Implementation that Adheres to Cryptography Best Practices

The AWS Encryption SDK generates a unique data encryption key (DEK) for each data object it encrypts. This follows the cryptography best practice of using unique DEKs for each encryption operation.

The SDK encrypts your data using a secure, authenticated, symmetric key algorithm. For more information, see Supported Algorithms (p. 7).

A Data Format that Stores Encrypted DEKs with the Corresponding Encrypted Data

The AWS Encryption SDK uses a defined data format (p. 26) to store the encrypted DEKs and encrypted data together as one object. This means you don't need to keep track of or protect the DEKs that encrypt your data because the SDK does it for you.

A Framework for Protecting DEKs with Master Keys

The AWS Encryption SDK protects the DEKs that encrypt your data by encrypting them with one or more master keys. By providing a framework to encrypt DEKs with more than one master key, the SDK helps make your encrypted data portable. For example, you can encrypt data under multiple customer master keys (CMKs) in AWS Key Management Service (AWS KMS), each in a different AWS Region. Then you can copy the encrypted data to any of the regions and decrypt it without a dependency on the others. You can also encrypt data under a CMK in AWS KMS and a master key in an on-premises HSM, enabling you to later decrypt the data even if one master key is unavailable.

With the AWS Encryption SDK, you define a *master key provider*, which represents one or more *master keys*. Then you encrypt and decrypt your data using straightforward methods provided by the SDK. The SDK does the rest.

For more information about how this SDK works, see How the SDK Works (p. 2).

To get started, see Getting Started (p. 6).

The SDK is provided for free under the Apache license.

How the AWS Encryption SDK Works

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

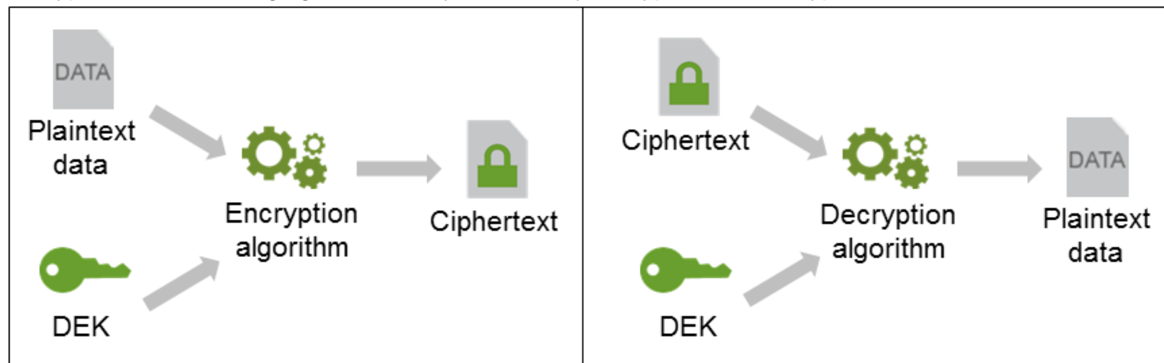
Topics

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

Symmetric Key Encryption

To encrypt data, the AWS Encryption SDK provides the raw data, known as *plaintext*, and a data encryption key (DEK) to an encryption algorithm. The algorithm uses those inputs to produce encrypted data, known as *ciphertext*. To decrypt data, the SDK provides ciphertext and the DEK to a decryption algorithm that uses those inputs to return the plaintext data.

When the same DEK is used to both encrypt and decrypt data, it is known as *symmetric key* encryption and decryption. The following figure shows symmetric key encryption and decryption.



Envelope Encryption

The security of your encrypted data depends on protecting the data encryption key (DEK) that can decrypt it. One accepted best practice for protecting the DEK is to encrypt it. To do this, you need another encryption key, known as a *master key*. This practice of using a master key to encrypt DEKs is known as *envelope encryption*. Some of the benefits of envelope encryption include the following.

Protecting DEKs

When you encrypt a DEK, you don't have to worry about where to store it because the DEK is inherently protected by encryption. You can safely store the encrypted DEK with the encrypted data. The AWS Encryption SDK does this for you; it combines the encrypted DEK and the encrypted data into one object.

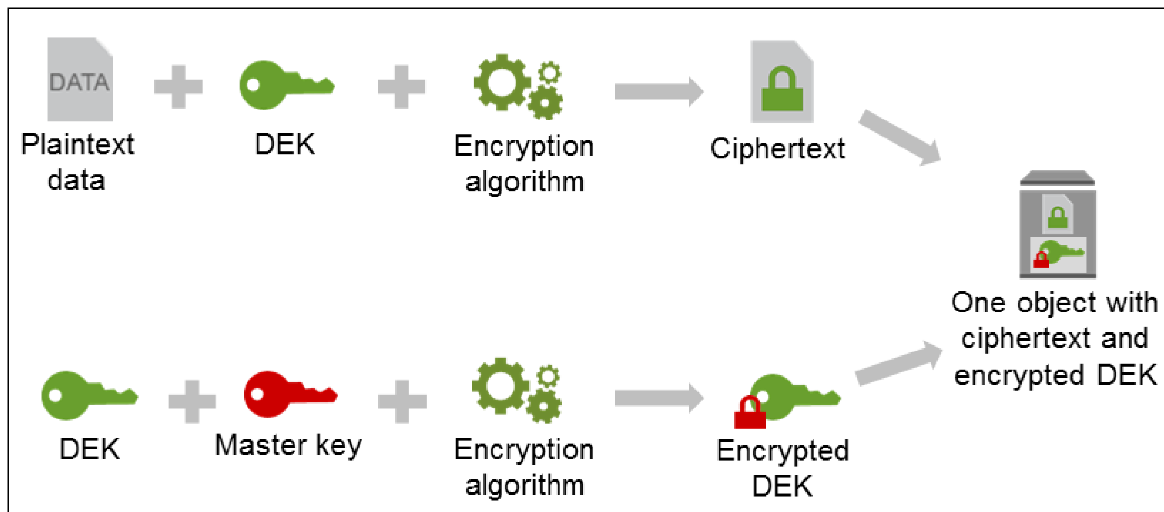
Encrypting the Same Data Under Multiple Master Keys

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

Combining the Strengths of Multiple Algorithms

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

The AWS Encryption SDK uses envelope encryption. It encrypts each of your DEKs with a master key. The master key is an unencrypted (plaintext) key that can decrypt one or more DEKs. The following figure shows how this works.



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

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

If you don't have a key management system, we recommend AWS KMS. The AWS Encryption SDK integrates with AWS KMS to help you protect and use your master keys. You can also use the SDK with

other master key providers, including custom ones that you define. Even if you don't use AWS, you can still use this SDK.

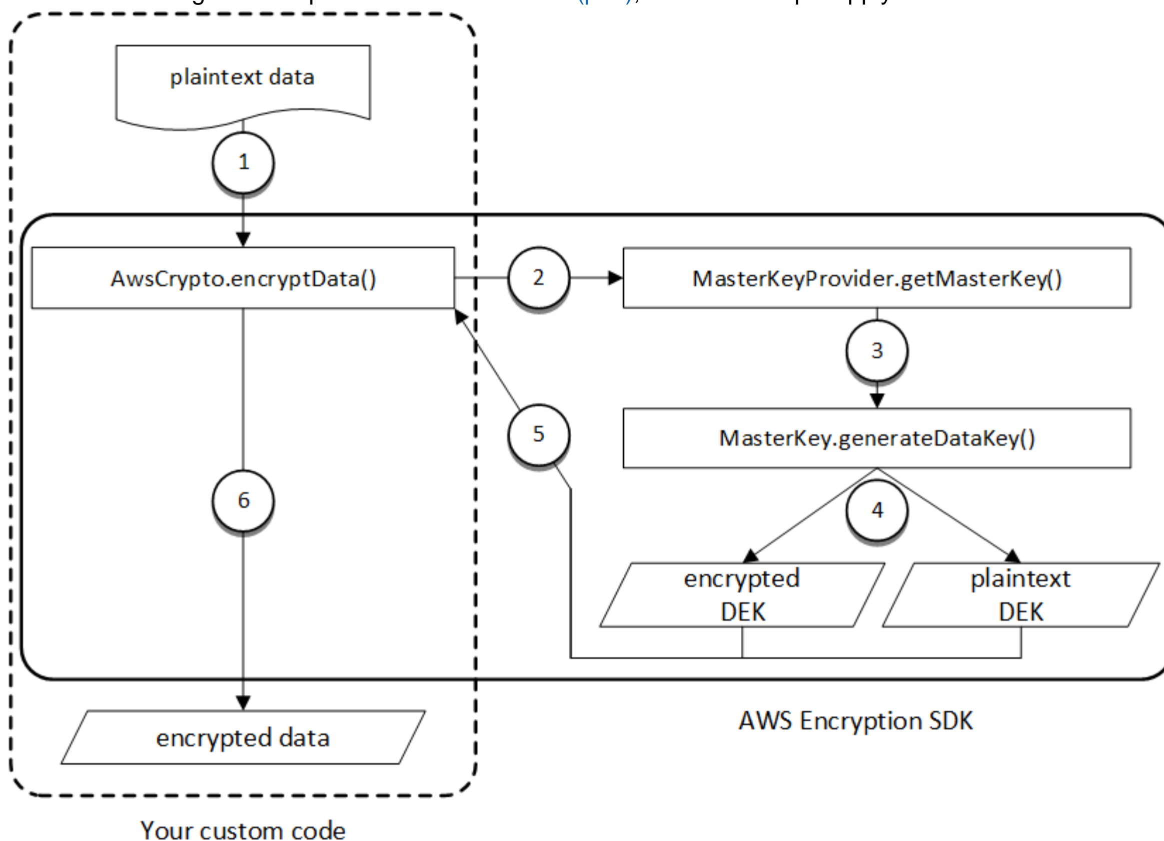
Encryption Workflows

After you define a master key provider, you can use the AWS Encryption SDK to encrypt and decrypt data. The SDK provides methods that operate on strings, byte arrays, and byte streams. The following diagrams show a high-level overview of how these methods work.

For code examples in the supported programming languages, see Programming Languages (p. 8).

Encryption

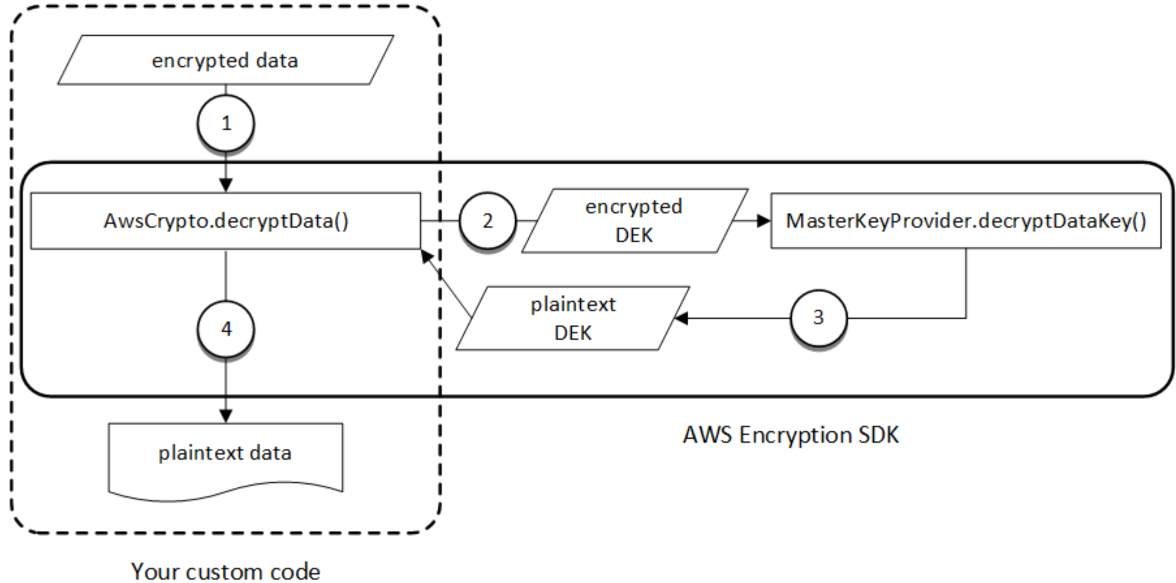
The following diagram shows an overview of how the AWS Encryption SDK encrypts data. The method names in this diagram are specific to the Java client (p. 8), but the concepts apply to all clients.



1. Your application passes data to one of the encryption methods.
2. The encryption method uses a master key provider to determine which master key to use.
3. The master key generates a DEK.
4. The master key returns two copies of the DEK, one in plaintext and one encrypted by the master key.
5. The encryption method uses the plaintext DEK to encrypt the data, then deletes the plaintext DEK.
6. The encryption method returns a single object (p. 26) that contains the encrypted data and the encrypted DEK.

Decryption

The following diagram shows an overview of how the AWS Encryption SDK decrypts data. The method names in this diagram are specific to the Java client (p. 8), but the concepts apply to all clients.



1. Your application passes encrypted data to one of the decryption methods.
2. The decryption method extracts the encrypted DEK from the encrypted data, then sends the encrypted DEK to a master key provider for decryption.
3. The master key provider decrypts the encrypted DEK, then returns the plaintext DEK to the decryption method.
4. The decryption method uses the plaintext DEK to decrypt the data, then deletes the plaintext DEK. The decryption method returns the plaintext data.

Getting Started with the AWS Encryption SDK

To use the AWS Encryption SDK, you need a master key provider. If you don't have one, we recommend using AWS Key Management Service (AWS KMS). The SDK provides built-in support for master keys stored in AWS KMS, and some of the example code in the SDK requires a customer master key (CMK) 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 CMK in AWS KMS. To learn how, see Creating Keys in the *AWS Key Management Service Developer Guide*.
3. Create an IAM user with an access key. To learn how, see Creating IAM Users in the *IAM User Guide*. When you create the user, for **Access type**, choose **Programmatic access**. After you create the user, choose **Download .csv** to save the access key. Store the file in a secure location.

   An access key consists of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. We recommend that you use AWS Identity and Access Management (IAM) access keys instead of AWS (root) account access keys. IAM lets you securely control access to AWS services and resources in your AWS account.
4. Download and install the AWS Encryption SDK. To learn how, see the installation instructions for the programming language (p. 8) that you want to use.

Supported Algorithms in the AWS Encryption SDK

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

The SDK implements AES-GCM in one of three ways, as described in the following list. By default, the SDK uses AES-GCM with key derivation and signing, and with a 256-bit encryption key.

For more information about how these implementations are represented and used in the library, see AWS Encryption SDK Algorithms Reference (p. 34).

Implement AES-GCM with Key Derivation and Signing

In this implementation, the SDK uses the data encryption key as an input to the HMAC-based extract-and-expand key derivation function (HKDF) to derive the AES-GCM encryption key. The SDK also adds an Elliptic Curve Digital Signature Algorithm (ECDSA) signature.

The HKDF helps protect against accidental reuse of a data encryption key. The ECDSA signature helps provide stronger authenticity and nonrepudiation of the plaintext data. Use this implementation when the users who encrypt data and those who decrypt it are not equally trusted. This implementation helps protect against some users of a master key attempting to impersonate other users of the master key.

By default, the SDK uses this implementation with a 256-bit encryption key.

Implement AES-GCM with Key Derivation Only

This implementation is like the previous one, but without the ECDSA signature. Use this implementation when the users who encrypt data and those who decrypt it are equally trusted.

Implement AES-GCM without Key Derivation or Signing

In this implementation, the SDK doesn't use a key derivation function to derive the encryption key. Instead, it uses the data encryption key directly as the AES-GCM encryption key. We don't recommend that you use this implementation to generate ciphertext, but the SDK provides it for compatibility reasons.

AWS Encryption SDK Programming Languages

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

Topics

- AWS Encryption SDK for Java (p. 8)
- AWS Encryption SDK for Python (p. 16)

AWS Encryption SDK for Java

For information about installing and using the AWS Encryption SDK for Java, see the following topics.

Topics

- Prerequisites (p. 8)
- Installation (p. 9)
- AWS Encryption SDK for Java Example Code (p. 9)

Prerequisites

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

A Java 8 development environment

If you don't have one, go to Java SE Downloads on the Oracle website, then download and install the Java SE Development Kit (JDK). We recommend Java 8.

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

Bouncy Castle

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

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

```
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcprov-ext-jdk15on</artifactId>
  <version>1.56</version>
</dependency>
```

AWS SDK for Java (Optional)

Although you don't need the AWS SDK for Java to use the AWS Encryption SDK for Java, you do need it to use AWS Key Management Service (AWS KMS) as a master key provider, and to use some of the example Java code (p. 9) in this guide. For more information about installing and configuring the AWS SDK for Java, see AWS SDK for Java.

Installation

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

Manually

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

Using Apache Maven

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

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

After you install the SDK, get started by looking at the example Java code (p. 9) in this guide and the Javadoc on GitHub.

AWS Encryption SDK for Java Example Code

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

Topics

- Encrypting and Decrypting Strings (p. 9)
- Encrypting and Decrypting Byte Streams (p. 11)
- Encrypting and Decrypting Byte Streams with Multiple Master Key Providers (p. 13)

Encrypting and Decrypting Strings

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

```
/*
```

```
 * Copyright 2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use this
 file except
 * in compliance with the License. A copy of the License is located at
 *
 * http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed on an "AS IS"
 BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License
 for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.crypto.examples;

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

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

/**
 * <p>
 * Encrypts and then decrypts a string under a KMS key
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>KMS Key Arn
 * <li>String to encrypt
 * </ol>
 */
public class StringExample {
    private static String keyArn;
    private static String data;

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

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

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

        // Encrypt the data
        //
        // Most encrypted data should have associated encryption context
        // to protect integrity. Here, we'll just use a placeholder value.
        //
        // For more information see:
        // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-Integrity-
of-Your-Encrypted-Data-by-Using-AWS-Key-Management
        final Map<String, String> context = Collections.singletonMap("Example", "String");

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

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

```
        // We need to check the encryption context (and ideally key) to ensure that
        // this was the ciphertext we expected
        if (!decryptResult.getMasterKeyIds().get(0).equals(keyArn)) {
            throw new IllegalStateException("Wrong key id!");
        }

        // The SDK may add information to the encryption context, so we check to ensure
        // that all of our values are present
        for (final Map.Entry<String, String> e : context.entrySet()) {
            if (!e.getValue().equals(decryptResult.getEncryptionContext().get(e.getKey())))
 {
                throw new IllegalStateException("Wrong Encryption Context!");
            }
        }

        // Now that we know we have the correct data, we can output it.
        System.out.println("Decrypted: " + decryptResult.getResult());
    }
}
```

Encrypting and Decrypting Byte Streams

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

```
/*
 * Copyright 2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use this
 file except
 * in compliance with the License. A copy of the License is located at
 *
 * http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed on an "AS IS"
 BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License
 for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.crypto.examples;

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

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

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

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

```
 * <p>
 * Arguments:
 * <ol>
 * <li>fileName
 * </ol>
 *
 * <p>
 * This program demonstrates using a normal java {@link SecretKey} object as a {@link
 MasterKey} to
 * encrypt and decrypt streaming data.
 */
public class FileStreamingExample {
    private static String srcFile;

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

        // In this example, we'll pretend that we loaded this key from
        // some existing store but actually just generate a random one
        SecretKey cryptoKey = retrieveEncryptionKey();

        // Convert key into a provider. We'll use AES GCM because it is
        // a good secure algorithm.
        JceMasterKey masterKey = JceMasterKey.getInstance(cryptoKey, "Example",
 "RandomKey", "AES/GCM/NoPadding");

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

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

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

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

        // Let's decrypt the file now, remembering to check the encryption context
        in = new FileInputStream(srcFile + ".encrypted");
        CryptoInputStream<JceMasterKey> decryptingStream =
 crypto.createDecryptingStream(masterKey, in);
        // Does it have the right encryption context?
        if
 (!"FileStreaming".equals(decryptingStream.getCryptoResult().getEncryptionContext().get("Example")))
 {
            throw new IllegalStateException("Bad encryption context");
        }

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

    /**
     * In the real world, this key will need to be persisted somewhere. For this demo we'll
 generate
     * a new random one each time.
     */
```

```
    private static SecretKey retrieveEncryptionKey() {
        SecureRandom rnd = new SecureRandom();
        byte[] rawKey = new byte[16]; // 128 bits
        rnd.nextBytes(rawKey);
        return new SecretKeySpec(rawKey, "AES");
    }
}
```

Encrypting and Decrypting Byte Streams with Multiple Master Key Providers

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

```
/*
 * Copyright 2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use this
 file except
 * in compliance with the License. A copy of the License is located at
 *
 * http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed on an "AS IS"
 BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License
 for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.crypto.examples;

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

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

/**
 * <p>
 * Encrypts a file using both KMS and an asymmetric key pair.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>KMS KeyArn
 * <li>File Name
 * </ol>
 *
 * Some organizations want the ability to decrypt their data even if KMS is unavailable.
 This
```

```
 * program demonstrates one possible way of accomplishing this by generating an "Escrow"
 RSA
 * key-pair and using that in addition to the KMS key for encryption. The organization
 would keep
 * the RSA private key someplace secure (such as an offline HSM) and distribute the public
 key their
 * developers. This way all standard use would use KMS for decryption, however the
 organization
 * maintains the ability to decrypt all ciphertexts in a completely offline manner.
 */
public class EscrowedEncryptExample {
    private static PublicKey publicEscrowKey;
    private static PrivateKey privateEscrowKey;

    public static void main(final String[] args) throws Exception {
        // In the real world, the public key would be distributed by the organization.
        // For this demo, we'll just generate a new random one each time.
        generateEscrowKeyPair();

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

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

        escrowDecrypt(fileName);
    }

    private static void standardEncrypt(final String kmsArn, final String fileName) throws
 Exception {
        // Standard user encrypting to both KMS and the escrow public key
        // 1. Instantiate the SDK
        final AwsCrypto crypto = new AwsCrypto();

        // 2. Instantiate the providers
        final KmsMasterKeyProvider kms = new KmsMasterKeyProvider(kmsArn);
        // Note that the standard user does not have access to the private escrow
        // key and so simply passes in "null"
        final JceMasterKey escrowPub = JceMasterKey.getInstance(publicEscrowKey, null,
 "Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

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

        // 4. Encrypt the file
        // To simplify the code, we'll be omitted Encryption Context this time. Production
 code
        // should always use Encryption Context. Please see the other examples for more
 information.
        final FileInputStream in = new FileInputStream(fileName);
        final FileOutputStream out = new FileOutputStream(fileName + ".encrypted");
        final CryptoOutputStream<?> encryptingStream =
 crypto.createEncryptingStream(provider, out);

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

    private static void standardDecrypt(final String kmsArn, final String fileName) throws
 Exception {
        // A standard user decrypts the file. They can just use the same provider from
 before
        // or could use a provider just referring to the KMS key. It doesn't matter.
```

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

        // 2. Instantiate the providers
        final KmsMasterKeyProvider kms = new KmsMasterKeyProvider(kmsArn);
        // Note that the standard user does not have access to the private escrow
        // key and so simply passes in "null"
        final JceMasterKey escrowPub = JceMasterKey.getInstance(publicEscrowKey, null,
"Escrow", "Escrow",
                "RSA/ECB/OAEPWithSHA-512AndMGF1Padding");

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

        // 4. Decrypt the file
        // To simplify the code, we'll be omitted Encryption Context this time. Production
code
        // should always use Encryption Context. Please see the other examples for more
information.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".decrypted");
        final CryptoOutputStream<?> decryptingStream =
crypto.createDecryptingStream(provider, out);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();
    }

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

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

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

        // 3. Decrypt the file
        // To simplify the code, we'll be omitted Encryption Context this time. Production
code
        // should always use Encryption Context. Please see the other examples for more
information.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".deescrowed");
        final CryptoOutputStream<?> decryptingStream =
crypto.createDecryptingStream(escrowPriv, out);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();

    }

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

```
    }
}
```

AWS Encryption SDK for Python

For information about installing and using the AWS Encryption SDK for Python, see the following topics.

Topics

- Prerequisites (p. 16)
- Installation (p. 16)
- AWS Encryption SDK for Python Example Code (p. 16)

Prerequisites

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

A supported version of Python

To use this SDK, you need Python 2.7, or Python 3.3 or later. To download Python, see Python downloads.

The pip installation tool for Python

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

Installation

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

To install the latest version

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

To install a specific version

The following example installs version 1.2.0.

```
pip install aws-encryption-sdk=1.2.0
```

When you use pip to install the SDK on Linux, pip builds the `cryptography` library, one of the SDK's dependencies. If your Linux environment doesn't have the tools needed to build the `cryptography` library, you must install them. For more information, see Building `cryptography` on Linux.

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

After you install the SDK, get started by looking at the example Python code (p. 16) in this guide.

AWS Encryption SDK for Python Example Code

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

Topics

- Encrypting and Decrypting Strings (p. 17)
- Encrypting and Decrypting Byte Streams (p. 18)
- Encrypting and Decrypting Byte Streams with Multiple Master Key Providers (p. 19)

Encrypting and Decrypting Strings

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

```
"""
Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License"). You may not use this file
 except
in compliance with the License. A copy of the License is located at

https://aws.amazon.com/apache-2-0/

or in the "license" file accompanying this file. This file is distributed on an "AS IS"
 BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License
 for the
specific language governing permissions and limitations under the License.
"""

from __future__ import print_function

import aws_encryption_sdk


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

    :param str key_arn: Amazon Resource Name (Arn) of the KMS CMK
    :param bytes source_plaintext: Data to encrypt
    :param botocore_session: existing botocore session instance
    :type botocore_session: botocore.session.Session
    """

    # Create the KMS Master Key Provider
    kms_kwargs = dict(key_ids=[key_arn])
    if botocore_session is not None:
        kms_kwargs['botocore_session'] = botocore_session
    master_key_provider = aws_encryption_sdk.KMSMasterKeyProvider(**kms_kwargs)

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

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

    # Validate that the cycled plaintext is identical to the source plaintext
    assert cycled_plaintext == source_plaintext
```

```
    # Validate that the encryption context used by the decryptor has all the key-pairs from
 the encryptor
    assert all(
        pair in decrypted_header.encryption_context.items()
        for pair in encryptor_header.encryption_context.items()
    )

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

Encrypting and Decrypting Byte Streams

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

```
"""
Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License"). You may not use this file
 except
in compliance with the License. A copy of the License is located at

https://aws.amazon.com/apache-2-0/

or in the "license" file accompanying this file. This file is distributed on an "AS IS"
 BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License
 for the
specific language governing permissions and limitations under the License.
"""

import filecmp
import os

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


class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
    """Randomly generates and provides 256-bit keys consistently per unique key id."""
    provider_id = 'static-random'

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

    def _get_raw_key(self, key_id):
        """Retrieves a static, randomly generated, symmetric key for the specified key id.

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

```
def cycle_file(source_plaintext_filename):
    """Encrypts and then decrypts a file under a custom static Master Key Provider.

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

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

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

    # Validate that the cycled plaintext is identical to the source plaintext
    assert filecmp.cmp(source_plaintext_filename, cycled_plaintext_filename)

    # Validate that the encryption context used by the decryptor has all the key-pairs from
 the encryptor
    assert all(
        pair in decryptor.header.encryption_context.items()
        for pair in encryptor.header.encryption_context.items()
    )
    return ciphertext_filename, cycled_plaintext_filename
```

Encrypting and Decrypting Byte Streams with Multiple Master Key Providers

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

```
"""
Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License"). You may not use this file
 except
in compliance with the License. A copy of the License is located at
```

```
https://aws.amazon.com/apache-2-0/

or in the "license" file accompanying this file. This file is distributed on an "AS IS"
 BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License
 for the
specific language governing permissions and limitations under the License.
"""

import filecmp
import os

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


class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
    """Randomly generates and provides 4096-bit RSA keys consistently per unique key id."""
    provider_id = 'static-random'

    def __init__(self, **kwargs):
        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: Key ID
        :returns: Wrapping key which 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 under both a KMS Master Key Provider and a custom
 static Master Key Provider.

    :param str key_arn: Amazon Resource Name (Arn) of the KMS CMK
    :param str source_plaintext_filename: Filename of file to encrypt
    :param botocore_session: existing botocore session instance
    :type botocore_session: botocore.session.Session
    """
```

```
    ciphertext_filename = source_plaintext_filename + '.encrypted'
    cycled_kms_plaintext_filename = source_plaintext_filename + '.kms.decrypted'
    cycled_static_plaintext_filename = source_plaintext_filename + '.static.decrypted'

    # Create KMS Master Key Provider
    kms_kwargs = dict(key_ids=[key_arn])
    if botocore_session is not None:
        kms_kwargs['botocore_session'] = botocore_session
    kms_master_key_provider = aws_encryption_sdk.KMSMasterKeyProvider(**kms_kwargs)

    # Create Static Master Key Provider and add to KMS Master Key Provider
    static_key_id = os.urandom(8)
    static_master_key_provider = StaticRandomMasterKeyProvider()
    static_master_key_provider.add_master_key(static_key_id)

    # Add Static Master Key Provider to KMS Master Key Provider
    kms_master_key_provider.add_master_key_provider(static_master_key_provider)

    # Encrypt plaintext with both KMS and Static Master Keys
    with open(source_plaintext_filename, 'rb') as plaintext, open(ciphertext_filename,
'wb') as ciphertext:
        with aws_encryption_sdk.stream(
            source=plaintext,
            mode='e',
            key_provider=kms_master_key_provider
        ) as encryptor:
            for chunk in encryptor:
                ciphertext.write(chunk)

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

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

    # Validate that the cycled 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)

    # Validate that the encryption context used by the decryptor has all the key-pairs from
the encryptor
    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
```

Frequently Asked Questions

- How is the AWS Encryption SDK different from the AWS SDKs? (p. 23)
- How is the AWS Encryption SDK different from the Amazon S3 encryption client? (p. 23)
- Which cryptographic algorithms are supported by the AWS Encryption SDK, and which one is the default? (p. 24)
- How is the initialization vector (IV) generated and where is it stored? (p. 24)
- How is each data encryption key (DEK) generated, encrypted, and decrypted? (p. 24)
- How do I keep track of the data encryption keys (DEKs) used to encrypt my data? (p. 24)
- How does the AWS Encryption SDK store encrypted data encryption keys (DEKs) with their encrypted data? (p. 24)
- How much overhead does the AWS Encryption SDK's message format add to my encrypted data? (p. 24)
- Can I use my own master key provider? (p. 25)
- Can I encrypt data under more than one master key? (p. 25)
- Which data types can I encrypt with the AWS Encryption SDK? (p. 25)
- How does the AWS Encryption SDK encrypt and decrypt input/output (I/O) streams? (p. 25)

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

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

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

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

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

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

The Amazon S3 encryption client in the AWS SDK for Java, AWS SDK for Ruby, and AWS SDK for .NET provides encryption and decryption for data that you store in Amazon Simple Storage Service

(Amazon S3). These clients are tightly coupled to Amazon S3 and are intended for use only with data stored there.

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

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

The AWS Encryption SDK uses the Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM), known as AES-GCM. The SDK supports 256-bit, 192-bit, and 128-bit encryption keys. In all cases, the length of the initialization vector (IV) is 12 bytes; the length of the authentication tag is 16 bytes. By default, the SDK uses the data encryption key (DEK) 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 Algorithms (p. 7).

For implementation details about the supported algorithms, see Algorithms Reference (p. 34).

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

The AWS Encryption SDK randomly generates a unique IV value for each encryption operation, and stores it in the returned object. For more information, see AWS Encryption SDK Message Format Reference (p. 26).

How is each data encryption key (DEK) generated, encrypted, and decrypted?

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

How do I keep track of the data encryption keys (DEKs) used to encrypt my data?

The AWS Encryption SDK does this for you. When you encrypt data, the AWS Encryption SDK generates a unique symmetric data encryption key (DEK) for that data. Then the SDK encrypts the DEK and stores it (in encrypted form) as part of the returned ciphertext. When you decrypt data, the AWS Encryption SDK extracts the encrypted DEK from the ciphertext, decrypts it, and then uses it to decrypt the data.

How does the AWS Encryption SDK store encrypted data encryption keys (DEKs) with their encrypted data?

The encryption operations in the AWS Encryption SDK return a single data structure, or *message*, that contains the encrypted data and the corresponding encrypted DEK. The message format consists of at least two parts: a *header* and a *body*. In some cases the message format consists of a third part known as a *footer*. The message header contains the encrypted DEK and information about how the message body is formed. The message body contains the encrypted data. The message footer contains a signature that authenticates the message header and message body. For more information, see AWS Encryption SDK Message Format Reference (p. 26).

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

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

- The size of the plaintext data
- Which of the supported algorithms is used
- Whether additional authenticated data (AAD) is provided, and the length of that AAD
- The number and type of master key providers
- The frame size (when framed data (p. 32) is used)

When you use the AWS Encryption SDK with its default configuration, with one CMK in AWS KMS as the master key, with no AAD, and encrypt nonframed data, the overhead is approximately 600 bytes. In general, you can reasonably assume that the AWS Encryption SDK adds overhead of 1 KB or less, not including the provided AAD. For more information, see AWS Encryption SDK Message Format Reference (p. 26).

Can I use my own master key provider?

Yes. The implementation details vary depending on which of the supported programming languages (p. 8) you use. However, all supported languages allow you to define custom master key providers and master keys.

Can I encrypt data under more than one master key?

Yes. The AWS Encryption SDK encrypts the data that you pass to the encryption methods with a unique data encryption key (DEK), and then encrypts that DEK with a master key. You can encrypt the DEK with additional master keys to add redundancy, in case one of the master keys is unavailable for decryption. We provide examples of this pattern in the example code for the supported programming languages (p. 8).

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

The AWS Encryption SDK can encrypt raw bytes (byte arrays), I/O streams (byte streams), and strings. We provide example code for each of the supported programming languages (p. 8).

How does the AWS Encryption SDK encrypt and decrypt input/output (I/O) streams?

The AWS Encryption SDK creates an encrypting or decrypting stream that wraps an underlying I/O stream. The encrypting or decrypting stream performs a cryptographic operation on a read or write call. For example, it can read plaintext data on the underlying stream and encrypt it before returning the result. Or it can read ciphertext from an underlying stream and decrypt it before returning the result. We provide example code for encrypting and decrypting streams for each of the supported programming languages (p. 8).

AWS Encryption SDK Reference

> The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.
>
> To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 8).

The AWS Encryption SDK uses the supported algorithms (p. 7) to return a single data structure or *message* that contains encrypted data and the corresponding encrypted data keys. The following topics explain the algorithms and the data structure. Use this information to build libraries that can read and write ciphertexts that are compatible with this SDK.

Topics

- AWS Encryption SDK Message Format Reference (p. 26)
- AWS Encryption SDK Algorithms Reference (p. 34)
- Body Additional Authenticated Data (AAD) Reference for the AWS Encryption SDK (p. 36)
- AWS Encryption SDK Message Format Examples (p. 37)

AWS Encryption SDK Message Format Reference

> The information on this page is a reference for building your own encryption library that is compatible with the AWS Encryption SDK. If you are not building your own compatible encryption library, you likely do not need this information.
>
> To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 8).

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

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

Topics

- Header Structure (p. 27)
- Body Structure (p. 31)
- Footer Structure (p. 33)

Header Structure

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

Header Structure

Field	Length, in bytes
Version (p. 27)	1
Type (p. 27)	1
Algorithm ID (p. 28)	2
Message ID (p. 28)	16
AAD Length (p. 28)	2
AAD (p. 28)	Variable. Equal to the value specified in the previous 2 bytes (AAD Length).
Encrypted Data Key Count (p. 29)	2
Encrypted Data Key(s) (p. 29)	Variable. Determined by the number of encrypted data keys and the length of each.
Content Type (p. 30)	1
Reserved (p. 30)	4
IV Length (p. 30)	1
Frame Length (p. 30)	4
Header Authentication (p. 30)	Variable. Determined by the algorithm (p. 34) that generated the message.

Version

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

Type

The type of this message format. The type indicates the kind of structure. The only supported type is described as *customer authenticated encrypted data*. Its type value is 128, encoded as byte `80` in hexadecimal notation.

Algorithm ID

An identifier for the algorithm used. It is a 2-byte value interpreted as a 16-bit unsigned integer. For more information about the algorithms, see AWS Encryption SDK Algorithms Reference (p. 34).

Message ID

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

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

AAD Length

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

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 algorithms with signing (p. 34) are used, the encryption context must contain the key-value pair `{'aws-crypto-public-key', Qtxt}`. Qtxt represents the elliptic curve point Q compressed according to SEC 1 version 2.0 and then base64-encoded. The encryption context can contain additional values, but the maximum length of the constructed AAD is 2^16 - 1 bytes.

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

AAD Structure

Field	Length, in bytes
Key-Value Pair Count (p. 28)	2
Key Length (p. 28)	2
Key (p. 29)	Variable. Equal to the value specified in the previous 2 bytes (Key Length).
Value Length (p. 29)	2
Value (p. 29)	Variable. Equal to the value specified in the previous 2 bytes (Value Length).

Key-Value Pair Count

The number of key-value pairs in the AAD. It is a 2-byte value interpreted as a 16-bit unsigned integer that specifies the number of key-value pairs in the AAD. The maximum number of key-value pairs in the AAD is 2^16 - 1.

Key Length

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

Key

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

Value Length

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

Value

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

Encrypted Data Key Count

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

Encrypted Data Key(s)

A sequence of encrypted data keys. The length of the sequence is determined by the number of encrypted data keys and the length of each. The sequence contains at least one encrypted data key.

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

Encrypted Data Key Structure

Field	Length, in bytes
Key Provider ID Length (p. 29)	2
Key Provider ID (p. 29)	Variable. Equal to the value specified in the previous 2 bytes (Key Provider ID Length).
Key Provider Information Length (p. 29)	2
Key Provider Information (p. 29)	Variable. Equal to the value specified in the previous 2 bytes (Key Provider Information Length).
Encrypted Data Key Length (p. 30)	2
Encrypted Data Key (p. 30)	Variable. Equal to the value specified in the previous 2 bytes (Encrypted Data Key Length).

Key Provider ID Length

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

Key Provider ID

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

Key Provider Information Length

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

Key Provider Information

The key provider information. It is determined by the key provider.

When AWS KMS is the key provider, the following are true:

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

Encrypted Data Key Length

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

Encrypted Data Key

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

Content Type

The type of encrypted content, either non-framed or framed.

Non-framed content is not broken into parts; it is a single encrypted blob. Non-framed content is type 1, encoded as the byte `01` in hexadecimal notation.

Framed content is broken into equal-length parts; each part is encrypted separately. Framed content is type 2, encoded as the byte `02` in hexadecimal notation.

Reserved

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

IV Length

The length of the initialization vector (IV). It is a 1-byte value interpreted as an 8-bit unsigned integer that specifies the number of bytes that contain the IV. This value is determined by the IV bytes value of the algorithm (p. 34) that generated the message.

Frame Length

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

Header Authentication

The header authentication is determined by the algorithm (p. 34) that generated the message. The header authentication is calculated over the entire header up to, but not including, the header authentication structure. It consists of an IV and an authentication tag. The bytes are appended in the order shown.

Header Authentication Structure

Field	Length, in bytes
IV (p. 30)	Variable. Determined by the IV bytes value of the algorithm (p. 34) that generated the message.
Authentication Tag (p. 31)	Variable. Determined by the authentication tag bytes value of the algorithm (p. 34) that generated the message.

IV

The initialization vector (IV) used to calculate the header authentication tag. It is a unique value generated only for this use.

Authentication Tag

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

Body Structure

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

Topics

- Non-Framed Data (p. 31)
- Framed Data (p. 32)

Non-Framed Data

Non-framed data is encrypted in a single blob with a unique IV and body AAD (p. 36). The following table describes the fields that form non-framed data. The bytes are appended in the order shown.

Non-Framed Body Structure

Field	Length, in bytes
IV (p. 31)	Variable. Equal to the value specified in the IV Length (p. 30) byte of the header.
Encrypted Content Length (p. 31)	8
Encrypted Content (p. 31)	Variable. Equal to the value specified in the previous 8 bytes (Encrypted Content Length).
Authentication Tag (p. 32)	Variable. Determined by the algorithm implementation (p. 34) used.

IV

The initialization vector (IV) to use with the encryption algorithm (p. 34).

Encrypted Content Length

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

Technically, the maximum allowed value is 2^63 - 1, or 8 exbibytes (8 EiB). However, in practice the maximum value is 2^36 - 32, or 64 gibibytes (64 GiB), due to restrictions imposed by the implemented algorithms (p. 34).

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

Encrypted Content

The encrypted content (ciphertext) as returned by the encryption algorithm (p. 34).

Authentication Tag

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

Framed Data

Framed data is divided into equal-length parts, except for the last part. Each frame is encrypted separately with a unique IV and body AAD (p. 36).

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

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

Framed Body Structure, Regular Frame

Field	Length, in bytes
Sequence Number (p. 32)	4
IV (p. 32)	Variable. Equal to the value specified in the IV Length (p. 30) byte of the header.
Encrypted Content (p. 32)	Variable. Equal to the value specified in the Frame Length (p. 30) of the header.
Authentication Tag (p. 32)	Variable. Determined by the algorithm used, as specified in the Algorithm ID (p. 28) of the header.

Sequence Number

The frame sequence number. It is an incremental counter number for the frame. It is a 4-byte value interpreted as a 32-bit unsigned integer that specifies the number of bytes that contain the encrypted content.

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 IV is a randomly generated value of length specified by the algorithm (p. 34) used.

Encrypted Content

The encrypted content (ciphertext) for the frame, as returned by the encryption algorithm (p. 34).

Authentication Tag

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

Framed Body Structure, Final Frame

Field	Length, in bytes
Sequence Number End (p. 33)	4

Field	Length, in bytes
Sequence Number (p. 33)	4
IV (p. 33)	Variable. Equal to the value specified in the IV Length (p. 30) byte of the header.
Encrypted Content Length (p. 33)	4
Encrypted Content (p. 33)	Variable. Equal to the value specified in the previous 4 bytes (Encrypted Content Length).
Authentication Tag (p. 33)	Variable. Determined by the algorithm used, as specified in the Algorithm ID (p. 28) of the header.

Sequence Number End

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

Sequence Number

The frame sequence number. It is an incremental counter number for the frame. It is a 4-byte value interpreted as a 32-bit unsigned integer that specifies the number of bytes that contain the encrypted content.

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 IV is a randomly generated value of length specified by the algorithm (p. 34) used.

Encrypted Content Length

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

Encrypted Content

The encrypted content (ciphertext) for the frame, as returned by the encryption algorithm (p. 34).

Authentication Tag

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

Footer Structure

When the algorithms with signing (p. 34) are used, the message format contains a footer. The message footer contains a signature calculated over the message header and body. The following table describes the fields that form the footer. The bytes are appended in the order shown.

Footer Structure

Field	Length, in bytes
Signature Length (p. 34)	2

Field	Length, in bytes
Signature (p. 34)	Variable. Equal to the value specified in the previous 2 bytes (Signature Length).

Signature Length

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

Signature

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

AWS Encryption SDK Algorithms Reference

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

To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 8).

To build your own library that can read and write ciphertexts that are compatible with the AWS Encryption SDK, you should understand how the SDK implements the supported algorithms to encrypt raw data. The SDK supports nine algorithm implementations. An implementation specifies the encryption algorithm and mode, encryption key length, key derivation algorithm (if one applies), and signature algorithm (if one applies). The following table contains an overview of each implementation. By default, the SDK uses the first implementation in the following table. The list that follows the table provides more information.

AWS Encryption SDK Algorithm Implementations

Algorithm ID (in 2-byte hex)	Algorithm Name	Data Encryption Key Length (in bits)	Algorithm Mode	IV Length (in bytes)	Authentication Tag Length (in bytes)	Key Derivation Algorithm	Signature Algorithm
`03 78`	AES	256	GCM	12	16	HKDF with SHA-384	ECDSA with P-384
`03 46`	AES	192	GCM	12	16	HKDF with SHA-384	ECDSA with P-384
`02 14`	AES	128	GCM	12	16	HKDF with SHA-256	ECDSA with P-256
`01 78`	AES	256	GCM	12	16	HKDF with SHA-256	Not applicable
`01 46`	AES	192	GCM	12	16	HKDF with SHA-256	Not applicable
`01 14`	AES	128	GCM	12	16	HKDF with SHA-256	Not applicable

Algorithm ID (in 2-byte hex)	Algorithm Name	Data Encryption Key Length (in bits)	Algorithm Mode	IV Length (in bytes)	Authenticat Tag Length (in bytes)	Key Derivation Algorithm	Signature Algorithm
`00 78`	AES	256	GCM	12	16	Not applicable	Not applicable
`00 46`	AES	192	GCM	12	16	Not applicable	Not applicable
`00 14`	AES	128	GCM	12	16	Not applicable	Not applicable

Algorithm ID

A 2-byte value that uniquely identifies an algorithm's implementation. This value is stored in the ciphertext's message header (p. 27).

Algorithm Name

The encryption algorithm used. For all algorithm implementations, the SDK uses the Advanced Encryption Standard (AES) encryption algorithm.

Data Encryption Key Length

The length of the data encryption key (DEK). The SDK supports 256-bit, 192-bit, and 128-bit keys. The DEK is obtained from a master key provider. For some implementations, this DEK is used as input to an HMAC-based extract-and-expand key derivation function (HKDF). The output of the HKDF is used as the DEK in the encryption algorithm. For more information, see **Key Derivation Algorithm** in this list.

Algorithm Mode

The mode used with the encryption algorithm. For all algorithm implementations, the SDK uses Galois/Counter Mode (GCM).

IV Length

The length of the initialization vector (IV) used with AES-GCM.

Authentication Tag Length

The length of the authentication tag used with AES-GCM.

Key Derivation Algorithm

The HMAC-based extract-and-expand key derivation function (HKDF) used to derive the DEK. The SDK uses the HKDF defined in RFC 5869, with the following specifics:

- The hash function used is either SHA-384 or SHA-256, as specified by the algorithm ID.
- For the extract step:
  - No salt is used. Per the RFC, this means that the salt is set to a string of zeros. The string length is equal to the length of the hash function output; that is, 48 bytes for SHA-384 and 32 bytes for SHA-256.
  - The input keying material is the DEK received from the master key provider.
- For the expand step:
  - The input pseudorandom key is the output from the extract step.
  - The input info is a concatenation of the algorithm ID followed by the message ID.

- The length of the output keying material is the **Data Encryption Key Length** described previously. This output is used as the data encryption key (DEK) in the encryption algorithm.

Signature Algorithm

The signature algorithm used to generate a signature over the ciphertext header and body. The SDK uses the Elliptic Curve Digital Signature Algorithm (ECDSA) with the following specifics:

- The elliptic curve used is either the P-384 or P-256 curve, as specified by the algorithm ID. These curves are defined in FIPS PUB 186-4.
- The hash function used is SHA-384 (with the P-384 curve) or SHA-256 (with the P-256 curve).

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

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

To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 8).

Regardless of which type of body data (p. 31) is used to form the message body (non-framed or framed), you must provide additional authenticated data (AAD) to the AES-GCM algorithm (p. 34) for each cryptographic operation. For more information about AAD, see the definition section in the Galois/Counter Mode of Operation (GCM) specification.

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

Body AAD Structure

Field	Length, in bytes
Message ID (p. 36)	16
Body AAD Content (p. 36)	Variable. See **Body AAD Content** in the following list.
Sequence Number (p. 37)	4
Content Length (p. 37)	8

Message ID

The same Message ID (p. 28) value set in the message header.

Body AAD Content

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

For non-framed data (p. 31), use the value `AWSKMSEncryptionClient Single Block`.

For regular frames in framed data (p. 32), use the value `AWSKMSEncryptionClient Frame`.

For the final frame in framed data (p. 32), use the value `AWSKMSEncryptionClient Final Frame`.

Sequence Number

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

For framed data (p. 32), this is the frame sequence number.

For non-framed data (p. 31), use the value 1, encoded as the 4 bytes `00 00 00 01` in hexadecimal notation.

Content Length

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

AWS Encryption SDK Message Format Examples

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

To use the AWS Encryption SDK in one of the supported programming languages, see Programming Languages (p. 8).

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

Topics

- Non-Framed Data (p. 37)
- Framed Data (p. 39)

Non-Framed Data

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

```
+--------+
| Header |
+--------+
01                                          Version (1.0)
80                                          Type (128, customer authenticated encrypted
 data)
0378                                        Algorithm ID (see Algorithms Reference)
B8929B01 753D4A45 C0217F39 404F70FF         Message ID (random 128-bit value)
008E                                        AAD Length (142)
0004                                        AAD Key-Value Pair Count (4)
0005                                        AAD Key-Value Pair 1, Key Length (5)
30746869 73                                 AAD Key-Value Pair 1, Key ("0This")
0002                                        AAD Key-Value Pair 1, Value Length (2)
6973                                        AAD Key-Value Pair 1, Value ("is")
0003                                        AAD Key-Value Pair 2, Key Length (3)
31616E                                      AAD Key-Value Pair 2, Key ("1an")
000A                                        AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                      AAD Key-Value Pair 2, Value ("encryption")
0008                                        AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                           AAD Key-Value Pair 3, Key ("2context")
0007                                        AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                             AAD Key-Value Pair 3, Value ("example")
0015                                        AAD Key-Value Pair 4, Key Length (21)
```

```
6177732D 63727970 746F2D70 75626C69        AAD Key-Value Pair 4, Key ("aws-crypto-public-
key")
632D6B65 79
0044                                       AAD Key-Value Pair 4, Value Length (68)
41734738 67473949 6E4C5075 3136594B        AAD Key-Value Pair 4, Value
 ("AsG8gG9InLPu16YKlqXTOD+nykG8YqHAhqecj8aXfD2e5B4gtVE73dZkyClA+rAMOQ==")
6C715854 4F442B6E 796B4738 59714841
68716563 6A386158 66443265 35423467
74564537 33645A6B 79436C41 2B72414D
4F513D3D
0002                                       Encrypted Data Key Count (2)
0007                                       Encrypted Data Key 1, Key Provider ID Length (7)
6177732D 6B6D73                            Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                       Encrypted Data Key 1, Key Provider Information
 Length (75)
61726E3A 6177733A 6B6D733A 75732D77        Encrypted Data Key 1, Key Provider Information
 ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                       Encrypted Data Key 1, Encrypted Data Key Length
 (167)
01010200 7857A1C1 F7370545 4ECA7C83        Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C28 4116449A
0F2A0383 659EF802 0110803B B23A8133
3A33605C 48840656 C38BCB1F 9CCE7369
E9A33EBE 33F46461 0591FECA 947262F3
418E1151 21311A75 E575ECC5 61A286E0
3E2DEBD5 CB005D
0007                                       Encrypted Data Key 2, Key Provider ID Length (7)
6177732D 6B6D73                            Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                       Encrypted Data Key 2, Key Provider Information
 Length (78)
61726E3A 6177733A 6B6D733A 63612D63        Encrypted Data Key 2, Key Provider Information
 ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                       Encrypted Data Key 2, Encrypted Data Key Length
 (167)
01010200 78FAFFFB D6DE06AF AC72F79B        Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040CB2 A820D0CC
76616EF2 A6B30D02 0110803B 8073D0F1
FDD01BD9 B0979082 099FDBFC F7B13548
3CC686D7 F3CF7C7A CCC52639 122A1495
71F18A46 80E2C43F A34C0E58 11D05114
2A363C2A E11397
01                                         Content Type (1, non-framed data)
00000000                                   Reserved
0C                                         IV Length (12)
00000000                                   Frame Length (0, non-framed data)
734C1BBE 032F7025 84CDA9D0                 IV
2C82BB23 4CBF4AAB 8F5C6002 622E886C        Authentication Tag
+------+
```

```
| Body |
+------+
D39DD3E5 915E0201 77A4AB11                IV
00000000 0000028E                         Encrypted Content Length (654)
E8B6F955 B5F22FE4 FD890224 4E1D5155       Encrypted Content
5871BA4C 93F78436 1085E4F8 D61ECE28
59455BD8 D76479DF C28D2E0B BDB3D5D3
E4159DFE C8A944B6 685643FC EA24122B
6766ECD5 E3F54653 DF205D30 0081D2D8
55FCDA5B 9F5318BC F4265B06 2FE7C741
C7D75BCC 10F05EA5 0E2F2F40 47A60344
ECE10AA7 559AF633 9DE2C21B 12AC8087
95FE9C58 C65329D1 377C4CD7 EA103EC1
31E4F48A 9B1CC047 EE5A0719 704211E5
B48A2068 8060DF60 B492A737 21B0DB21
C9B21A10 371E6179 78FAFB0B BAAEC3F4
9D86E334 701E1442 EA5DA288 64485077
54C0C231 AD43571A B9071925 609A4E59
B8178484 7EB73A4F AAE46B26 F5B374B8
12B0000C 8429F504 936B2492 AAF47E94
A5BA804F 7F190927 5D2DF651 B59D4C2F
A15D0551 DAEBA4AF 2060D0D5 CB1DA4E6
5E2034DB 4D19E7CD EEA6CF7E 549C86AC
46B2C979 AB84EE12 202FD6DF E7E3C09F
C2394012 AF20A97E 369BCBDA 62459D3E
C6FFB914 FEFD4DE5 88F5AFE1 98488557
1BABBAE4 BE55325E 4FB7E602 C1C04BEE
F3CB6B86 71666C06 6BF74E1B 0F881F31
B731839B CF711F6A 84CA95F5 958D3B44
E3862DF6 338E02B5 C345CFF8 A31D54F3
6920AA76 0BF8E903 552C5A04 917CCD11
D4E5DF5C 491EE86B 20C33FE1 5D21F0AD
6932E67C C64B3A26 B8988B25 CFA33E2B
63490741 3AB79D60 D8AEFBE9 2F48E25A
978A019C FE49EE0A 0E96BF0D D6074DDB
66DFF333 0E10226F 0A1B219C BE54E4C2
2C15100C 6A2AA3F1 88251874 FDC94F6B
9247EF61 3E7B7E0D 29F3AD89 FA14A29C
76E08E9B 9ADCDF8C C886D4FD A69F6CB4
E24FDE26 3044C856 BF08F051 1ADAD329
C4A46A1E B5AB72FE 096041F1 F3F3571B
2EAFD9CB B9EB8B83 AE05885A 8F2D2793
1E3305D9 0C9E2294 E8AD7E3B 8E4DEC96
6276C5F1 A3B7E51E 422D365D E4C0259C
50715406 822D1682 80B0F2E5 5C94
65B2E942 24BEEA6E A513F918 CCEC1DE3      Authentication Tag
+--------+
| Footer |
+--------+
0067                                     Signature Length (103)
30650230 7229DDF5 B86A5B64 54E4D627      Signature
CBE194F1 1CC0F8CF D27B7F8B F50658C0
BE84B355 3CED1721 A0BE2A1B 8E3F449E
1BEB8281 023100B2 0CB323EF 58A4ACE3
1559963B 889F72C3 B15D1700 5FB26E61
331F3614 BC407CEE B86A66FA CBF74D9E
34CB7E4B 363A38
```

Framed Data

The following example shows the message format for framed data.

```
+--------+
```

```
| Header |
+--------+
01                                          Version (1.0)
80                                          Type (128, customer authenticated encrypted
 data)
0378                                        Algorithm ID (see Algorithms Reference)
6E7C0FBD 4DF4A999 717C22A2 DDFE1A27         Message ID (random 128-bit value)
008E                                        AAD Length (142)
0004                                        AAD Key-Value Pair Count (4)
0005                                        AAD Key-Value Pair 1, Key Length (5)
30746869 73                                 AAD Key-Value Pair 1, Key ("0This")
0002                                        AAD Key-Value Pair 1, Value Length (2)
6973                                        AAD Key-Value Pair 1, Value ("is")
0003                                        AAD Key-Value Pair 2, Key Length (3)
31616E                                      AAD Key-Value Pair 2, Key ("1an")
000A                                        AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                      AAD Key-Value Pair 2, Value ("encryption")
0008                                        AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                           AAD Key-Value Pair 3, Key ("2context")
0007                                        AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                             AAD Key-Value Pair 3, Value ("example")
0015                                        AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69         AAD Key-Value Pair 4, Key ("aws-crypto-public-
key")
632D6B65 79
0044                                        AAD Key-Value Pair 4, Value Length (68)
416A4173 7569326F 7430364C 4B77715A         AAD Key-Value Pair 4, Value
 ("AjAsui2ot06LKwqZXDJnU/Aqc2vD+0OkpOZ1cc8Tg2qd7rs5aLTg7lvfUEW/86+/5w==")
58444A6E 552F4171 63327644 2B304F6B
704F5A31 63633854 67327164 37727335
614C5467 376C7666 5545572F 38362B2F
35773D3D
0002                                        EncryptedDataKeyCount (2)
0007                                        Encrypted Data Key 1, Key Provider ID Length (7)
6177732D 6B6D73                             Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                        Encrypted Data Key 1, Key Provider Information
 Length (75)
61726E3A 6177733A 6B6D733A 75732D77         Encrypted Data Key 1, Key Provider Information
 ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                        Encrypted Data Key 1, Encrypted Data Key Length
 (167)
01010200 7857A1C1 F7370545 4ECA7C83         Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C3F F02C897B
7A12EB19 8BF2D802 0110803B 24003D1F
A5474FBC 392360B5 CB9997E0 6A17DE4C
A6BD7332 6BF86DAB 60D8CCB8 8295DBE9
4707E356 ADA3735A 7C52D778 B3135A47
9F224BF9 E67E87
0007                                        Encrypted Data Key 2, Key Provider ID Length (7)
6177732D 6B6D73                             Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                        Encrypted Data Key 2, Key Provider Information
 Length (78)
61726E3A 6177733A 6B6D733A 63612D63         Encrypted Data Key 2, Key Provider Information
 ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
```

```
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                   Encrypted Data Key 2, Encrypted Data Key Length
 (167)
01010200 78FAFFFB D6DE06AF AC72F79B    Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C36 CD985E12
D218B674 5BBC6102 0110803B 0320E3CD
E470AA27 DEAB660B 3E0CE8E0 8B1A89E4
57DCC69B AAB1294F 21202C01 9A50D323
72EBAAFD E24E3ED8 7168E0FA DB40508F
556FBD58 9E621C
02                                     Content Type (2, framed data)
00000000                               Reserved
0C                                     IV Length (12)
00000100                               Frame Length (256)
4ECBD5C0 9899CA65 923D2347             IV
0B896144 0CA27950 CA571201 4DA58029    Authentication Tag
+------+
| Body |
+------+
00000001                               Frame 1, Sequence Number (1)
6BD3FE9C ADBCB213 5B89E8F1             Frame 1, IV
1F6471E0 A51AF310 10FA9EF6 F0C76EDF    Frame 1, Encrypted Content
F5AFA33C 7D2E8C6C 9C5D5175 A212AF8E
FBD9A0C3 C6E3FB59 C125DBF2 89AC7939
BDEE43A8 0F00F49E ACBBD8B2 1C785089
A90DB923 699A1495 C3B31B50 0A48A830
201E3AD9 1EA6DA14 7F6496DB 6BC104A4
DEB7F372 375ECB28 9BF84B6D 2863889F
CB80A167 9C361C4B 5EC07438 7A4822B4
A7D9D2CC 5150D414 AF75F509 FCE118BD
6D1E798B AEBA4CDB AD009E5F 1A571B77
0041BC78 3E5F2F41 8AF157FD 461E959A
BB732F27 D83DC36D CC9EBC05 00D87803
57F2BB80 066971C2 DEEA062F 4F36255D
E866C042 E1382369 12E9926B BA40E2FC
A820055F FB47E428 41876F14 3B6261D9
5262DB34 59F5D37E 76E46522 E8213640
04EE3CC5 379732B5 F56751FA 8E5F26AD    Frame 1, Authentication Tag
00000002                               Frame 2, Sequence Number (2)
F1140984 FF25F943 959BE514             Frame 2, IV
216C7C6A 2234F395 F0D2D9B9 304670BF    Frame 2, Encrypted Content
A1042608 8A8BCB3F B58CF384 D72EC004
A41455B4 9A78BAC9 36E54E68 2709B7BD
A884C1E1 705FF696 E540D297 446A8285
23DFEE28 E74B225A 732F2C0C 27C6BDA2
7597C901 65EF3502 546575D4 6D5EBF22
1FF787AB 2E38FD77 125D129C 43D44B96
778D7CEE 3C36625F FF3A985C 76F7D320
ED70B1F3 79729B47 E7D9B5FC 02FCE9F5
C8760D55 7779520A 81D54F9B EC45219D
95941F7E 5CBAEAC8 CEC13B62 1464757D
AC65B6EF 08262D74 44670624 A3657F7F
2A57F1FD E7060503 AC37E197 2F297A84
DF1172C2 FA63CF54 E6E2B9B6 A86F582B
3B16F868 1BBC5E4D 0B6919B3 08D5ABCF
FECDC4A4 8577F08B 99D766A1 E5545670
A61F0A3B A3E45A84 4D151493 63ECA38F    Frame 2, Authentication Tag
FFFFFFFF                               Final Frame, Sequence Number End
00000003                               Final Frame, Sequence Number (3)
35F74F11 25410F01 DD9E04BF             Final Frame, IV
0000008E                               Final Frame, Encrypted Content Length (142)
```

```
F7A53D37 2F467237 6FBD0B57 D1DFE830        Final Frame, Encrypted Content
B965AD1F A910AA5F 5EFFFFF4 BC7D431C
BA9FA7C4 B25AF82E 64A04E3A A0915526
88859500 7096FABB 3ACAD32A 75CFED0C
4A4E52A3 8E41484D 270B7A0F ED61810C
3A043180 DF25E5C5 3676E449 0986557F
C051AD55 A437F6BC 139E9E55 6199FD60
6ADC017D BA41CDA4 C9F17A83 3823F9EC
B66B6A5A 80FDB433 8A48D6A4 21CB
811234FD 8D589683 51F6F39A 040B3E3B        Final Frame, Authentication Tag
+--------+
| Footer |
+--------+
0066                                       Signature Length (102)
30640230 085C1D3C 63424E15 B2244448        Signature
639AED00 F7624854 F8CF2203 D7198A28
758B309F 5EFD9D5D 2E07AD0B 467B8317
5208B133 02301DF7 2DFC877A 66838028
3C6A7D5E 4F8B894E 83D98E7C E350F424
7E06808D 0FE79002 E24422B9 98A0D130
A13762FF 844D
```

Document History for the AWS Encryption SDK Developer Guide

The following table describes the significant changes to this documentation.

Latest documentation update: March 21, 2017

Change	Description	Date
Update	Expanded the Message Format Reference (p. 26) documentation into a new AWS Encryption SDK Reference (p. 26) section. Added a section about the AWS Encryption SDK's Supported Algorithms (p. 7).	March 21, 2017
New release	The AWS Encryption SDK now supports the Python (p. 16) programming language, in addition to Java (p. 8).	March 21, 2017
Initial release	Initial release of the AWS Encryption SDK and this documentation.	March 22, 2016