aws

개발자 가이드

# AWS Encryption SDK

# AWS Encryption SDK: 개발자 가이드

# Table of Contents

# 란 무엇입니까 AWS Encryption SDK?

는 모든 사용자가 업계 표준 및 모범 사례를 사용하여 데이터를 쉽게 암호화하고 해독할 수 있도록 설계된 클라이언트 측 암호화 라이브러리 AWS Encryption SDK 입니다. 그럼으로써 데이터의 암호화 및 복호화 방법보다 애플리케이션의 핵심 기능에 집중할 수 있습니다. AWS Encryption SDK 는 Apache 2.0 라이선스에 따라 무료로 제공됩니다.

는 다음과 같은 질문에 AWS Encryption SDK 답합니다.

- 어떤 암호화 알고리즘을 사용해야 하나요?
- 이 알고리즘을 어떻게, 어떤 모드에서 사용해야 하나요?
- 암호화 키는 어떻게 생성하나요?
- 암호화 키를 보호하려면 어떻게 해야 하며 어디에 저장해야 하나요?
- 암호화된 데이터를 이동 가능하게 만들려면 어떻게 해야 하나요?
- 의도한 수신자가 내 암호화된 데이터를 읽을 수 있도록 하려면 어떻게 해야 하나요?
- 기록된 시점과 읽은 시점 사이에 내 암호화된 데이터가 수정되지 않도록 하려면 어떻게 해야 하나요?
- 에서 AWS KMS 반환하는 데이터 키를 사용하려면 어떻게 해야 합니까?

를 사용하여 데이터를 보호하는 데 사용할 래핑 키를 결정하는 마스터 키 공급자 또는 키링을 AWS Encryption SDK정의합니다. 그런 다음에서 제공하는 간단한 방법을 사용하여 데이터를 암호화하고 해독합니다 AWS Encryption SDK. 는 나머지를 AWS Encryption SDK 수행합니다.

이 없으면 애플리케이션의 핵심 기능보다 암호화 솔루션을 구축하는 데 더 많은 노력을 AWS Encryption SDK기울일 수 있습니다. 는 다음 사항을 제공하여 이러한 질문에 AWS Encryption SDK 답합니다.

암호화 모범 사례에 따른 기본 구현

기본적으로는 암호화하는 각 데이터 객체에 대해 고유한 데이터 키를 AWS Encryption SDK 생성합니다. 이는 각 암호화 작업에 고유한 데이터 키를 사용하는 암호화 모범 사례를 따릅니다.

는 안전하고 인증된 대칭 키 알고리즘을 사용하여 데이터를 AWS Encryption SDK 암호화합니다. 자세한 내용은 the section called “지원 알고리즘 제품군” 단원을 참조하십시오.

래핑 키를 사용하여 데이터 키를 보호하기 위한 프레임워크

는 하나 이상의 래핑 키로 데이터를 암호화하여 데이터를 암호화하는 데이터 키를 AWS Encryption SDK 보호합니다. 는 둘 이상의 래핑 키로 데이터 키를 암호화하는 프레임워크를 제공하여 암호화된 데이터를 이식할 수 있도록 AWS Encryption SDK 합니다.

예를 들어 AWS KMS key 의 AWS KMS 및 온프레미스 HSM의 키로 데이터를 암호화합니다. 어느 하나의 래핑 키를 사용할 수 없거나 호출자가 두 키를 모두 사용할 권한이 없는 경우 데이터를 복호화하는 데 두 래핑 키 중 어느 것을 사용해도 됩니다.

암호화된 데이터와 함께 암호화된 데이터 키를 저장하는 형식 메시지

는 암호화된 데이터와 암호화된 데이터 키를 정의된 데이터 형식을 사용하는 암호화된 메시지에 함께 AWS Encryption SDK 저장합니다. 즉,에서 데이터를 암호화하는 데이터 키를 추적하거나 보호할 필요가 없습니다 AWS Encryption SDK .

의 일부 언어 구현에는 AWS SDK가 AWS Encryption SDK 필요하지만 에는 AWS Encryption SDK 가 필요하지 않으며 AWS 서비스에 종속 AWS 계정 되지 않습니다. 를 사용하여 데이터를 AWS KMS keys 보호하도록 선택한 AWS 계정 경우에만이 필요합니다.

## 오픈 소스 리포지토리에서 개발

는 GitHub의 오픈 소스 리포지토리에서 개발 AWS Encryption SDK 됩니다. 이러한 리포지토리를 사용하여 코드를 보고, 문제를 읽고 제출하고, 언어 구현과 관련된 정보를 찾을 수 있습니다.

- AWS Encryption SDK for C - aws-encryption-sdk-c
- AWS Encryption SDK for .NET - `aws-encryption-sdk`리포지토리의 .NET 디렉터리입니다.
- AWS 암호화 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
- AWS Encryption SDK for Rust - `aws-encryption-sdk`리포지토리의 Rust 디렉터리입니다.
- AWS Encryption SDK for Go - `aws-encryption-sdk`리포지토리의 Go 디렉터리

# 암호화 라이브러리 및 서비스와의 호환성

AWS Encryption SDK 는 여러 프로그래밍 언어로 지원됩니다. 모든 언어 구현은 상호 연동이 가능합니다. 하나의 언어 구현으로 암호화하고 다른 언어 구현으로 복호화할 수 있습니다. 상호 연동성에는 언어 제약 조건이 적용될 수 있습니다. 이 경우 이러한 제약 조건은 언어 구현에 대한 주제에 설명되어 있습니다. 또한 암호화 및 복호화를 수행할 때는 호환되는 키링이나 마스터 키 및 마스터 키 공급자를 사용해야 합니다. 자세한 내용은 the section called “키링 호환성”을 참조하세요.

그러나는 다른 라이브러리와 상호 작용할 AWS Encryption SDK 수 없습니다. 각 라이브러리는 암호화된 데이터를 다른 형식으로 반환하므로 한 라이브러리로 암호화하고 다른 라이브러리로 복호화할 수 없습니다.

DynamoDB Encryption Client 및 Amazon S3 클라이언트 측 암호화

는 DynamoDB Encryption Client 또는 Amazon S3 클라이언트 측 암호화로 암호화된 데이터를 해독할 수 AWS Encryption SDK 없습니다. 이러한 라이브러리는이 AWS Encryption SDK 반환하는 암호화된 메시지를 해독할 수 없습니다.

AWS Key Management Service (AWS KMS)

는 AWS KMS keys 및 데이터 키를 사용하여 다중 리전 KMS 키를 포함한 데이터를 보호할 AWS Encryption SDK 수 있습니다. 예를 들어의 하나 이상의 AWS KMS keys 에서 데이터를 암호화 AWS Encryption SDK 하도록를 구성할 수 있습니다 AWS 계정. 그러나 AWS Encryption SDK 를 사용하여 해당 데이터를 복호화해야 합니다.

는 AWS KMS 암호화 또는 재암호화 작업이 반환하는 사이퍼텍스트를 해독할 수 AWS Encryption SDK 없습니다. ReEncrypt 마찬가지로 AWS KMS 복호화 작업은이 AWS Encryption SDK 반환하는 암호화된 메시지를 복호화할 수 없습니다.

는 대칭 암호화 KMS 키만 AWS Encryption SDK 지원합니다. 비대칭 KMS 키는 암호화 또는 AWS Encryption SDK로그인에 사용할 수 없습니다. AWS Encryption SDK 는 메시지에 서명하는 알고리즘 제품군에 대한 자체 ECDSA 서명 키를 생성합니다.

# 지원 및 유지 관리

AWS Encryption SDK 는 버전 관리 및 수명 주기 단계를 포함하여 AWS SDK 및 도구가 사용하는 것과 동일한 유지 관리 정책을 사용합니다. 프로그래밍 AWS Encryption SDK 언어에 사용할 수 있는 최신 버전의를 사용하고 새 버전이 릴리스되면 업그레이드하는 것이 좋습니다. 버전에 1.7.x 이전 AWS Encryption SDK 버전에서 버전 2.0.x 이상으로 업그레이드하는 등 중요한 변경이 필요한 경우 도움이 되는 자세한 지침을 제공합니다.

의 각 프로그래밍 언어 구현 AWS Encryption SDK 은 별도의 오픈 소스 GitHub 리포지토리에서 개발됩니다. 각 버전의 수명 주기 및 지원 단계는 리포지토리마다 다를 수 있습니다. 예를 들어 특정 버전의 AWS Encryption SDK 는 한 프로그래밍 언어의 일반 가용성(전체 지원) 단계이지만 다른 프로그래밍 언어의 end-of-support 단계일 수 있습니다. 가능하면 완전히 지원되는 버전을 사용하고 더 이상 지원되지 않는 버전은 피하는 것이 좋습니다.

프로그래밍 언어에 맞는 AWS Encryption SDK 버전의 수명 주기 단계를 찾으려면 각 AWS Encryption SDK 리포지토리의 `SUPPORT_POLICY.rst` 파일을 참조하세요.

- AWS Encryption SDK for C - SUPPORT_POLICY.rst
- AWS Encryption SDK .NET용 - SUPPORT_POLICY.rst
- AWS 암호화 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

자세한 내용은 SDK 및 도구 참조 안내서의 의 버전 AWS Encryption SDK AWS SDKs. AWS SDKs

## 자세히 알아보기

AWS Encryption SDK 및 클라이언트 측 암호화에 대한 자세한 내용은 다음 소스를 참조하십시오.

- 이 SDK에서 사용되는 용어 및 개념에 대한 도움말은 의 개념 AWS Encryption SDK 섹션을 참조하세요.
- 모범 사례 지침은 모범 사례 AWS Encryption SDK 섹션을 참조하세요.
- 이 SDK의 작동 방식에 대한 자세한 내용은 SDK 작동 방식 섹션을 참조하세요.
- 에서 옵션을 구성하는 방법을 보여주는 예제는 섹션을 AWS Encryption SDK참조하세요구성 AWS Encryption SDK.
- 자세한 기술 정보는 레퍼런스 섹션을 참조하세요.
- 의 기술 사양은 GitHub의 AWS Encryption SDK 사양을 AWS Encryption SDK참조하세요.
- 사용에 대한 질문에 대한 답변을 보려면 AWS Crypto 도구 토론 포럼을 AWS Encryption SDK읽고 게시하십시오.

다양한 프로그래밍 언어로를 구현하는 AWS Encryption SDK 방법에 대한 정보입니다.

- C: GitHub의 , AWS Encryption SDK for C AWS Encryption SDK C 설명서 및 aws-encryption-sdk-c 리포지토리를 참조하세요.
- C#/.NET: AWS Encryption SDK .NET용 문서와, GitHub에 있는 **aws-encryption-sdk** 리포지토리의 aws-encryption-sdk-net 디렉터리를 참조하세요.
- 명령줄 인터페이스: AWS Encryption SDK 명령줄 인터페이스, AWS 암호화 CLI용 문서 읽기 및 GitHub의 aws-encryption-sdk-cli 리포지토리를 참조하세요.
- Java: GitHub의 AWS Encryption SDK for Java, AWS Encryption SDK Javadoc 및 aws-encryption-sdk-java 리포지토리를 참조하세요.

  JavaScript: GitHub의 the section called “JavaScript” 및 aws-encryption-sdk-javascript 리포지토리를 참조하세요.
- Python: GitHub의 AWS Encryption SDK for Python, AWS Encryption SDK Python 설명서 및 aws-encryption-sdk-python 리포지토리를 참조하세요.

## 피드백 보내기

우리는 여러분의 의견을 환영합니다. 질문이나 의견이 있거나 보고해야 할 문제가 있는 경우 다음 리소스를 사용하세요.

- 에서 잠재적 보안 취약성을 발견한 경우 AWS 보안에 알리 AWS Encryption SDK세요. 공개적으로 GitHub 문제를 작성하지 마세요.
- 에 대한 피드백을 제공하려면 사용 중인 프로그래밍 언어에 대한 문제를 GitHub 리포지토리에 AWS Encryption SDK제출합니다.
- 이 문서에 대한 피드백을 제공하려면 이 페이지의 피드백 링크를 사용하세요. GitHub에서 이 문서의 오픈 소스 리포지토리인 aws-encryption-sdk-docs에 문제를 제기하거나 기고할 수도 있습니다.

## 의 개념 AWS Encryption SDK

이 섹션에서는에 사용되는 개념을 소개 AWS Encryption SDK하고 용어집과 참조를 제공합니다. 의 AWS Encryption SDK 작동 방식과 이를 설명하는 데 사용하는 용어를 이해하는 데 도움이 되도록 설계되었습니다.

도움이 필요하세요?

- 가 봉투 암호화를 AWS Encryption SDK 사용하여 데이터를 보호하는 방법을 알아봅니다.

- 봉투 암호화의 구성 요소인 데이터를 보호하는 데이터 키와 데이터 키를 보호하는 래핑 키에 대해 알아봅니다.
- 사용하는 래핑 키를 결정하는 키링과 마스터 키 공급자에 대해 알아봅니다.
- 암호화 프로세스에 무결성을 더하는 암호화 컨텍스트에 대해 알아봅니다. 이는 선택 사항이지만 권장되는 모범 사례입니다.
- 암호화 메서드가 반환하는 암호화된 메시지에 대해 알아봅니다.
- 그런 다음 원하는 프로그래밍 언어로를 사용할 준비가 AWS Encryption SDK 되었습니다.

주제

- 봉투 암호화
- 데이터 키
- 래핑 키
- 키링 및 마스터 키 공급자
- 암호화 컨텍스트
- 암호화된 메시지
- 알고리즘 제품군
- 암호화 구성 요소 관리자
- 대칭 및 비대칭 암호화
- 키 커밋
- 커밋 정책
- 디지털 서명

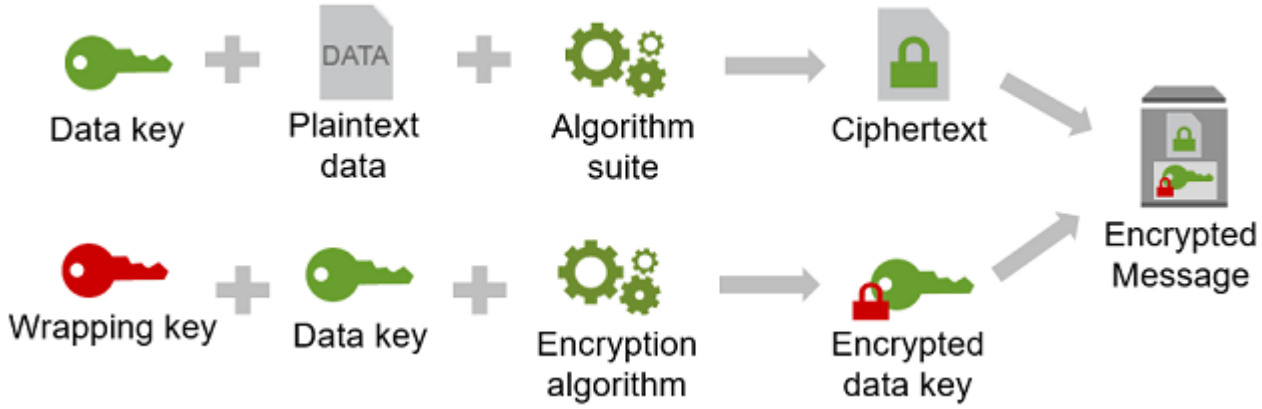
## 봉투 암호화

암호화된 데이터의 보안은 부분적으로 복호화할 수 있는 데이터 키를 보호하는 데 달려 있습니다. 데이터 키를 보호하기 위해 널리 인정되는 모범 사례 중 하나는 데이터 키를 암호화하는 것입니다. 이렇게 하려면 키-암호화 키 또는 래핑 키라고 하는 또 다른 암호화 키가 필요합니다. 래핑 키를 사용하여 데이터 키를 암호화하는 방법을 봉투 암호화라고 합니다.

데이터 키 보호

는 고유한 데이터 키로 각 메시지를 AWS Encryption SDK 암호화합니다. 그러면 지정한 래핑 키에서 데이터 키가 암호화됩니다. 반환된 암호화된 메시지에 암호화된 데이터와 함께 암호화된 데이터 키를 저장합니다.

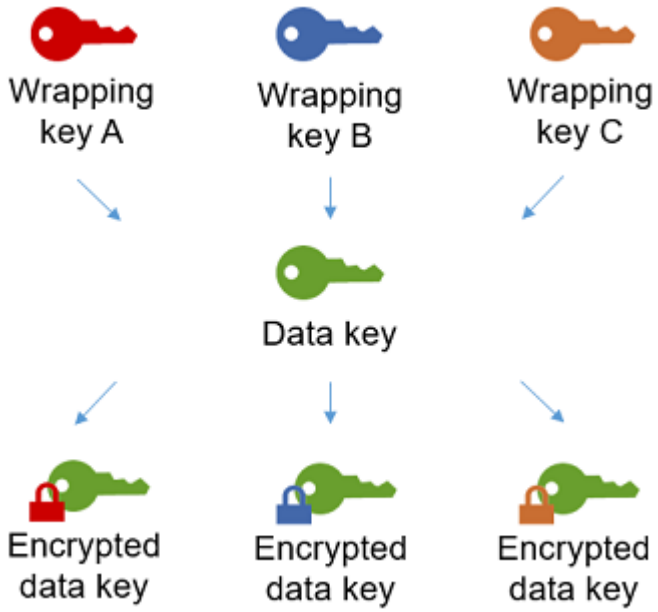
래핑 키를 지정하려면 키링 또는 마스터 키 공급자를 사용합니다.




여러 개의 래핑 키로 동일한 데이터 암호화

여러 개의 래핑 키로 데이터 키를 암호화할 수 있습니다. 사용자마다 다른 래핑 키를 제공하거나, 유형이나 위치가 다른 래핑 키를 제공할 수 있습니다. 각 래핑 키는 동일한 데이터 키를 암호화합니다. 는 암호화된 모든 데이터 키를 암호화된 데이터와 함께 암호화된 메시지에 AWS Encryption SDK 저장합니다.

데이터를 복호화하려면 암호화된 데이터 키 중 하나를 복호화할 수 있는 래핑 키를 제공해야 합니다.



여러 알고리즘의 강점 결합

기본적으로 데이터를 암호화하기 위해는 AES-GCM 대칭 암호화, 키 파생 함수(HKDF) 및 서명이 포함된 정교한 알고리즘 제품군을 AWS Encryption SDK 사용합니다. 데이터 키를 암호화하려면 래핑 키에 적합한 대칭 또는 비대칭 암호화 알고리즘을 지정할 수 있습니다.

일반적으로 대칭 키 암호화 알고리즘이 비대칭 또는 퍼블릭 키 암호화보다 빠르고 더 작은 사이퍼텍스트를 생성합니다. 그러나 퍼블릭 키 알고리즘은 고유한 역할 구분을 제공하고 키 관리가 더 쉽습니다. 각각의 장점을 결합하려면 대칭 키 암호화로 데이터를 암호화한 다음 퍼블릭 키 암호화로 데이터 키를 암호화하면 됩니다.

## 데이터 키

데이터 키는 AWS Encryption SDK 가 데이터를 암호화하는 데 사용하는 암호화 키입니다. 각 데이터 키는 암호화 키 요구 사항을 준수하는 바이트 배열입니다. 데이터 키 캐싱을 사용하지 않는 한는 고유한 데이터 키를 AWS Encryption SDK 사용하여 각 메시지를 암호화합니다.

데이터 키를 지정, 생성, 구현, 확장, 보호 또는 사용할 필요가 없습니다. 암호화 및 복호화 작업을 호출할 때 AWS Encryption SDK 가 이를 대신 수행합니다.

데이터 키를 보호하기 위해 AWS Encryption SDK 는 래핑 키 또는 마스터 키라고 하는 하나 이상의 키 암호화 키로 암호화합니다. 는 일반 텍스트 데이터 키를 AWS Encryption SDK 사용하여 데이터를 암호화한 후 가능한 한 빨리 메모리에서 제거합니다. 그런 다음 암호화 작업이 반환하는 암호화된 메시지에 암호화된 데이터와 함께 암호화된 데이터 키를 저장합니다. 자세한 내용은 the section called “SDK 작동 방식”을 참조하세요.

> **Tip**
>
> 에서는 데이터 키와 데이터 암호화 키를 AWS Encryption SDK구분합니다. 기본 제품군을 포함하여 지원되는 알고리즘 제품군 중 일부는 데이터 키가 암호화 한도에 도달하지 않도록 하는 키 유도 함수를 사용합니다. 키 유도 함수는 데이터 키를 입력으로 받아 실제로 데이터를 암호화하는 데 사용되는 데이터 암호화 키를 반환합니다. 이러한 이유로 데이터가 데이터 키에 "의해" 암호화되는 것이 아니라 데이터 키"에서" 암호화된다고 말하는 경우가 많습니다.

암호화된 각 데이터 키에는 해당 데이터 키를 암호화한 래핑 키의 식별자를 비롯한 메타데이터가 포함됩니다. 이 메타데이터 AWS Encryption SDK 를 사용하면에서 복호화 시 유효한 래핑 키를 더 쉽게 식별할 수 있습니다.

# 래핑 키

래핑 키는 AWS Encryption SDK 가 데이터를 암호화하는 데이터 키를 암호화하는 데 사용하는 키-암호화 키입니다. 각각의 일반 텍스트 데이터 키는 한 개 또는 여러 개의 래핑 키로 암호화될 수 있습니다. 키링 또는 마스터 키 공급자를 구성할 때 데이터를 보호하기 위해 사용할 래핑 키를 결정합니다.

> **Note**
>
> 래핑 키는 키링 또는 마스터 키 공급자에 있는 키를 말합니다. 마스터 키는 일반적으로 마스터 키 공급자를 사용할 때 인스턴스화하는 `MasterKey` 클래스와 연결됩니다.

는 (AWS KMS) 대칭AWS KMS keys(다중 리전 KMS 키 포함), 원시 AES-GCM(고급 암호화 표준/갈루아 카운터 모드) 키 및 원시 RSA 키와 같이 AWS Key Management Service 일반적으로 사용되는 여러 래핑 키를 AWS Encryption SDK 지원합니다. 또한 자체 래핑 키를 확장하거나 구현할 수도 있습니다.

봉투 암호화를 사용할 때는 래핑 키를 무단 액세스로부터 보호해야 합니다. 다음 중 한 가지 방법으로 이를 수행할 수 있습니다.

- AWS Key Management Service (AWS KMS)와 같이 이러한 용도로 설계된 웹 서비스를 사용합니다.
- AWS CloudHSM에서 제공하는 것과 같은 하드웨어 보안 모듈(HSM)을 사용합니다.
- 다른 키 관리 도구 및 서비스를 사용합니다.

키 관리 시스템이 없는 경우를 사용하는 것이 좋습니다 AWS KMS. 는 AWS Encryption SDK 와 통합되어 래핑 키를 보호하고 사용하는 AWS KMS 데 도움이 됩니다. 그러나 AWS Encryption SDK 에는 AWS 또는 AWS 서비스가 필요하지 않습니다.

## 키링 및 마스터 키 공급자

암호화 및 복호화에 사용하는 래핑 키를 지정하려면 키링 또는 마스터 키 공급자를 사용합니다. 에서 AWS Encryption SDK 제공하거나 자체 구현을 설계하는 키링 및 마스터 키 공급자를 사용할 수 있습니다. AWS Encryption SDK 는 언어 제약 조건에 따라 서로 호환되는 키링과 마스터 키 공급자를 제공합니다. 자세한 내용은 키링 호환성을 참조하세요.

키링은 데이터 키를 생성, 암호화, 복호화합니다. 키링을 정의할 때 데이터 키를 암호화하는 래핑 키를 지정할 수 있습니다. 대부분의 키링은 하나 이상의 래핑 키를 지정하거나, 래핑 키를 제공하고 보호하는 서비스를 지정합니다. 래핑 키가 없는 키링을 정의하거나 추가 구성 옵션을 사용하여 더 복잡한 키

링을 정의할 수도 있습니다. 에서 AWS Encryption SDK 정의하는 키링을 선택하고 사용하는 데 도움이 필요하면 섹션을 참조하세요키링.

키링은 다음 프로그래밍 언어로 지원됩니다.

- AWS Encryption SDK for C
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK .NET용
- 의 버전 3.x AWS Encryption SDK for Java
- 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
- AWS Encryption SDK for Rust 버전 1.x
- Go AWS Encryption SDK 용의 버전 0.1.x 이상

마스터 키 공급자는 키링의 대안입니다. 마스터 키 공급자는 지정한 래핑 키(또는 마스터 키)를 반환합니다. 각 마스터 키는 하나의 마스터 키 공급자와 연결되지만 마스터 키 공급자는 일반적으로 여러 마스터 키를 제공합니다. 마스터 키 공급자는 Java, Python 및 AWS Encryption CLI에서 지원됩니다.

암호화를 위해 키링(또는 마스터 키 공급자)을 지정해야 합니다. 복호화를 위해 동일한 키링(또는 마스터 키 공급자)을 지정하거나 다른 키링을 지정할 수 있습니다. 암호화할 때 AWS Encryption SDK 는 지정한 모든 래핑 키를 사용하여 데이터 키를 암호화합니다. 복호화할 때 AWS Encryption SDK 는 사용자가 지정한 래핑 키만 사용하여 암호화된 데이터 키를 복호화합니다. 복호화를 위해 래핑 키를 지정하는 것은 선택 사항이지만 AWS Encryption SDK 모범 사례입니다.

래핑 키 지정에 대한 자세한 내용은 래핑 키 선택 섹션을 참조하세요.

## 암호화 컨텍스트

암호화 작업의 보안을 개선하려면 모든 데이터 암호화 요청에 암호화 컨텍스트를 포함시킵니다. 암호화 컨텍스트를 사용하는 것은 선택 사항이지만 권장되는 암호화 모범 사례입니다.

암호화 컨텍스트는 비밀이 아닌 임의의 추가 인증 데이터를 포함하는 키-값 페어 세트입니다. 암호화 컨텍스트에는 사용자가 선택한 모든 데이터가 포함될 수 있지만 일반적으로 파일 유형, 목적 또는 소유권에 대한 데이터와 같이 로깅 및 추적에 유용한 데이터로 구성됩니다. 데이터를 암호화하면 암호화 컨텍스트는 암호화된 데이터에 암호화 방식으로 바인딩되므로 데이터를 복호화할 때 동일한 암호화 컨텍스트가 필요합니다. 또한 AWS Encryption SDK 는 반환하는 암호화된 메시지의 헤더에 암호화 컨텍스트를 일반 텍스트로 포함시킵니다.

에서 AWS Encryption SDK 사용하는 암호화 컨텍스트는 지정한 암호화 컨텍스트와 암호화 자료 관리자(CMM)가 추가하는 퍼블릭 키 페어로 구성됩니다. 특히, 서명이 포함된 암호화 알고리즘을 사용할 때마다 CMM은 예약된 이름(`aws-crypto-public-key`)과 퍼블릭 확인 키를 나타내는 값으로 구성된 이름-값 페어를 암호화 컨텍스트에 추가합니다. 암호화 컨텍스트의 `aws-crypto-public-key` 이름은에 의해 예약 AWS Encryption SDK 되며 암호화 컨텍스트의 다른 페어에서 이름으로 사용할 수 없습니다. 자세한 내용은 메시지 형식 참조의 AAD를 참조하세요.

다음 예제 암호화 컨텍스트는 요청에서 지정된 두 개의 암호화 컨텍스트 페어와 CMM이 추가하는 퍼블릭 키 페어로 구성되어 있습니다.

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

데이터를 복호화하려면 암호화된 메시지를 전달합니다. 는 암호화된 메시지 헤더에서 암호화 컨텍스트를 추출할 AWS Encryption SDK 수 있으므로 암호화 컨텍스트를 별도로 제공할 필요가 없습니다. 하지만 암호화 컨텍스트는 암호화된 메시지를 올바르게 복호화하고 있는지 확인하는 데 도움이 될 수 있습니다.

- AWS Encryption SDK Command Line Interface(CLI)에서 복호화 명령에 암호화 컨텍스트를 제공하는 경우 CLI는 일반 텍스트 데이터를 반환하기 전에 암호화된 메시지의 암호화 컨텍스트에 해당 값이 있는지 확인합니다.
- 다른 프로그래밍 언어 구현의 경우 복호화 응답에 암호화 컨텍스트와 일반 텍스트 데이터가 포함됩니다. 애플리케이션의 복호화 함수는 일반 텍스트 데이터를 반환하기 전에 항상 복호화 응답의 암호화 컨텍스트에 암호화 요청(또는 하위 집합)의 암호화 컨텍스트가 포함되어 있는지 확인해야 합니다.

> **Note**
>
> 다음 버전은 모든 암호화 요청에서 암호화 컨텍스트를 요구하는 데 사용할 수 있는 필수 암호화 컨텍스트 CMM을 지원합니다.
>
> - 의 버전 3.x AWS Encryption SDK for Java
> - for .NET 버전 AWS Encryption SDK 4.x
> - 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
> - AWS Encryption SDK for Rust 버전 1.x
> - Go AWS Encryption SDK 용의 버전 0.1.x 이상

암호화 컨텍스트를 선택할 때는 해당 값이 비밀이 아님을 기억해야 합니다. 암호화 컨텍스트는 AWS Encryption SDK 가 반환하는 암호화된 메시지의 헤더에 일반 텍스트로 표시됩니다. AWS Key Management Service를 사용하는 경우 암호화 컨텍스트는와 같은 감사 레코드 및 로그에 일반 텍스트로 표시될 수도 있습니다 AWS CloudTrail.

코드에서 암호화 컨텍스트를 제출하고 확인하는 예제는 선호하는 프로그래밍 언어의 예제를 참조하세요.

## 암호화된 메시지

를 사용하여 데이터를 암호화하면 암호화된 메시지가 반환 AWS Encryption SDK됩니다.

암호화된 메시지는 암호화된 데이터와 함께 데이터 키의 암호화된 사본, 알고리즘 ID, 선택 사항으로 암호화 컨텍스트와 디지털 서명을 포함하는 이동 가능한 형식화된 데이터 구조입니다. AWS Encryption SDK 의 암호화 작업은 암호화된 메시지를 반환하고 복호화 작업은 암호화된 메시지를 입력으로 사용합니다.

암호화된 데이터와 암호화된 데이터 키를 결합하면 복호화 작업이 간소화되고, 암호화된 데이터 키를 암호화 데이터와 독립적으로 저장하고 관리할 필요가 없습니다.

암호화된 메시지에 대한 기술 정보는 암호화된 메시지 형식을 참조하세요.

## 알고리즘 제품군

는 알고리즘 제품군을 AWS Encryption SDK 사용하여 암호화 및 복호화 작업이 반환하는 암호화된 메시지의 데이터를 암호화하고 서명합니다. AWS Encryption SDK 에서는 여러 알고리즘 제품군을 지원합니다. 지원하는 알고리즘은 모두 AES(Advanced Encryption Standard)를 기본 알고리즘으로 사용하고 이 기본 알고리즘을 다른 알고리즘 및 값과 조합합니다.

는 모든 암호화 작업의 기본값으로 권장 알고리즘 제품군을 AWS Encryption SDK 설정합니다. 기본값은 표준과 모범 사례가 개선됨에 따라 변경될 수 있습니다. 데이터 암호화 요청이나 암호화 구성 요소 관리자(CMM)를 생성할 때 대체 알고리즘 제품군을 지정할 수 있지만 상황에 따라 대안이 필요한 경우가 아니라면 기본값을 사용하는 것이 가장 좋습니다. 현재 기본값은 HMAC 기반 추출 및 확장 키 유도 함수(HKDF), 키 커밋, 타원 곡선 디지털 서명 알고리즘(ECDSA) 서명, 256비트 암호화 키를 갖춘 AES-GCM입니다.

애플리케이션에 고성능이 필요하고 데이터를 암호화하는 사용자와 데이터를 복호화하는 사용자의 신뢰도가 동일하다면 디지털 서명이 없는 알고리즘 제품군을 지정하는 것을 고려할 수 있습니다. 하지만 키 커밋과 키 유도 함수가 포함된 알고리즘 제품군을 사용하는 것을 적극 권장합니다. 이러한 기능이 없는 알고리즘 제품군은 이하 버전과의 호환성을 위해서만 지원됩니다.

# 암호화 구성 요소 관리자

암호화 구성 요소 관리자(CMM)는 데이터를 암호화하고 복호화하는 데 사용되는 암호화 구성 요소를 조합합니다. 암호화 구성 요소에는 일반 텍스트 및 암호화된 데이터 키와 선택 사항인 메시지 서명 키가 포함됩니다. 사용자는 CMM과 직접 상호 작용하지는 않습니다. 암호화 및 복호화 메서드가 이를 대신 처리합니다.

에서 AWS Encryption SDK 제공하는 기본 CMM 또는 캐싱 CMM을 사용하거나 사용자 지정 CMM을 작성할 수 있습니다. 그리고 CMM을 지정할 수 있지만 필수는 아닙니다. 키링 또는 마스터 키 공급자를 지정하면가 기본 CMM을 AWS Encryption SDK 생성합니다. 기본 CMM은 사용자가 지정한 키링 또는 마스터 키 공급자로부터 암호화 또는 복호화 구성 요소를 가져옵니다. 여기에는 AWS Key Management Service(AWS KMS)와 같은 암호화 서비스에 대한 호출이 포함될 수 있습니다.

CMM은 AWS Encryption SDK 와 키링(또는 마스터 키 공급자) 간의 연결 역할을 하기 때문에 정책 적용 및 캐싱 지원과 같은 사용자 지정 및 확장을 위한 이상적인 지점입니다. 는 데이터 키 캐싱을 지원하는 캐싱 CMM을 AWS Encryption SDK 제공합니다. ???

## 대칭 및 비대칭 암호화

대칭 암호화는 동일한 키를 사용하여 데이터를 암호화하고 복호화합니다.

비대칭 암호화는 수학적으로 관련된 데이터 키 페어를 사용합니다. 페어의 키 중 하나가 데이터를 암호화하고, 페어의 다른 키만 데이터를 복호화할 수 있습니다.

는 봉투 암호화를 AWS Encryption SDK 사용합니다. 대칭 데이터 키로 데이터를 암호화합니다. 하나 이상의 대칭 또는 비대칭 래핑 키를 사용하여 대칭 데이터 키를 암호화합니다. 암호화된 데이터와 하나 이상의 데이터 키의 암호화된 사본이 포함된 암호화된 메시지를 반환합니다.

데이터 암호화(대칭 암호화)

데이터를 암호화하기 위해는 대칭 데이터 키와 대칭 암호화 알고리즘이 포함된 알고리즘 제품군을 AWS Encryption SDK 사용합니다. 데이터를 해독하기 위해는 동일한 데이터 키와 동일한 알고리즘 제품군을 AWS Encryption SDK 사용합니다.

데이터 키 암호화(대칭 또는 비대칭 암호화)

암호화 및 복호화 작업에 제공하는 키링 또는 마스터 키 공급자에 따라 대칭 데이터 키가 암호화 및 복호화되는 방식이 결정됩니다. 키링 또는 키링과 같은 대칭 암호화를 사용하는 마스터 AWS KMS 키 공급자 또는 원시 RSA 키링 또는와 같은 비대칭 암호화를 사용하는 키링 또는 마스터 키 공급자를 선택할 수 있습니다`JceMasterKey`.

# 키 커밋

는 각 사이퍼텍스트를 단일 일반 텍스트로만 해독할 수 있도록 보장하는 보안 속성인 키 커밋(강력성이라고도 함)을 AWS Encryption SDK 지원합니다. 이를 위해 키 커밋은 메시지를 암호화한 데이터 키만 복호화에 사용되도록 보장합니다. 키 커밋으로 데이터를 암호화하고 복호화하는 것이 AWS Encryption SDK 모범 사례입니다.

AES를 포함한 대부분의 최신 대칭 암호는 AWS Encryption SDK 가 각 일반 텍스트 메시지를 암호화하는 데 사용하는 고유 데이터 키와 같은 단일 비밀 키로 일반 텍스트를 암호화합니다. 동일한 데이터 키로 이 데이터를 복호화하면 원본과 동일한 일반 텍스트가 반환됩니다. 다른 키를 사용한 복호화는 일반적으로 실패합니다. 하지만 서로 다른 두 개의 키로 사이퍼텍스트를 복호화할 수 있습니다. 드문 경우이긴 하지만, 몇 바이트의 사이퍼텍스트를 다르지만 여전히 식별할 수 있는 일반 텍스트로 복호화할 수 있는 키를 찾는 경우가 있습니다.

는 AWS Encryption SDK 항상 하나의 고유한 데이터 키로 각 일반 텍스트 메시지를 암호화합니다. 여러 래핑 키(또는 마스터 키)로 해당 데이터 키를 암호화할 수 있지만 래핑 키는 항상 동일한 데이터 키를 암호화합니다. 하지만 정교하고 수동으로 조작된 암호화된 메시지에는 실제로 각각 다른 래핑 키로 암호화된 서로 다른 데이터 키가 포함될 수 있습니다. 예를 들어 한 사용자가 암호화된 메시지를 복호화하여 0x0(false)이 반환됐지만 동일한 암호화된 메시지를 다른 사용자가 복호화하면 0x1(true)이 반환될 수 있습니다.

이 시나리오를 방지하기 위해는 암호화 및 복호화 시 키 커밋을 AWS Encryption SDK 지원합니다. 는 키 커밋으로 메시지를 AWS Encryption SDK 암호화할 때 암호화 텍스트를 생성한 고유한 데이터 키를 비밀이 아닌 데이터 키 식별자인 키 커밋 문자열에 암호화 방식으로 바인딩합니다. 그런 다음 암호화된 메시지의 메타데이터에 키 커밋 문자열을 저장합니다. 키 커밋으로 메시지를 복호화하면 AWS Encryption SDK 는 데이터 키가 암호화된 메시지의 유일한 키인지 확인합니다. 데이터 키 확인에 실패하면 복호화 작업이 실패합니다.

키 커밋에 대한 지원은 버전 1.7.x에 도입되었으며, 키 커밋으로 메시지를 복호화할 수는 있지만 키 커밋으로 암호화하지는 않습니다. 이 버전을 사용하면 키 커밋으로 사이퍼텍스트를 복호화하는 기능을 완전히 배포할 수 있습니다. 버전 2.0.x에서는 키 커밋이 완전히 지원됩니다. 기본적으로 키 커밋을 통해서만 암호화하고 복호화합니다. 이는 이전 버전의 로 암호화된 사이퍼텍스트를 해독할 필요가 없는 애플리케이션에 이상적인 구성입니다 AWS Encryption SDK.

키 커밋을 통한 암호화 및 복호화가 모범 사례이기는 하지만, 사용 시기를 직접 결정하고 적용 속도를 조정할 수 있습니다. 버전 1.7.x부터는 기본 알고리즘 제품군을 설정하고 사용할 수 있는 알고리즘 제품군을 제한하는 커밋 정책을 AWS Encryption SDK 지원합니다. 이 정책은 키 커밋으로 데이터를 암호화 및 복호화할지 여부를 결정합니다.

키 커밋을 사용하면 암호화된 메시지 크기가 약간 더 커지며(+ 30바이트) 처리하는 데 시간이 더 걸립니다. 애플리케이션이 크기나 성능에 매우 민감한 경우 키 커밋을 사용하지 않도록 설정할 수 있습니다. 하지만 꼭 필요한 경우에만 이를 설정하세요.

키 커밋 기능을 포함하여 버전 1.7.x 및 2.0.x로 마이그레이션하는 방법에 대한 자세한 내용은 마이그레이션 AWS Encryption SDK 섹션을 참조하세요. 키 커밋에 대한 기술 정보는 the section called “알고리즘 참조” 및 the section called “메시지 형식 참조” 섹션을 참조하세요.

## 커밋 정책

커밋 정책은 애플리케이션이 키 커밋을 사용하여 암호화 및 복호화할지 여부를 결정하는 구성 설정입니다. 키 커밋으로 데이터를 암호화하고 복호화하는 것이 AWS Encryption SDK 모범 사례입니다.

커밋 정책에는 세 가지 값이 있습니다.

> **Note**
>
> 전체 표를 보려면 가로 또는 세로로 스크롤해야 할 수 있습니다.

커밋 정책 값

| 값 | 키 커밋으로 암호화 | 키 커밋 없이 암호화 | 키 커밋으로 복호화 | 키 커밋 없이 복호화 |
|---|---|---|---|---|
| ForbidEncryptAllowDecrypt | ✗ | ✓ | ✓ | ✓ |
| RequireEncryptAllowDecrypt | ✓ | ✗ | ✓ | ✓ |
| RequireEncryptRequireDecrypt | ✓ | ✗ | ✓ | ✗ |

커밋 정책 설정은 AWS Encryption SDK 버전 1.7.x에 도입되었습니다. 지원되는 모든 프로그래밍 언어에서 유효합니다.

- `ForbidEncryptAllowDecrypt`는 키 커밋 유무에 관계없이 복호화하지만, 키 커밋으로 암호화하지는 않습니다. 버전 1.7.x에 도입된이 값은 애플리케이션을 실행하는 모든 호스트가 키 커밋으로 암호화된 사이퍼텍스트를 발견하기 전에 키 커밋으로 복호화할 수 있도록 설계되었습니다.
- `RequireEncryptAllowDecrypt`는 항상 키 커밋으로 암호화합니다. 키 커밋 유무에 관계없이 복호화할 수 있습니다. 이 값은 버전 2.0.x에 도입되었으며, 이를 사용하면 키 커밋으로 암호화를 시작하지만 키 커밋 없이 레거시 사이퍼텍스트를 복호화할 수 있습니다.
- `RequireEncryptRequireDecrypt`는 키 커밋을 통해서만 암호화 및 복호화합니다. 이 값은 버전 2.0.x의 기본값입니다. 모든 사이퍼텍스트가 키 커밋으로 암호화되는 것이 확실한 경우에 이 값을 사용합니다.

커밋 정책 설정에 따라 사용할 수 있는 알고리즘 제품군이 결정됩니다. 버전 1.7.x부터는 서명 유무에 관계없이 키 커밋을 위한 알고리즘 제품군을 AWS Encryption SDK 지원합니다. 커밋 정책과 충돌하는 알고리즘 제품군을 지정하는 경우 AWS Encryption SDK 에서 오류를 반환합니다.

커밋 정책을 설정하는 데 도움이 필요하면 커밋 정책 설정 섹션을 참조하세요.

## 디지털 서명

는 인증된 AWS Encryption SDK 암호화 알고리즘인 AES-GCM을 사용하여 데이터를 암호화하고 복호화 프로세스는 디지털 서명을 사용하지 않고 암호화된 메시지의 무결성과 신뢰성을 확인합니다. 그러나 AES-GCM은 대칭 키를 사용하기 때문에 사이퍼텍스트를 복호화하는 데 사용되는 데이터 키를 복호화할 수 있는 사용자라면 누구나 암호화된 새 사이퍼텍스트를 수동으로 생성할 수 있어 잠재적인 보안 문제가 발생할 수 있습니다. 예를 들어를 래핑 키 AWS KMS key 로 사용하는 경우 `kms:Decrypt` 권한이 있는 사용자는를 호출하지 않고 암호화된 사이퍼텍스트를 생성할 수 있습니다`kms:Encrypt`.

이 문제를 방지하기 위해는 암호화된 메시지 끝에 타원 곡선 디지털 서명 알고리즘(ECDSA) 서명을 추가하는 것을 AWS Encryption SDK 지원합니다. 서명 알고리즘 제품군을 사용하면는 암호화된 각 메시지에 대해 임시 프라이빗 키와 퍼블릭 키 페어를 AWS Encryption SDK 생성합니다. 는 데이터 키의 암호화 컨텍스트에 퍼블릭 키를 AWS Encryption SDK 저장하고 프라이빗 키를 삭제합니다. 이렇게 하면 아무도 퍼블릭 키로 확인하는 다른 서명을 생성할 수 없습니다. 이 알고리즘은 퍼블릭 키를 메시지 헤더의 추가 인증 데이터로 암호화된 데이터 키에 바인딩하여 메시지를 복호화할 수 있는 사용자가 퍼블릭 키를 변경하거나 서명 확인에 영향을 미치지 않도록 합니다.

서명 확인은 복호화 시 상당한 성능 비용을 추가시킵니다. 데이터를 암호화하는 사용자와 데이터를 복호화하는 사용자의 신뢰도가 같으면 서명이 포함되지 않은 알고리즘 제품군을 사용하는 것이 좋습니다.

> **Note**
>
> 래핑 암호화 구성 요소에 대한 키링 또는 액세스가 암호화 도구와 복호화기 간에 구분되지 않는 경우 디지털 서명은 암호화 값을 제공하지 않습니다.

비대칭 RSA AWS KMS 키링을 포함한 AWS KMS 키링은 AWS KMS 키 정책 및 IAM 정책에 따라 암호화 도구와 복호화기 간에 구분할 수 있습니다.

암호화 특성으로 인해 다음 키링은 암호화 도구와 복호화자 간에 구분할 수 없습니다.

- AWS KMS 계층적 키링
- AWS KMS ECDH 키링
- Raw AES 키링
- Raw RSA 키링
- 원시 ECDH 키링

# AWS Encryption SDK 작동 방식

이 섹션의 워크플로에서는 AWS Encryption SDK 가 데이터를 암호화하고 암호화된 메시지를 해독하는 방법을 설명합니다. 이 워크플로는 기본 기능을 사용하는 기본 프로세스를 설명합니다. 사용자 지정 구성 요소 정의 및 사용에 대한 자세한 내용은 지원되는 각 언어 구현의 GitHub 리포지토리를 참조하세요.

는 봉투 암호화를 AWS Encryption SDK 사용하여 데이터를 보호합니다. 각 메시지는 고유한 데이터 키로 암호화됩니다. 그러면 지정한 래핑 키를 사용해 데이터 키가 암호화됩니다. 암호화된 메시지를 해독하기 위해 AWS Encryption SDK 는 지정한 래핑 키를 사용하여 하나 이상의 암호화된 데이터 키를 해독합니다. 그런 다음 사이퍼텍스트를 복호화하여 일반 텍스트 메시지를 반환할 수 있습니다.

AWS Encryption SDK에서 사용하는 용어에 대해 도움이 필요하신가요? the section called “개념”을(를) 참조하세요.

## 가 데이터를 AWS Encryption SDK 암호화하는 방법

는 문자열, 바이트 배열 및 바이트 스트림을 암호화하는 메서드를 AWS Encryption SDK 제공합니다. 코드 예제는 각 프로그래밍 언어 섹션의 예제 항목을 참조하세요.

1. 데이터를 보호하는 래핑 키를 지정하는 키링(또는 마스터 키 공급자)을 생성합니다.

2. 키링 및 일반 텍스트 데이터를 암호화 메서드에 전달합니다. 비밀이 아닌 선택적 암호화 컨텍스트를 전달하는 것이 좋습니다.
3. 암호화 메서드는 키링에 암호화 자료를 요청합니다. 키링은 메시지에 대해 일반 텍스트 데이터 키 하나와 지정된 각 래핑 키로 암호화된 해당 데이터 키의 사본 하나를 반환합니다.
4. 암호화 메서드는 일반 텍스트 데이터 키를 사용하여 데이터를 암호화한 후 일반 텍스트 데이터 키를 삭제합니다. 암호화 컨텍스트를 제공하는 경우( AWS Encryption SDK 모범 사례) 암호화 메서드는 암호화 컨텍스트를 암호화된 데이터에 암호적으로 바인딩합니다.
5. 암호화 메서드는 암호화된 데이터, 암호화된 데이터 키 및 암호화 컨텍스트를 포함한 기타 메타데이터(사용한 경우)가 포함된 암호화된 메시지를 반환합니다.

## 가 암호화된 메시지를 AWS Encryption SDK 복호화하는 방법

는 암호화된 메시지를 복호화하고 일반 텍스트를 반환하는 메서드를 AWS Encryption SDK 제공합니다. 코드 예제는 각 프로그래밍 언어 섹션의 예제 항목을 참조하세요.

암호화된 메시지를 복호화하는 키링(또는 마스터 키 공급자)은 메시지를 암호화하는 데 사용된 것과 호환되어야 합니다. 해당 래핑 키 중 하나가 암호화된 메시지의 암호화된 데이터 키를 복호화할 수 있어야 합니다. 키링 및 마스터 키 공급자와의 호환성에 대한 자세한 내용은 the section called “키링 호환성” 섹션을 참조하세요.

1. 데이터를 복호화할 수 있는 래핑 키를 사용하여 키링 또는 마스터 키 공급자를 생성합니다. 암호화 메서드에 제공한 것과 동일한 키링을 사용하거나 다른 키를 사용할 수 있습니다.
2. 암호화된 메시지와 키링을 복호화 메서드에 전달합니다.
3. 복호화 메서드는 키링 또는 마스터 키 공급자에게 암호화된 메시지의 암호화된 데이터 키 중 하나를 복호화하도록 요청합니다. 암호화된 데이터 키를 포함하여 암호화된 메시지의 정보를 전달합니다.
4. 키링은 해당 래핑 키를 사용하여 암호화된 데이터 키 중 하나를 복호화합니다. 성공하면 응답에 일반 텍스트 데이터 키가 포함됩니다. 키링 또는 마스터 키 공급자가 지정한 래핑 키가 암호화된 데이터 키를 복호화할 수 없는 경우 복호화 호출이 실패합니다.
5. 복호화 메서드는 일반 텍스트 데이터 키를 사용하여 데이터를 복호화하고 일반 텍스트 데이터 키를 삭제하며 일반 텍스트 데이터를 반환합니다.

# 에서 지원되는 알고리즘 제품군 AWS Encryption SDK

알고리즘 제품군은 암호화 알고리즘 및 관련 값의 모음입니다. 암호화 시스템은 알고리즘 구현을 사용하여 사이퍼텍스트를 생성합니다.

AWS Encryption SDK 알고리즘 제품군은 AES-GCM이라고 하는 Galois/Counter Mode(GCM)의 고급 암호화 표준(AES) 알고리즘을 사용하여 원시 데이터를 암호화합니다. 는 256비트, 192비트 및 128비트 암호화 키를 AWS Encryption SDK 지원합니다. 초기화 벡터(IV)의 길이는 항상 12바이트입니다. 인증 태그의 길이는 항상 16바이트입니다.

기본적으로는 HMAC 기반 extract-and-expand 키 유도 함수(HKDF), 서명 및 256비트 암호화 키가 있는 AES-GCM이 있는 알고리즘 제품군을 AWS Encryption SDK 사용합니다. 커밋 정책에 키 커밋이 필요한 경우는 키 커밋도 지원하는 알고리즘 제품군을 AWS Encryption SDK 선택합니다. 그렇지 않으면 키 파생 및 서명이 있지만 키 커밋이 아닌 알고리즘 제품군을 선택합니다.

## 권장: AES-GCM(키 유도. 서명, 키 커밋 포함)

는 HMAC 기반 extract-and-expand 키 유도 함수(HKDF)에 256비트 데이터 암호화 키를 제공하여 AES-GCM 암호화 키를 도출하는 알고리즘 제품군을 AWS Encryption SDK 권장합니다. 는 타원 곡선 디지털 서명 알고리즘(ECDSA) 서명을 AWS Encryption SDK 추가합니다. 키 커밋을 지원하기 위해 이 알고리즘 제품군은 암호화된 메시지의 메타데이터에 저장되는 키 커밋 문자열(비밀이 아닌 데이터 키 식별자)도 유도합니다. 이 키 커밋 문자열도 데이터 암호화 키 유도와 유사한 절차를 사용하여 HKDF를 통해 유도됩니다.

AWS Encryption SDK 알고리즘 제품군

| 암호화 알고리즘 | 데이터 암호화 키 길이(비트) | 키 유도 알고리즘 | 서명 알고리즘 | 키 커밋 |
|---|---|---|---|---|
| AES-GCM | 256 | HKDF(SHA-384 사용) | ECDSA(P-384 및 SHA-384 사용) | HKDF(SHA-512 사용) |

HKDF를 사용하면 실수로 데이터 암호화 키를 재사용하는 것을 방지하고 데이터 키를 과도하게 사용할 위험을 줄일 수 있습니다.

서명을 위해 이 알고리즘 제품군은 암호화 해시 함수 알고리즘(SHA-384)과 함께 ECDSA를 사용합니다. ECDSA는 기본 마스터 키에 대한 정책에 명시되지 않았더라도 기본적으로 사용됩니다. 메시지 서명은 메시지 발신자가 메시지를 암호화할 권한이 있는지 확인하고 부인 방지 기능을 제공합니다. 특히 마스터 키에 대한 권한 부여 정책에서 한 사용자 세트에 대해 데이터 암호화를 허용하고 다른 사용자 세트에 데이터 복호화를 허용하는 경우에 유용합니다.

키 커밋이 포함된 알고리즘 제품군은 각 사이퍼텍스트가 하나의 일반 텍스트로만 복호화되도록 합니다. 이를 위해 암호화 알고리즘에 대한 입력으로 사용된 데이터 키의 자격 증명을 검증합니다. 암호화

할 때 이러한 알고리즘 제품군은 키 커밋 문자열을 유도합니다. 복호화하기 전에 데이터 키가 키 커밋 문자열과 일치하는지 검증합니다. 그러지 않으면 복호화 호출이 실패합니다.

## 기타 지원 알고리즘 제품군

는 이전 버전과의 호환성을 위해 다음과 같은 대체 알고리즘 제품군을 AWS Encryption SDK 지원합니다. 일반적으로 이러한 알고리즘 제품군은 사용하지 않는 것이 좋습니다. 하지만 서명이 성능을 크게 저해할 수 있다는 점을 잘 알고 있기 때문에 이러한 경우를 위해 키 유도가 포함된 키 커밋 제품군이 제공됩니다. 성능 절충을 더 많이 해야 하는 애플리케이션을 위해 서명, 키 커밋 및 키 유도가 없는 제품군이 계속 제공됩니다.

AES-GCM(키 커밋 없음)

키 커밋이 없는 알고리즘 제품군은 복호화하기 전에 데이터 키를 검증하지 않습니다. 따라서 이러한 알고리즘 제품군은 단일 사이퍼텍스트를 다른 일반 텍스트 메시지로 복호화할 수 있습니다. 그러나 키 커밋이 포함된 알고리즘 제품군은 약간 더 큰(+30바이트) 암호화된 메시지를 생성하여 처리 시간이 더 오래 걸리기 때문에 모든 애플리케이션에 가장 적합한 선택은 아닙니다.

는 키 파생, 키 커밋, 서명이 있는 알고리즘 제품군과 키 파생 및 키 커밋이 있지만 서명은 없는 알고리즘 제품군을 AWS Encryption SDK 지원합니다. 키 커밋이 없는 알고리즘 제품군은 사용하지 않는 것이 좋습니다. 꼭 필요한 경우 키 유도 및 키 커밋은 포함되지만 서명은 없는 알고리즘 제품군을 사용하는 것이 좋습니다. 그러나 애플리케이션 성능 프로파일이 알고리즘 제품군 사용을 지원하는 경우 키 커밋, 키 유도 및 서명이 포함된 알고리즘 제품군을 사용하는 것이 모범 사례입니다.

서명 없는 AES-GCM

서명이 없는 알고리즘 모음에는 신뢰성 및 부인 방지를 제공하는 ECDSA 서명이 없습니다. 해당 제품군은 데이터를 암호화하는 사용자와, 데이터를 복호화하는 사용자를 동등하게 신뢰할 수 있는 경우에만 사용하세요.

서명 없는 알고리즘 제품군을 사용하는 경우 키 유도 및 키 커밋이 포함된 알고리즘 제품군을 사용하는 것이 좋습니다.

AES-GCM(키 유도 없음)

키 유도가 없는 알고리즘 제품군은 키 유도 함수를 사용하여 고유 키를 유도하는 대신 데이터 암호화 키를 AES-GCM 암호화 키로 사용합니다. 이 제품군을 사용하여 사이퍼텍스트를 생성하는 것은 권장하지 않지만는 호환성을 위해 이를 AWS Encryption SDK 지원합니다.

이러한 제품군이 라이브러리에서 어떻게 표시되고 사용되는지에 대한 자세한 내용은 the section called “알고리즘 참조” 섹션을 참조하세요.

# 와 AWS Encryption SDK 함께 사용 AWS KMS

를 사용하려면 래핑 키로 키링 또는 마스터 키 공급자를 구성 AWS Encryption SDK해야 합니다. 키 인프라가 없는 경우 AWS Key Management Service (AWS KMS)를 사용하는 것이 좋습니다. 의 많은 코드 예제에는가 AWS Encryption SDK 필요합니다AWS KMS key.

와 상호 작용하려면에 원하는 프로그래밍 언어에 맞는 AWS SDK AWS KMS가 AWS Encryption SDK 필요합니다. AWS Encryption SDK 클라이언트 라이브러리는 AWS SDKs와 함께 작동하여에 저장된 마스터 키를 지원합니다 AWS KMS.

에서를 사용할 준비를 하려면 AWS Encryption SDK AWS KMS

1. 를 생성합니다 AWS 계정. 방법을 알아보려면 AWS 지식 센터의 새 Amazon Web Services 계정을 생성하고 활성화하려면 어떻게 해야 합니까?를 참조하세요.
2. 대칭 암호화를 생성합니다 AWS KMS key. 도움말은 AWS Key Management Service 개발자 가이드의 키 생성을 참조하세요.

   > **Tip**
   >
   > AWS KMS key 프로그래밍 방식으로를 사용하려면의 키 ID 또는 Amazon 리소스 이름(ARN)이 필요합니다 AWS KMS key. AWS KMS key의 ID 및 ARN을 찾으려면 AWS Key Management Service 개발자 가이드의 키 ID 및 ARN 찾기를 참조하세요.

3. 액세스 키 ID와 보안 액세스 키를 생성합니다. IAM 사용자의 액세스 키 ID와 보안 액세스 키를 사용하거나 AWS Security Token Service 를 사용하여 액세스 키 ID, 보안 액세스 키 및 세션 토큰이 포함된 임시 보안 자격 증명으로 새 세션을 생성할 수 있습니다. 보안 모범 사례로 IAM 사용자 또는 AWS (루트) 사용자 계정과 연결된 장기 자격 증명 대신 임시 자격 증명을 사용하는 것이 좋습니다.

   액세스 키를 사용하여 IAM 사용자를 생성하려면 IAM 사용 설명서의 IAM 사용자 생성을 참조하세요.

   임시 보안 자격 증명을 생성하려면 IAM 사용 설명서의 임시 보안 자격 증명 요청을 참조하세요.

4. AWS SDK for Java, AWS SDK for JavaScript AWS SDK for Python (Boto) 또는 AWS SDK for C++ (C의 경우)의 지침과 3단계에서 생성한 액세스 키 ID 및 보안 액세스 키를 사용하여 AWS 자격 증명을 설정합니다. 임시 자격 증명을 생성한 경우 세션 토큰도 지정해야 합니다.

이 절차를 통해 AWS SDKs에 AWS 대한 요청에 서명할 수 있습니다. 와 상호 작용하는의 코드 샘플 AWS Encryption SDK 은이 단계를 완료했다고 AWS KMS 가정합니다.

5. 를 다운로드하여 설치합니다 AWS Encryption SDK. 방법을 알아보려면 사용하려는 프로그래밍 언어의 설치 지침을 참조하세요.

# 모범 사례 AWS Encryption SDK

AWS Encryption SDK 는 업계 표준 및 모범 사례를 사용하여 데이터를 쉽게 보호할 수 있도록 설계되었습니다. 많은 모범 사례가 기본값으로 선택되어 있지만 일부 사례는 필요한 상황이라면 선택적으로 사용을 권장합니다.

최신 버전 사용

사용을 시작할 때 원하는 프로그래밍 언어로 제공되는 최신 버전을 AWS Encryption SDK사용합니다. 를 사용한 경우 가능한 한 빨리 각 최신 버전으로 AWS Encryption SDK업그레이드하세요. 이렇게 하면 권장 구성을 사용하고 새로운 보안 속성을 활용하여 데이터를 보호할 수 있습니다. 마이그레이션 및 배포 지침을 포함하여 지원되는 버전에 대한 자세한 내용은 지원 및 유지 관리 및 의 버전 AWS Encryption SDK 섹션을 참조하세요.

새 버전에서 코드의 요소가 더 이상 사용되지 않는 경우 가능한 한 빨리 해당 요소를 교체합니다. 일반적으로 지원 중단 경고와 코드 주석에서 좋은 대안을 제시해 줍니다.

중요한 업그레이드를 더 쉽게 하고 오류 발생을 줄이기 위해 임시적 또는 일시적으로 릴리스를 제공하는 경우가 있습니다. 이러한 릴리스와 함께 제공되는 설명서를 사용하면 프로덕션 워크플로를 중단하지 않고 애플리케이션을 업그레이드할 수 있습니다.

기본값 사용

는 모범 사례를 기본값으로 AWS Encryption SDK 설계합니다. 가능하면 설정된 기본값을 사용하세요. 기본값이 실용적이지 않은 경우 서명이 없는 알고리즘 제품군과 같은 대안을 제공합니다. 또한 고급 사용자에게도 사용자 지정 키링, 마스터 키 공급자, 암호화 구성 요소 관리자(CMM)와 같은 사용자 지정 기능을 제공합니다. 이러한 고급 대안을 신중하게 사용하고 가능하면 보안 엔지니어의 확인을 받도록 하세요.

암호화 컨텍스트 사용

암호화 작업의 보안을 개선하려면 모든 데이터 암호화 요청에 의미 있는 값을 포함하는 암호화 컨텍스트를 포함시키세요. 암호화 컨텍스트를 사용하는 것은 선택 사항이지만 권장되는 암호화 모범 사례입니다. 암호화 컨텍스트는 AWS Encryption SDK에서 인증된 암호화를 위한 추가 인증 데이터(AAD)를 제공합니다. 비밀은 아니지만 암호화 컨텍스트는 암호화된 데이터의 무결성과 신뢰성을 보호하는 데 도움이 될 수 있습니다.

에서는 암호화할 때만 암호화 컨텍스트를 AWS Encryption SDK지정합니다. 복호화 시는이 AWS Encryption SDK 반환하는 암호화된 메시지의 헤더에 암호화 컨텍스트를 AWS Encryption SDK 사용합니다. 애플리케이션에서 일반 텍스트 데이터를 반환하기 전에, 메시지를 암호화하는 데 사용한

암호화 컨텍스트가 메시지를 복호화하는 데 사용된 암호화 컨텍스트에 포함되어 있는지 확인합니다. 자세한 내용은 사용 중인 프로그래밍 언어의 예를 참조하세요.

명령줄 인터페이스를 사용하면 AWS Encryption SDK 가 암호화 컨텍스트를 확인합니다.

래핑 키 보호

는 고유한 데이터 키를 AWS Encryption SDK 생성하여 각 일반 텍스트 메시지를 암호화합니다. 그런 다음 사용자가 제공한 래핑 키를 사용하여 데이터 키를 암호화합니다. 래핑 키를 분실하거나 삭제하면 암호화된 데이터를 복구할 수 없습니다. 키가 보호되지 않으면 데이터가 취약해질 수 있습니다.

AWS Key Management Service(AWS KMS)와 같이 보안 키 인프라로 보호되는 래핑 키를 사용하세요. 원시 AES 또는 원시 RSA 키를 사용하는 경우 보안 요구 사항에 부합하는 무작위성 소스 및 내구성이 뛰어난 스토리지를 사용하세요. 하드웨어 보안 모듈(HSM) 또는와 같은 HSMs을 제공하는 서비스에 래핑 키를 생성하고 저장하는 것이 모범 사례 AWS CloudHSM입니다.

키 인프라의 인증 메커니즘을 사용하여 래핑 키에 대한 액세스를 필요한 사용자로만 제한하세요. 최소 권한과 같은 모범 사례 원칙을 구현하세요. AWS KMS keys를 사용할 때는 모범 사례 원칙을 구현하는 키 정책 및 IAM 정책을 사용합니다.

래핑 키 지정

암호화할 때뿐만 아니라 복호화할 때도 명시적으로 래핑 키를 지정하는 것이 항상 가장 좋습니다. 이렇게 하면는 지정한 키만 AWS Encryption SDK 사용합니다. 이렇게 하면 의도한 암호화 키만 사용할 수 있습니다. 또한 래핑 키의 경우 AWS KMS 다른 AWS 계정 또는 리전에서 실수로 키를 사용하지 못하게 하거나 사용할 권한이 없는 키로 복호화를 시도하여 성능이 향상됩니다.

암호화할 때 AWS Encryption SDK 공급되는 키링 및 마스터 키 공급자는 래핑 키를 지정해야 합니다. 그리고 사용자가 지정한 래핑 키만 모두 사용합니다. 또한 원시 AES 키링, 원시 RSA 키링, JCEMasterKeys를 사용하여 암호화 및 복호화할 때도 래핑 키를 지정해야 합니다.

그러나 AWS KMS 키링 및 마스터 키 공급자를 사용하여 복호화할 때는 래핑 키를 지정할 필요가 없습니다. 는 암호화된 데이터 키의 메타데이터에서 키 식별자를 가져올 AWS Encryption SDK 수 있습니다. 하지만 래핑 키를 지정하는 것이 권장되는 모범 사례입니다.

AWS KMS 래핑 키로 작업할 때이 모범 사례를 지원하려면 다음을 수행하는 것이 좋습니다.

- 래핑 AWS KMS 키를 지정하는 키링을 사용합니다. 암호화 및 복호화 시 이러한 키링은 사용자가 지정하는 지정된 래핑 키만 사용합니다.
- AWS KMS 마스터 키 및 마스터 키 공급자를 사용하는 경우의 버전 1.7.x에 도입된 엄격 모드 생성자를 사용합니다 AWS Encryption SDK. 지정한 래핑 키로만 암호화하고 복호화하는 공급자를

생성합니다. 항상 모든 래핑 키로 복호화하는 마스터 키 공급자의 생성자는 버전 1.7.x에서 더 이상 사용되지 않으며 버전 2.0.x에서 삭제되었습니다.

복호화를 위한 AWS KMS 래핑 키를 지정하는 것이 실용적이지 않은 경우 검색 공급자를 사용할 수 있습니다. C 및 JavaScript AWS Encryption SDK 의는 AWS KMS 검색 키링을 지원합니다. 검색 모드가 있는 마스터 키 공급자는 버전 1.7.x 이상에서 Java 및 Python으로 사용할 수 있습니다. AWS KMS 래핑 키를 사용한 복호화에만 사용되는 이러한 검색 공급자는 데이터 키를 암호화한 래핑 키를 사용하도록 AWS Encryption SDK 에 명시적으로 지시합니다.

검색 공급자를 사용해야 하는 경우 검색 필터 기능을 사용하여 해당 공급자가 사용하는 래핑 키를 제한합니다. 예를 들어 AWS KMS 리전 검색 키링은 특정 AWS 리전의 래핑 키만 사용합니다. 특정의 래핑 키만 사용하도록 AWS KMS 키링 및 AWS KMS 마스터 키 공급자를 구성할 수도 있습니다 AWS 계정. ??? 또한 항상 그렇듯이 키 정책 및 IAM 정책을 사용하여 AWS KMS 래핑 키에 대한 액세스를 제어합니다.

디지털 서명 사용

서명 기능이 있는 알고리즘 제품군을 사용하는 것이 가장 좋습니다. 디지털 서명은 메시지 발신자가 메시지를 보낼 권한이 있는지 확인하고 메시지의 무결성을 보호합니다. 모든 버전의는 기본적으로 서명이 있는 알고리즘 제품군을 AWS Encryption SDK 사용합니다.

보안 요구 사항에 디지털 서명이 포함되지 않은 경우 디지털 서명이 없는 알고리즘 제품군을 선택할 수 있습니다. 그러나 특히 한 사용자 그룹이 데이터를 암호화하고 다른 사용자 그룹이 해당 데이터를 복호화하는 경우에 디지털 서명을 사용하는 것이 좋습니다.

키 커밋 사용

키 커밋 보안 기능을 사용하는 것이 가장 좋습니다. 키 커밋은 데이터를 암호화한 고유 데이터 키의 ID를 확인함으로써 두 개 이상의 일반 텍스트 메시지를 반환할 수 있는 사이퍼텍스트를 복호화하는 것을 방지합니다.

는 버전 2.0.x부터 키 커밋을 사용하여 암호화 및 복호화에 대한 전체 지원을 AWS Encryption SDK 제공합니다. 기본적으로 모든 메시지는 키 커밋을 통해 암호화되고 복호화됩니다. 의 버전 1.7.x는 키 커밋으로 사이퍼텍스트를 복호화할 AWS Encryption SDK 수 있습니다. 이 버전은 이하 버전 사용자가 버전 2.0.x를 성공적으로 배포하는 데 도움이 되도록 설계되었습니다.

키 커밋에 대한 지원에는 새로운 알고리즘 제품군과 키 커밋이 없는 사이퍼텍스트보다 단 30바이트 더 큰 사이퍼텍스트를 생성하는 새로운 메시지 형식이 포함됩니다. 이 설계는 성능에 미치는 영향을 최소화하여 대부분의 사용자가 키 커밋의 이점을 누릴 수 있도록 했습니다. 애플리케이션이 크기와 성능에 매우 민감한 경우 커밋 정책 설정을 사용하여 키 커밋을 비활성화하거나가 커밋 없이

메시지를 복호화 AWS Encryption SDK 하도록 허용할 수 있지만 그렇게 해야 하는 경우에만 가능합니다.

암호화된 데이터 키의 수 제한

복호화하는 메시지, 특히 신뢰할 수 없는 출처에서 온 메시지의 암호화된 데이터 키의 수를 제한하는 것이 좋습니다. 암호를 복호화할 수 없는 수많은 암호화된 데이터 키가 포함된 메시지를 복호화하면 지연 시간이 길어지고, 비용이 늘어나며, 계정을 공유하는 타사 및 애플리케이션의 성능이 저하되고, 키 인프라가 고갈될 가능성이 있습니다. 제한이 없을 경우, 암호화된 메시지는 최대 65,535(2^16 - 1)개의 암호화된 데이터 키를 보유할 수 있습니다. 자세한 내용은 암호화된 데이터 키 제한을 참조하세요.

이러한 모범 사례의 기반이 되는 AWS Encryption SDK 보안 기능에 대한 자세한 내용은 AWS 보안 블로그의 향상된 클라이언트 측 암호화: 명시적 KeyIds 및 키 커밋을 참조하세요.

# 구성 AWS Encryption SDK

AWS Encryption SDK 는 사용하기 쉽도록 설계되었습니다. AWS Encryption SDK 에는 여러 구성 옵션이 있지만 기본값은 대부분의 애플리케이션에서 실용적이고 안전하도록 신중하게 선택됩니다. 하지만 성능을 개선하거나 사용자 지정 기능을 포함하여 설계하려면 구성을 조정해야 할 수도 있습니다.

구현을 구성할 때 AWS Encryption SDK 모범 사례를 검토하고 최대한 많이 구현하세요.

주제



## 프로그래밍 언어 선택

AWS Encryption SDK 는 여러 프로그래밍 언어로 제공됩니다. 언어 구현은 서로 다른 방식으로 구현될 수 있지만 완전히 상호 연동되고 동일한 기능을 제공하도록 설계되었습니다. 일반적으로 애플리케이션과 호환되는 라이브러리를 사용합니다. 하지만 특정 구현을 위한 프로그래밍 언어를 선택할 수도 있습니다. 예를 들어 키링으로 작업하려는 경우 AWS Encryption SDK for C 또는를 선택할 수 있습니다 AWS Encryption SDK for JavaScript.

## 래핑 키 선택

는 고유한 대칭 데이터 키를 AWS Encryption SDK 생성하여 각 메시지를 암호화합니다. 데이터 키 캐싱을 사용하지 않는 한 데이터 키를 구성하거나, 관리하거나 사용할 필요가 없습니다. 에서 AWS Encryption SDK 자동으로 수행합니다.

하지만 각 데이터 키를 암호화하려면 래핑 키를 하나 이상 선택해야 합니다. AWS Encryption SDK 는 다양한 크기의 AES 대칭 키 및 RSA 비대칭 키를 지원합니다. 또한 AWS Key Management Service(AWS KMS) 대칭 암호화 AWS KMS keys도 지원합니다. 래핑 키의 안전과 내구성은 사용자의 책임입니다. 따라서 하드웨어 보안 모듈 또는와 같은 키 인프라 서비스에서 암호화 키를 사용하는 것이 좋습니다 AWS KMS.

암호화 및 복호화를 위한 래핑 키를 지정하려면 키링(C, Java, JavaScript, .NET 및 Python) 또는 마스터 키 공급자(Java, Python, AWS Encryption CLI)를 사용합니다. 하나의 래핑 키를 지정하거나, 같거나 다른 유형의 여러 래핑 키를 지정할 수 있습니다. 여러 래핑 키를 사용하여 데이터 키를 래핑하는 경우 각 래핑 키는 동일한 데이터 키의 사본을 암호화합니다. 암호화된 데이터 키(래핑 키당 하나)는가 AWS Encryption SDK 반환하는 암호화된 메시지에 암호화된 데이터와 함께 저장됩니다. 데이터를 복호화하려면가 먼저 래핑 키 중 하나를 사용하여 암호화된 데이터 키를 복호화해야 AWS Encryption SDK 합니다.

키링 또는 마스터 키 공급자 AWS KMS key 에서를 지정하려면 지원되는 AWS KMS 키 식별자를 사용합니다. 키의 키 식별자에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 키 식별자를 AWS KMS 참조하세요.

- AWS Encryption SDK for Java AWS Encryption SDK for JavaScript AWS Encryption SDK for Python또는 AWS 암호화 CLI를 사용하여 암호화할 때 KMS 키에 유효한 키 식별자(키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN)를 사용할 수 있습니다. 로 암호화 AWS Encryption SDK for C할 때는 키 ID 또는 키 ARN만 사용할 수 있습니다.

  암호화할 때 KMS 키의 별칭 이름 또는 별칭 ARN을 지정하면 AWS Encryption SDK 는 해당 별칭과 현재 연결된 키 ARN을 저장하지만 별칭은 저장하지 않습니다. 별칭을 변경해도 데이터 키를 복호화하는 데 사용되는 KMS 키에는 영향을 주지 않습니다.
- 엄격 모드(특정 래핑 키를 지정하는 경우)에서 복호화할 때는 키 ARN을 사용하여 AWS KMS keys를 식별해야 합니다. 이 요구 사항은 AWS Encryption SDK의 모든 언어 구현에 적용됩니다.

  AWS KMS 키링으로 암호화하면는 암호화된 데이터 키의 메타데이터 AWS KMS key 에의 키 ARN 을 AWS Encryption SDK 저장합니다. 엄격 모드에서 복호화할 때 AWS Encryption SDK 는 래핑 키를 사용하여 암호화된 데이터 키를 복호화하기 전에 키링(또는 마스터 키 공급자)에 동일한 키 ARN 이 나타나는지 확인합니다. 다른 키 식별자를 사용하는 경우 식별자가 동일한 키를 참조 AWS KMS key하더라도 AWS Encryption SDK 는를 인식하거나 사용하지 않습니다.

원시 AES 키 또는 원시 RSA 키 페어를 키링의 래핑 키로 지정하려면 네임스페이스와 이름을 지정해야 합니다. 마스터 키 공급자에서 `Provider ID`는 네임스페이스에 해당하고 `Key ID`는 이름에 해당합니다. 복호화할 때는 암호화할 때 사용한 것과 정확히 동일한 네임스페이스와 이름을 각 원시 래핑 키에

사용해야 합니다. 다른 네임스페이스 또는 이름을 사용하는 경우 키 구성 요소가 같더라도는 래핑 키를 인식하거나 사용하지 AWS Encryption SDK 않습니다.

# 다중 리전 사용 AWS KMS keys

에서 AWS Key Management Service (AWS KMS) 다중 리전 키를 래핑 키로 사용할 수 있습니다 AWS Encryption SDK. 하나의에서 다중 리전 키로 암호화하는 경우 다른에서 관련 다중 리전 키를 사용하여 복호화할 AWS 리전수 있습니다 AWS 리전. 다중 리전 키에 대한 지원은의 버전 2.3.x AWS Encryption SDK 및 AWS Encryption CLI의 버전 3.0.x에 도입되었습니다.

AWS KMS 다중 리전 키는 키 구성 요소와 키 ID AWS 리전 가 동일한 서로 다른 AWS KMS keys 의 집합입니다. 이러한 관련 키를 다른 리전에서 마치 동일한 키인 것처럼 사용할 수 있습니다. 다중 리전 키는 리전 간 호출 없이 한 리전에서 암호화하고 다른 리전에서 복호화해야 하는 일반적인 재해 복구 및 백업 시나리오를 지원합니다 AWS KMS. 다중 리전 키에 대한 자세한 내용은 AWS Key Management Service 개발자 가이드의 다중 리전 키 사용을 참조하세요.

다중 리전 키를 지원하기 위해 에는 AWS KMS multi-Region-aware 키링과 마스터 키 공급자가 AWS Encryption SDK 포함됩니다. 각 프로그래밍 언어의 새로운 다중 리전 인식 기호는 단일 리전 키와 다중 리전 키를 모두 지원합니다.

- 단일 리전 키의 경우 다중 리전 인식 기호는 단일 리전 AWS KMS 키링 및 마스터 키 공급자처럼 작동합니다. 데이터를 암호화한 단일 리전 키로만 사이퍼텍스트 복호화를 시도합니다.
- 다중 리전 키의 경우 multi-Region-aware 기호는 데이터를 암호화한 것과 동일한 다중 리전 키 또는 지정한 리전의 관련 다중 리전 복제본 키를 사용하여 사이퍼텍스트를 복호화하려고 시도합니다.

KMS 키를 두 개 이상 사용하는 다중 리전 인식 키링 및 마스터 키 공급자에서는 단일 리전 및 다중 리전 키를 여러 개 지정할 수 있습니다. 그러나 각 관련 다중 리전 복제본 키 세트에서 하나의 키만 지정할 수 있습니다. 키 ID가 같은 키 식별자를 두 개 이상 지정하면 생성자 호출이 실패합니다.

표준 단일 리전 키링 및 마스터 키 공급자와 함께 다중 리전 AWS KMS 키를 사용할 수도 있습니다. 하지만 암호화하고 복호화하려면 동일한 리전에서 동일한 다중 리전 키를 사용해야 합니다. 단일 리전 키링 및 마스터 키 공급자는 데이터를 암호화한 키로만 사이퍼텍스트 복호화를 시도합니다.

다음 예제는 다중 리전 키와 새로운 다중 리전 인식 키링 및 마스터 키 공급자를 사용하여 데이터를 암호화하고 복호화하는 방법을 보여줍니다. 이 예제에서는 `us-east-1` 리전의 데이터를 암호화하고 각 `us-west-2` 리전의 관련 다중 리전 복제본 키를 사용하여 리전의 데이터를 복호화합니다. 이 예제를 실행하기 전에 예제 다중 리전 키 ARN을 AWS 계정의 유효한 값으로 바꿉니다.

C

다중 리전 키로 암호화하려면 `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` 메서드를 사용하여 키링을 인스턴스화합니다. 다중 리전 키를 지정합니다.

이 간단한 예제에는 암호화 컨텍스트가 포함되어 있지 않습니다. C에서 암호화 컨텍스트를 사용하는 예제는 문자열 암호화 및 복호화 섹션을 참조하세요.

전체 예제는 GitHub의 AWS Encryption SDK for C 리포지토리에서 kms_multi_region_keys.cpp를 참조하세요.

```
/* 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);

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

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

C# / .NET

미국 동부(버지니아 북부)(us-east-1) 리전에서 다중 리전 키로 암호화하려면 다중 리전 키의 키 식별자와 지정된 리전의 AWS KMS 클라이언트가 있는 `CreateAwsKmsMrkKeyringInput` 객체를 인스턴스화합니다. 그런 다음 `CreateAwsKmsMrkKeyring()` 메서드를 사용하여 키링을 생성합니다.

`CreateAwsKmsMrkKeyring()` 메서드는 정확히 하나의 다중 리전 키로 키링을 생성합니다. 다중 리전 키를 비롯한 여러 래핑 키로 암호화하려면 `CreateAwsKmsMrkMultiKeyring()` 메서드를 사용합니다.

전체 예제는 GitHub의 AWS Encryption SDK for .NET 리포지토리에서 AwsKmsMrkKeyringExample.cs를 참조하세요.

```
//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.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

이 예제에서는 us-east-1 리전의 다중 리전 키를 사용하여 `hello.txt` 파일을 암호화합니다. 이 예제에서는 리전 요소를 사용하여 키 ARN을 지정하므로 이 예제에서는 `--wrapping-keys` 파라미터의 region 속성을 사용하지 않습니다.

래핑 키의 키 ID가 리전을 지정하지 않는 경우 `--wrapping-keys key=$keyID region=us-east-1`과 같은 `--wrapping-keys`의 region 속성을 사용하여 리전을 지정할 수 있습니다.

```
# 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

다중 리전 키를 사용하여 암호화하려면 `AwsKmsMrkAwareMasterKeyProvider`를 인스턴스화하고 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 BasicMultiRegionKeyEncryptionExample.java AWS Encryption SDK for Java 리포지토리에서 섹션을 참조하세요.

```
//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();
```

### JavaScript Browser

다중 리전 키로 암호화하려면 `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` 메서드를 사용하여 키링을 만들고 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_multi_region_simple.ts를 참조하세요.

```
/* 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

다중 리전 키로 암호화하려면 `buildAwsKmsMrkAwareStrictMultiKeyringNode()` 메서드를 사용하여 키링을 만들고 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_multi_region_simple.ts를 참조하세요.

```
//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

AWS KMS 다중 리전 키로 암호화하려면 `MRKAwareStrictAwsKmsMasterKeyProvider()` 메서드를 사용하고 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for Python 리포지토리에서 mrk_aware_kms_provider.py를 참조하세요.

```
* 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_R

# 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"
    }

# Encrypt your plaintext data
ciphertext, encrypt_header = client.encrypt(
        source=source_plaintext,
        encryption_context=encryption_context,
        key_provider=strict_mrk_key_provider
)
```

다음으로, 사이퍼텍스트를 us-west-2 리전으로 이동시킵니다. 사이퍼텍스트를 다시 암호화할 필요는 없습니다.

us-west-2 리전에서 엄격 모드로 사이퍼텍스트를 복호화하려면 us-west-2 리전에서 관련 다중 리전 키의 키 ARN을 사용하여 다중 리전 인식 기호를 인스턴스화합니다. 다른 리전(암호화된 us-

east-1 포함)에 있는 관련 다중 리전 키의 키 ARN을 지정하면 다중 리전 인식 기호가 해당 AWS KMS key에 대하여 리전 간 호출을 수행합니다.

엄격 모드에서 복호화할 때 다중 리전 인식 기호에는 키 ARN이 필요합니다. 여기에서는 관련 다중 리전 키의 각 집합에서 하나의 키 ARN만 허용합니다.

이 예제를 실행하기 전에 예제 다중 리전 키 ARN을의 유효한 값으로 바꿉니다 AWS 계정.

C

다중 리전 키를 사용하여 엄격 모드에서 복호화하려면 `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` 메서드를 사용하여 키링을 인스턴스화합니다. 로컬(us-west-2) 리전에서 관련 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for C 리포지토리에서 kms_multi_region_keys.cpp를 참조하세요.

```
/* 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

단일 다중 리전 키를 사용하여 엄격 모드에서 복호화하려면 입력을 결합하는 데 사용하고 암호화를 위한 키링을 만드는 데 사용한 것과 동일한 생성자와 메서드를 사용합니다. 관련 다중 리전 키의 키 ARN과 미국 서부(오레곤)(us-west-2) 리전의 AWS KMS 클라이언트를 사용하여 `CreateAwsKmsMrkKeyringInput` 객체를 인스턴스화합니다. 그런 다음 `CreateAwsKmsMrkKeyring()` 메서드를 사용하여 하나의 다중 리전 KMS 키로 다중 리전 키링을 생성합니다.

전체 예제는 GitHub의 AWS Encryption SDK for .NET 리포지토리에서 AwsKmsMrkKeyringExample.cs를 참조하세요.

```
// 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

us-west-2 리전에서 관련 다중 리전 키를 사용하여 복호화하려면 `--wrapping-keys` 파라미터의 key 속성을 사용하여 키 ARN을 지정합니다.

```
# 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 \
                     --buffer \
                     --output .
```

Java

엄격 모드에서 복호화하려면 `AwsKmsMrkAwareMasterKeyProvider`를 인스턴스화하고 로컬 (us-west-2) 리전에서 관련 다중 리전 키를 지정합니다.

전체 예제를 보려면 GitHub의 AWS Encryption SDK for Java 리포지토리에서 BasicMultiRegionKeyEncryptionExample.java를 참조하세요.

```
// 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

엄격 모드에서 복호화하려면 `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` 메서드를 사용하여 키링을 만들고 로컬(us-west-2) 리전에서 관련 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_multi_region_simple.ts를 참조하세요.

```
/* 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)
```

JavaScript Node.js

엄격 모드에서 복호화하려면 `buildAwsKmsMrkAwareStrictMultiKeyringNode()` 메서드를 사용하여 키링을 만들고 로컬(us-west-2) 리전에서 관련 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_multi_region_simple.ts를 참조하세요.

```
/* 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'
```

```
 * Specify a multi-Region key in us-west-2
 */
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

엄격 모드에서 복호화하려면 `MRKAwareStrictAwsKmsMasterKeyProvider()` 메서드를 사용하여 마스터 키 공급자를 생성합니다. 로컬(us-west-2) 리전에서 관련 다중 리전 키를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for Python 리포지토리에서 mrk_aware_kms_provider.py를 참조하세요.

```
# 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_R

# 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
)
```

AWS KMS 다중 리전 키를 사용하여 검색 모드에서 복호화할 수도 있습니다. 검색 모드에서 복호화할 때는 어떤 AWS KMS keys도 지정하지 않습니다. (단일 리전 AWS KMS 검색 키링에 대한 자세한 내용은 섹션을 참조하세요AWS KMS 검색 키링 사용.)

다중 리전 키로 암호화한 경우 검색 모드에서 다중 리전 인식 기호는 로컬 리전의 관련 다중 리전 키를 사용하여 복호화를 시도합니다. 존재하지 않는 경우 호출이 실패합니다. 검색 모드에서 AWS Encryption SDK 는 암호화에 사용되는 다중 리전 키에 대한 교차 리전 호출을 시도하지 않습니다.

> **Note**
>
> 검색 모드에서 다중 리전 인식 기호를 사용하여 데이터를 암호화하는 경우 암호화 작업이 실패합니다.

다음 예제는 검색 모드에서 다중 리전 인식 기호를 사용하여 복호화하는 방법을 보여줍니다. 를 지정하지 않으므로 AWS KMS key는 다른 소스에서 리전을 가져와 AWS Encryption SDK 야 합니다. 가능하면 로컬 리전을 명시적으로 지정하세요. 그렇지 않으면는 프로그래밍 언어의 AWS SDK에 구성된 리전에서 로컬 리전을 AWS Encryption SDK 가져옵니다.

이 예제를 실행하기 전에 예제 계정 ID와 다중 리전 키 ARN을의 유효한 값으로 바꿉니다 AWS 계정.

C

다중 리전 키를 사용하여 검색 모드에서 복호화하려면 `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` 메서드를 사용하여 키링을 빌드하고 `Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder()` 메서드를 사용하여 검색 필터를 빌드합니다. 로컬 리전을 지정하려면 `ClientConfiguration`을 정의하고 AWS KMS 클라이언트에서 이를 지정합니다.

전체 예제는 GitHub의 AWS Encryption SDK for C 리포지토리에서 kms_multi_region_keys.cpp를 참조하세요.

```
/* 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).Buil

/* 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)
        .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

AWS Encryption SDK for .NET에서 multi-Region-aware 검색 키링을 생성하려면 특정에 대해 AWS KMS 클라이언트를 가져오는 `CreateAwsKmsMrkDiscoveryKeyringInput` 객체와 KMS 키를 특정 AWS 파티션 AWS 리전및 계정으로 제한하는 선택적 검색 필터를 인스턴스화합니다. 그런 다음 입력 객체로 `CreateAwsKmsMrkDiscoveryKeyring()` 메

서드를 호출합니다. 전체 예제는 GitHub의 AWS Encryption SDK for .NET 리포지토리에서 AwsKmsMrkDiscoveryKeyringExample.cs를 참조하세요.

둘 이상의 AWS 리전에 대해 다중 리전 인식 검색 키링을 만들려면 `CreateAwsKmsMrkDiscoveryMultiKeyring()` 메서드를 사용하여 다중 키링을 만들거나, `CreateAwsKmsMrkDiscoveryKeyring()`을 사용하여 다중 리전 인식 검색 키링을 여러 개 만든 다음 `CreateMultiKeyring()` 메서드를 사용하여 하나의 다중 키링으로 결합합니다.

예제는 AwsKmsMrkDiscoveryMultiKeyringExample.cs를 참조하세요.

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

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

 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

검색 모드에서 복호화하려면 `--wrapping-keys` 파라미터의 discovery 속성을 사용합니다. discovery-account 및 discovery-partition 속성은 선택 사항이지만 권장되는 사항입니다.

리전을 지정하려면 이 명령에 `--wrapping-keys` 파라미터의 region 속성이 포함되어야 합니다.

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

$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys discovery=true \
                                     discovery-account=111122223333 \
                                     discovery-partition=aws \
                                     region=us-west-2 \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 1 \
                     --buffer \
                     --output .
```

Java

로컬 리전을 지정하려면 `builder().withDiscoveryMrkRegion` 파라미터를 사용합니다. 그러지 않으면 AWS Encryption SDK 는 AWS SDK for Java에서 구성된 리전으로부터 로컬 리전을 가져옵니다.

전체 예제를 보려면 GitHub의 AWS Encryption SDK for Java 리포지토리에서 DiscoveryMultiRegionDecryptionExample.java를 참조하세요.

```
// 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", 111122223333);

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

대칭형 다중 리전 키를 사용하여 검색 모드에서 복호화하려면 `AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser()` 메서드를 사용합니다.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_multi_region_discovery.ts를 참조하세요.

```
/* 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)
```

JavaScript Node.js

대칭형 다중 리전 키를 사용하여 검색 모드에서 복호화하려면 `AwsKmsMrkAwareSymmetricDiscoveryKeyringNode()` 메서드를 사용합니다.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_multi_region_discovery.ts를 참조하세요.

```
/* 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

다중 리전 키를 사용하여 검색 모드에서 복호화하려면 `MRKAwareDiscoveryAwsKmsMasterKeyProvider()` 메서드를 사용합니다.

전체 예제는 GitHub의 AWS Encryption SDK for Python 리포지토리에서 mrk_aware_kms_provider.py를 참조하세요.

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

# Instantiate the client
client = aws_encryption_sdk.EncryptionSDKClient()

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

# Use the multi-Region method to create the master key provider
# in discovery mode
mrk_discovery_key_provider =
 MRKAwareDiscoveryAwsKmsMasterKeyProvider(**decrypt_kwargs)

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

# 알고리즘 제품군 선택

는 지정한 래핑 키로 데이터 키를 암호화하기 위해 여러 대칭 및 비대칭 암호화 알고리즘을 AWS Encryption SDK 지원합니다. 그러나 이러한 데이터 키를 사용하여 데이터를 암호화하는 경우는 AWS Encryption SDK 기본적으로 키 파생, 디지털 서명 및 키 커밋과 함께 AES-GCM 알고리즘을 사용하는 권장 알고리즘 제품군으로 설정됩니다. 기본 알고리즘 제품군이 대부분의 애플리케이션에 적합할 가능성이 높지만 대체 알고리즘 제품군을 선택할 수도 있습니다. 예를 들어, 일부 신뢰 모델은 디지털 서

명이 없는 알고리즘 제품군으로 충분할 수 있습니다. AWS Encryption SDK 가 지원하는 알고리즘 제품군에 대한 자세한 내용은 에서 지원되는 알고리즘 제품군 AWS Encryption SDK 섹션을 참조하세요.

다음 예제에서는 암호화 시 대체 알고리즘 제품군을 선택하는 방법을 보여줍니다. 이 예제에서는 키 유도 및 키 커밋은 있지만 디지털 서명은 없는 권장 AES-GCM 알고리즘 제품군을 선택합니다. 디지털 서명이 포함되지 않은 알고리즘 제품군으로 암호화하는 경우 복호화할 때 무서명 전용 복호화 모드를 사용합니다. 이 모드는 서명된 사이퍼텍스트가 발견되면 실패하는 모드로, 스트리밍 복호화 시 가장 유용합니다.

C

에서 대체 알고리즘 제품군을 지정하려면 CMM을 명시적으로 생성 AWS Encryption SDK for C해야 합니다. 그런 다음 `aws_cryptosdk_default_cmm_set_alg_id`를 CMM 및 선택한 알고리즘 제품군과 함께 사용합니다.

```
/* 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_cmm_release(cmm);

/* 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);
    return AWS_OP_ERR;
}
```

디지털 서명 없이 암호화된 데이터를 복호화할 때는 AWS_CRYPTOSDK_DECRYPT_UNSIGNED를 사용합니다. 그러면 서명된 사이퍼텍스트가 발견된 경우 복호화가 실패합니다.

```
/* 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(alloc, 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);
    return AWS_OP_ERR;
}
```

C# / .NET

AWS Encryption SDK for .NET에서 대체 알고리즘 제품군을 지정하려면 EncryptInput 객체의 `AlgorithmSuiteId` 속성을 지정합니다. AWS Encryption SDK for .NET에는 선호하는 알고리즘 제품군을 식별하는 데 사용할 수 있는 상수가 포함되어 있습니다.

AWS Encryption SDK for .NET에는 복호화를 스트리밍할 때 서명된 사이퍼텍스트를 감지하는 방법이 없습니다.이 라이브러리는 스트리밍 데이터를 지원하지 않기 때문입니다.

```
// 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

이 예제에서는 `hello.txt` 파일을 암호화할 때 `--algorithm` 파라미터를 사용하여 디지털 서명이 없는 알고리즘 제품군을 지정합니다.

```
# Specify an algorithm suite without signing

# 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 --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
```

이 예제에서는 복호화할 때 `--decrypt-unsigned` 파라미터를 사용합니다. 이 파라미터는 특히 항상 입력과 출력을 스트리밍하는 CLI를 사용하여 서명되지 않은 사이퍼텍스트를 복호화할 때 사용하는 것이 좋습니다.

```
# 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

대체 알고리즘 제품군을 지정하려면 `AwsCrypto.builder().withEncryptionAlgorithm()` 메서드를 사용합니다. 이 예제에서는 디지털 서명이 없는 대체 알고리즘 제품군을 지정합니다.

```
// Specify an algorithm suite without signing

// Instantiate the client
AwsCrypto crypto = AwsCrypto.builder()
    .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();
```

복호화를 위해 데이터를 스트리밍할 때는 `createUnsignedMessageDecryptingStream()` 메서드를 사용하여 복호화하는 모든 사이퍼텍스트가 서명되지 않았는지 확인합니다.

```
// 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

대체 알고리즘 제품군을 지정하려면 `suiteId` 파라미터를 `AlgorithmSuiteIdentifier` 열거형 값과 함께 사용합니다.

```
// 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, })
```

복호화할 때는 표준 `decrypt` 메서드를 사용합니다. 브라우저가 스트리밍을 지원하지 않기 때문에 브라우저의 AWS Encryption SDK for JavaScript 에는 `decrypt-unsigned` 모드가 없습니다.

```
// 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

대체 알고리즘 제품군을 지정하려면 `suiteId` 파라미터를 `AlgorithmSuiteIdentifier` 열거형 값과 함께 사용합니다.

```
// 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, })
```

디지털 서명 없이 암호화된 데이터를 복호화할 때는 decryptUnsignedMessageStream을 사용합니다. 서명된 사이퍼텍스트가 발견되면 이 메서드는 실패합니다.

```
// 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

대체 암호화 알고리즘을 지정하려면 `algorithm` 파라미터를 `Algorithm` 열거형 값과 함께 사용합니다.

```
# Specify an algorithm suite without signing

# Instantiate a client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R
                                                    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
)
```

디지털 서명 없이 암호화된 메시지를 복호화할 때, 특히 스트리밍 중에 복호화할 때는 `decrypt-unsigned` 스트리밍 모드를 사용합니다.

```
# Decrypt unsigned streaming data

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

# Decrypt with decrypt-unsigned
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
```

Rust

AWS Encryption SDK for Rust에서 대체 알고리즘 제품군을 지정하려면 암호화 요청에서 `algorithm_suite_id` 속성을 지정합니다.

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "AES_256_012";

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
```

```
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create Raw AES keyring
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

// Encrypt your plaintext data
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(raw_aes_keyring.clone())
    .encryption_context(encryption_context.clone())
    .algorithm_suite_id(AlgAes256GcmHkdfSha512CommitKey)
    .send()
    .await?;
```

## Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)
```

```
// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "AES_256_012"

// Optional: Create an encryption context
encryptionContext := map[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 material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create Raw AES keyring
aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  key,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
if err != nil {
    panic(err)
}

// Encrypt your plaintext data
algorithmSuiteId := mpltypes.ESDKAlgorithmSuiteIdAlgAes256GcmHkdfSha512CommitKey
res, err := encryptionClient.Encrypt(context.Background(), esdktypes.EncryptInput{
    Plaintext:         []byte(exampleText),
    EncryptionContext: encryptionContext,
    Keyring:           aesKeyring,
```

```
    AlgorithmSuiteId:  &algorithmSuiteId,
})
if err != nil {
    panic(err)
}
```

# 암호화된 데이터 키 제한

암호화된 메시지의 암호화된 데이터 키의 최대 수를 제한할 수 있습니다. 이 모범 사례 기능을 사용하면 암호화할 때 잘못 구성된 키링을 탐지하거나 복호화할 때 악성 사이퍼텍스트를 탐지할 수 있습니다. 이렇게 하면 키 인프라에 대하여 불필요하며 비용이 높고 잠재적으로 소모적인 호출 또한 방지합니다. 암호화된 데이터 키를 제한하는 것은 신뢰할 수 없는 소스의 메시지를 복호화할 때 가장 유용합니다.

대부분의 암호화된 메시지에는 암호화에 사용되는 래핑 키마다 암호화된 데이터 키가 하나씩 있지만 암호화된 메시지에는 최대 65,535개의 암호화된 데이터 키가 포함될 수 있습니다. 악의적인 공격자는 수천 개의 암호화된 데이터 키를 사용하여 암호화된 메시지를 구성할 수 있지만 이들 중 어느 것도 복호화될 수 없습니다. 따라서 AWS Encryption SDK 는 메시지에서 암호화된 데이터 키를 소진할 때까지 암호화된 각 데이터 키를 복호화하려고 시도합니다.

암호화된 데이터 키를 제한하려면 `MaxEncryptedDataKeys` 파라미터를 사용합니다. 이 파라미터는 AWS Encryption SDK버전 1.9.x 및 2.2.x부터 지원되는 모든 프로그래밍 언어에서 사용할 수 있습니다. 이는 선택 사항이며 암호화 및 복호화 시 유효합니다. 다음 예제에서는 세 가지의 서로 다른 래핑 키로 암호화된 데이터를 복호화합니다. `MaxEncryptedDataKeys` 값은 3으로 설정되어 있습니다.

C

```
/* 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

AWS Encryption SDK for .NET에서 암호화된 데이터 키를 제한하려면 AWS Encryption SDK for .NET에 대한 클라이언트를 인스턴스화하고 선택적 `MaxEncryptedDataKeys` 파라미터를 원하는 값으로 설정합니다. 그런 다음 구성된 AWS Encryption SDK 인스턴스에서 `Decrypt()` 메서드를 호출합니다.

```
// 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],
})

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

### Python

```
# Instantiate a client with limited encrypted data keys
client = aws_encryption_sdk.EncryptionSDKClient(max_encrypted_data_keys=3)

# Create an AWS KMS master key provider
master_key_provider = aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(
    key_ids=[key_arn1, key_arn2, key_arn3])

# Decrypt
```

```
plaintext, header = client.decrypt(source=ciphertext,
 key_provider=master_key_provider)
```

### Rust

```
// Instantiate the AWS Encryption SDK client with limited encrypted data keys
let esdk_config = AwsEncryptionSdkConfig::builder()
                    .max_encrypted_data_keys(max_encrypted_data_keys)
                    .build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "AES_256_012";

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

//  Generate `max_encrypted_data_keys` raw AES keyrings to use with your keyring
let mut raw_aes_keyrings: Vec<KeyringRef> = vec![];

assert!(max_encrypted_data_keys > 0, "max_encrypted_data_keys MUST be greater than
 0");

let mut i = 0;
while i < max_encrypted_data_keys {
    let aes_key_bytes = generate_aes_key_bytes();

    let raw_aes_keyring = mpl
        .create_raw_aes_keyring()
        .key_name(key_name)
        .key_namespace(key_namespace)
        .wrapping_key(aes_key_bytes)
        .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
        .send()
        .await?;

    raw_aes_keyrings.push(raw_aes_keyring);
    i += 1;
}

// Create a Multi Keyring with `max_encrypted_data_keys` AES Keyrings
```

```
let generator_keyring = raw_aes_keyrings.remove(0);

let multi_keyring = mpl
    .create_multi_keyring()
    .generator(generator_keyring)
    .child_keyrings(raw_aes_keyrings)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client with limited encrypted data keys
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{
    MaxEncryptedDataKeys: &maxEncryptedDataKeys,
})
if err != nil {
    panic(err)
}

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "RSA_2048_06"

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

//  Generate `maxEncryptedDataKeys` raw AES keyrings to use with your keyring
```

```
rawAESKeyrings := make([]mpltypes.IKeyring, 0, maxEncryptedDataKeys)
var i int64 = 0
for i < maxEncryptedDataKeys {
    key, err := generate256KeyBytesAES()
 if err != nil {
  panic(err)
 }
    aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
  KeyName:      keyName,
  KeyNamespace: keyNamespace,
  WrappingKey:  key,
  WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
 }
 aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
    if err != nil {
  panic(err)
 }
    rawAESKeyrings = append(rawAESKeyrings, aesKeyring)
    i++
}

// Create a Multi Keyring with `max_encrypted_data_keys` AES Keyrings
createMultiKeyringInput := mpltypes.CreateMultiKeyringInput{
    Generator:     rawAESKeyrings[0],
    ChildKeyrings: rawAESKeyrings[1:],
}
multiKeyring, err := matProv.CreateMultiKeyring(context.Background(),
 createMultiKeyringInput)
if err != nil {
    panic(err)
}
```

# 검색 필터 생성

KMS 키로 암호화된 데이터를 복호화할 때는 사용하는 래핑 키를 사용자가 지정한 키로만 제한하는 엄격 모드에서 복호화하는 것이 가장 좋습니다. 하지만 필요한 경우 래핑 키를 지정하지 않는 검색 모드에서 복호화할 수도 있습니다. 이 모드에서는 해당 KMS 키를 소유하거나 액세스할 AWS KMS 수 있는 사용자에 관계없이 암호화한 KMS 키를 사용하여 암호화된 데이터 키를 복호화할 수 있습니다.

검색 모드에서 복호화해야 하는 경우 항상 지정된 AWS 계정 및 파티션의 키로 사용할 수 있는 KMS 키를 제한하는 검색 필터를 사용하는 것이 좋습니다. 검색 필터는 선택 사항이지만 모범 사례입니다.

다음 표를 사용하여 검색 필터의 파티션 값을 확인하세요.

| 리전 | Partition |
| --- | --- |
| AWS 리전 | `aws` |
| 중국 리전 | `aws-cn` |
| AWS GovCloud (US) Regions | `aws-us-gov` |

이 섹션의 예제에서는 검색 필터를 만드는 방법을 보여줍니다. 코드를 사용하기 전에 예제 값을 AWS 계정 및 파티션의 유효한 값으로 바꿉니다.

C

전체 예제는 AWS Encryption SDK for C의 kms_discovery.cpp를 참조하세요.

```
/* 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).Buil
```

C# / .NET

전체 예제는 AWS Encryption SDK for .NET의 DiscoveryFilterExample.cs를 참조하세요.

```
// 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 \
                     --wrapping-keys discovery=true \
                                     discovery-account=111122223333 \
                                     discovery-partition=aws \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 1 \
                     --buffer \
                     --output .
```

Java

전체 예제는 AWS Encryption SDK for Java의 DiscoveryDecryptionExample.java를 참조하세요.

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

DiscoveryFilter discoveryFilter = new DiscoveryFilter("aws", 111122223333);
```

JavaScript (Node and Browser)

전체 예제를 보려면 AWS Encryption SDK for JavaScript의 kms_filtered_discovery.ts(Node.js) 및 kms_multi_region_discovery.ts(브라우저)를 참조하세요.

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

Python

전체 예제는 AWS Encryption SDK for Python의 discovery_kms_provider.py를 참조하세요.

```
# Create the discovery filter and specify the region
```

```
decrypt_kwargs = dict(
        discovery_filter=DiscoveryFilter(account_ids="111122223333",
 partition="aws"),
        discovery_region="us-west-2",
    )
```

Rust

```
let discovery_filter = DiscoveryFilter::builder()
    .account_ids(vec![111122223333.to_string()])
    .partition("aws".to_string())
    .build()?;
```

Go

```
import (
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
)

discoveryFilter := mpltypes.DiscoveryFilter{
    AccountIds: []string{111122223333},
    Partition:  "aws",
}
```

# 필요한 암호화 컨텍스트 CMM 구성

필요한 암호화 컨텍스트 CMM을 사용하여 암호화 작업에 암호화 컨텍스트를 요구할 수 있습니다. 암호화 컨텍스트는 비밀이 아닌 키-값 페어 세트입니다. 암호화 컨텍스트는 암호화된 데이터에 암호적으로 바인딩되므로 필드를 복호화하는 데 동일한 암호화 컨텍스트가 필요합니다. 필수 암호화 컨텍스트 CMM을 사용하는 경우 모든 암호화 및 복호화 호출에 포함되어야 하는 필수 암호화 컨텍스트 키(필수 키)를 하나 이상 지정할 수 있습니다.

> **Note**
>
> 필수 암호화 컨텍스트 CMM은 다음 버전에서만 지원됩니다.
>
> - 의 버전 3.x AWS Encryption SDK for Java
> - for .NET 버전 AWS Encryption SDK 4.x

- 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
- Go AWS Encryption SDK 용의 버전 0.1.x 이상

필요한 암호화 컨텍스트 CMM을 사용하여 데이터를 암호화하는 경우 지원되는 버전 중 하나로만 데이터를 복호화할 수 있습니다.

암호화 시는 사용자가 지정한 암호화 컨텍스트에 필요한 모든 암호화 컨텍스트 키가 포함되어 있는지 AWS Encryption SDK 확인합니다. 는 지정한 암호화 컨텍스트에 AWS Encryption SDK 서명합니다. 필수 키가 아닌 키-값 페어만 직렬화되어, 암호화 작업에서 반환되는 암호화된 메시지의 헤더에 일반 텍스트로 저장됩니다.

복호화 시, 필수 키를 나타내는 모든 키-값 페어가 포함된 암호화 컨텍스트를 제공해야 합니다. AWS Encryption SDK 는이 암호화 컨텍스트와 암호화된 메시지의 헤더에 저장된 키-값 페어를 사용하여 암호화 작업에서 지정한 원래 암호화 컨텍스트를 재구성합니다. AWS Encryption SDK 에서 원래 암호화 컨텍스트를 재구성할 수 없는 경우 복호화 작업이 실패합니다. 필수 키가 포함된 키-값 페어에 잘못된 값을 입력하면 암호화된 메시지를 복호화할 수 없습니다. 암호화 시 지정한 것과 동일한 키-값 페어를 제공해야 합니다.

**Important**

암호화 컨텍스트에서 필수 키에 어떤 값을 선택할지 신중하게 고려하세요. 복호화 시 동일한 키와 해당 값을 다시 제공할 수 있어야 합니다. 필수 키를 재생성할 수 없는 경우 암호화된 메시지를 복호화할 수 없습니다.

다음 예제에서는 필요한 암호화 컨텍스트 CMM으로 AWS KMS 키링을 초기화합니다.

C# / .NET

```
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.
    RequiredEncryptionContextKeys = new List<string>(encryptionContext.Keys)
};

// Create the required encryption context CMM
var requiredEcCMM = mpl.CreateRequiredEncryptionContextCMM(createCMMInput);
```

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()
        );
```

Python

필요한 암호화 컨텍스트 CMM과 AWS Encryption SDK for Python 함께를 사용하려면 재료 공급자 라이브러리(MPL)도 사용해야 합니다.

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Create your encryption context
encryption_context: Dict[str, str] = {
    "key1": "value1",
    "key2": "value2",
```

```
    "requiredKey1": "requiredValue1",
    "requiredKey2": "requiredValue2"
}

# Create a list of required encryption context keys
required_encryption_context_keys: List[str] = ["requiredKey1", "requiredKey2"]

# Instantiate the material providers library
mpl: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS keyring
 keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
    kms_key_id=kms_key_id,
    kms_client=boto3.client('kms', region_name="us-west-2")
)
kms_keyring: IKeyring = mpl.create_aws_kms_keyring(keyring_input)

# Create the required encryption context CMM
underlying_cmm: ICryptographicMaterialsManager = \
    mpl.create_default_cryptographic_materials_manager(
        CreateDefaultCryptographicMaterialsManagerInput(
            keyring=kms_keyring
        )
    )

required_ec_cmm: ICryptographicMaterialsManager = \
    mpl.create_required_encryption_context_cmm(
        CreateRequiredEncryptionContextCMMInput(
            required_encryption_context_keys=required_encryption_context_keys,
            underlying_cmm=underlying_cmm,
        )
    )
```

### Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create an AWS KMS client
```

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Create your encryption context
let encryption_context = HashMap::from([
    ("key1".to_string(), "value1".to_string()),
    ("key2".to_string(), "value2".to_string()),
    ("requiredKey1".to_string(), "requiredValue1".to_string()),
    ("requiredKey2".to_string(), "requiredValue2".to_string()),
]);

// Create a list of required encryption context keys
let required_encryption_context_keys: Vec<String> = vec![
    "requiredKey1".to_string(),
    "requiredKey2".to_string(),
];

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create the AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

kms_multi_keyring: IKeyring = mat_prov.create_aws_kms_multi_keyring(
    input=kms_multi_keyring_input
)

// Create the required encryption context CMM
let underlying_cmm = mpl
    .create_default_cryptographic_materials_manager()
    .keyring(kms_keyring)
    .send()
    .await?;

let required_ec_cmm = mpl
    .create_required_encryption_context_cmm()
    .underlying_cmm(underlying_cmm.clone())
```

```
    .required_encryption_context_keys(required_encryption_context_keys)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = defaultKmsKeyRegion
})

// Create an encryption context
encryptionContext := map[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",
```

```
}

// Create a list of required encryption context keys
requiredEncryptionContextKeys := []string{}
requiredEncryptionContextKeys = append(requiredEncryptionContextKeys,
 "requiredKey1", "requiredKey2")

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create the AWS KMS keyring
awsKmsKeyringInput := mpltypes.CreateAwsKmsKeyringInput{
    KmsClient: kmsClient,
    KmsKeyId:  utils.GetDefaultKMSKeyId(),
}
awsKmsKeyring, err := matProv.CreateAwsKmsKeyring(context.Background(),
 awsKmsKeyringInput)
if err != nil {
    panic(err)
}

// Create the required encryption context CMM
underlyingCMM, err :=
 matProv.CreateDefaultCryptographicMaterialsManager(context.Background(),
 mpltypes.CreateDefaultCryptographicMaterialsManagerInput{Keyring: awsKmsKeyring})
if err != nil {
    panic(err)
}
requiredEncryptionContextInput := mpltypes.CreateRequiredEncryptionContextCMMInput{
    UnderlyingCMM: underlyingCMM,
    RequiredEncryptionContextKeys: requiredEncryptionContextKeys,
}
requiredEC, err := matProv.CreateRequiredEncryptionContextCMM(context.Background(),
 requiredEncryptionContextInput)
if err != nil {
    panic(err)
}
```

# 커밋 정책 설정

커밋 정책은 애플리케이션이 키 커밋을 사용하여 암호화 및 복호화할지 여부를 결정하는 구성 설정입니다. 키 커밋으로 데이터를 암호화하고 복호화하는 것이 AWS Encryption SDK 모범 사례입니다.

커밋 정책을 설정하고 조정하는 것은 AWS Encryption SDK 버전 1.7.x 이하에서 버전 2.0.x 이상으로 마이그레이션하기 위한 중요한 단계입니다. 이 진행 과정은 마이그레이션 주제에 자세히 설명되어 있습니다.

AWS Encryption SDK 의 최신 버전(버전 2.0.x부터)의 기본 커밋 정책 값인 `RequireEncryptRequireDecrypt`는 대부분의 상황에 적합합니다. 하지만 키 커밋 없이 암호화된 사이퍼텍스트를 복호화해야 하는 경우에는 커밋 정책을 `RequireEncryptAllowDecrypt`로 변경해야 할 수도 있습니다. 각 프로그래밍 언어에서 커밋 정책을 설정하는 방법에 대한 예는 커밋 정책 설정 섹션을 참조하세요.

# 스트리밍 데이터로 작업

복호화를 위해 데이터를 스트리밍할 때는 무결성 검사가 완료된 후 디지털 서명이 확인되기 전에가 복호화된 일반 텍스트를 AWS Encryption SDK 반환한다는 점에 유의하세요. 서명이 확인될 때까지 일반 텍스트를 반환하거나 사용하지 않도록 하려면 전체 복호화 프로세스가 완료될 때까지 스트리밍된 일반 텍스트를 버퍼링하는 것이 좋습니다.

이 문제는 복호화를 위해 사이퍼텍스트를 스트리밍하는 경우와, 디지털 서명이 포함된 알고리즘 제품군(예: 기본 알고리즘 제품군)을 사용하는 경우에만 발생합니다.

버퍼링을 더 쉽게 하기 위해 Node.js AWS Encryption SDK for JavaScript 와 같은 일부 AWS Encryption SDK 언어 구현에는 복호화 방법의 일부로 버퍼링 기능이 포함됩니다. 항상 입력과 출력을 스트리밍하는 AWS Encryption CLI는 버전 1.9.x 및 2.2.x에 `--buffer` 파라미터를 도입했습니다. 다른 언어 구현에서는 기존의 버퍼링 기능을 사용할 수 있습니다. (.NET AWS Encryption SDK 용는 스트리밍을 지원하지 않습니다.)

디지털 서명이 없는 알고리즘 제품군을 사용하는 경우 각 언어 구현에서 `decrypt-unsigned` 기능을 사용해야 합니다. 이 기능은 사이퍼텍스트를 복호화하지만 서명된 사이퍼텍스트를 발견하면 실패합니다. 자세한 내용은 알고리즘 제품군 선택을 참조하세요.

# 데이터 키 캐싱

일반적으로 데이터 키 재사용은 권장되지 않지만는 데이터 키의 제한된 재사용을 제공하는 데이터 키 캐싱 옵션을 AWS Encryption SDK 제공합니다. 데이터 키 캐싱은 일부 애플리케이션의 성능을 향상시

키고 키 인프라에 대한 호출을 줄일 수 있습니다. 프로덕션 환경에서 데이터 키 캐싱을 사용하기 전에 보안 임계값을 조정하고, 데이터 키 재사용의 이점이 단점보다 큰지 테스트하세요.

# 의 키 스토어 AWS Encryption SDK

에서 AWS Encryption SDK키 스토어는 계층적 AWS KMS 키링에서 사용하는 계층적 데이터를 유지하는 Amazon DynamoDB 테이블입니다. 키 스토어는 계층적 키링을 사용하여 암호화 작업을 수행하기 AWS KMS 위해에서 수행해야 하는 호출 수를 줄이는 데 도움이 됩니다.

키 스토어는 계층적 키링이 봉투 암호화를 수행하고 데이터 암호화 키를 보호하는 데 사용하는 브랜치 키를 유지하고 관리합니다. 키 스토어는 활성 브랜치 키와 브랜치 키의 모든 이전 버전을 저장합니다. 활성 브랜치 키는 최신 버전의 브랜치 키입니다. 계층적 키링은 각 암호화 요청에 고유한 데이터 암호화 키를 사용하고 활성 브랜치 키에서 파생된 고유한 래핑 키로 각 데이터 암호화 키를 암호화합니다. 계층적 키링은 활성 브랜치 키와 파생된 래핑 키 사이에 설정된 계층 구조에 따라 달라집니다.

## 키 스토어 용어 및 개념

키 스토어

브랜치 키 및 비컨 키와 같은 계층적 데이터를 유지하는 DynamoDB 테이블입니다.

루트 키

키 스토어에서 브랜치 키와 비컨 키를 생성하고 보호하는 대칭 암호화 KMS 키입니다.

브랜치 키

봉투 암호화를 위한 고유한 래핑 키를 도출하기 위해 재사용되는 데이터 키입니다. 하나의 키 스토어에 여러 브랜치 키를 생성할 수 있지만 각 브랜치 키는 한 번에 하나의 활성 브랜치 키 버전만 가질 수 있습니다. 활성 브랜치 키는 최신 버전의 브랜치 키입니다.

브랜치 키는 kms:GenerateDataKeyWithoutPlaintext 작업을 AWS KMS keys 사용하여 파생됩니다.

래핑 키

암호화 작업에 사용되는 데이터 암호화 키를 암호화하는 데 사용되는 고유한 데이터 키입니다.

래핑 키는 브랜치 키에서 파생됩니다. 키 파생 프로세스에 대한 자세한 내용은 AWS KMS 계층적 키링 기술 세부 정보를 참조하세요.

데이터 암호화 키

암호화 작업에 사용되는 데이터 키입니다. 계층적 키링은 각 암호화 요청에 고유한 데이터 암호화 키를 사용합니다.

# 최소 권한 구현

키 스토어 및 AWS KMS 계층적 키링을 사용하는 경우 다음 역할을 정의하여 최소 권한 원칙을 따르는 것이 좋습니다.

키 스토어 관리자

키 스토어 관리자는 키 스토어와 키 스토어가 유지 및 보호하는 브랜치 키를 생성하고 관리할 책임이 있습니다. 키 스토어 관리자는 키 스토어 역할을 하는 Amazon DynamoDB 테이블에 대한 쓰기 권한이 있는 유일한 사용자여야 합니다. 및 `CreateKey`와 같은 권한 있는 관리자 작업에 액세스할 수 있는 유일한 사용자여야 합니다`VersionKey`. 키 스토어 작업을 정적으로 구성하는 경우에만 이러한 작업을 수행할 수 있습니다.

`CreateKey`는 키 스토어 허용 목록에 새 KMS 키 ARN을 추가할 수 있는 권한 있는 작업입니다. 이 KMS 키는 새 활성 브랜치 키를 생성할 수 있습니다. KMS 키가 브랜치 키 스토어에 추가되면 삭제할 수 없으므로이 작업에 대한 액세스를 제한하는 것이 좋습니다.

키 스토어 사용자

대부분의 사용 사례에서 키 스토어 사용자는 데이터를 암호화, 복호화, 서명 및 확인할 때 계층적 키링을 통해서만 키 스토어와 상호 작용합니다. 따라서 키 스토어 역할을 하는 Amazon DynamoDB 테이블에 대한 읽기 권한만 있으면 됩니다. 키 스토어 사용자는 , `GetActiveBranchKey` `GetBranchKeyVersion`및와 같이 암호화 작업을 가능하게 하는 사용 작업에 대한 액세스 권한만 있으면 됩니다`GetBeaconKey`. 사용하는 브랜치 키를 생성하거나 관리하는 데는 권한이 필요하지 않습니다.

키 스토어 작업이 정적으로 구성되거나 검색용으로 구성된 경우 사용 작업을 수행할 수 있습니다. 키 스토어 작업이 검색하도록 구성된 경우 관리자 작업(`CreateKey` 및 `VersionKey`)을 수행할 수 없습니다.

브랜치 키 스토어 관리자가 브랜치 키 스토어에 여러 KMS 키를 허용 목록에 추가한 경우 계층적 키링이 여러 KMS 키를 사용할 수 있도록 키 스토어 사용자가 검색을 위해 키 스토어 작업을 구성하는 것이 좋습니다.

# 키 스토어 생성

브랜치 키를 생성하거나 AWS KMS 계층적 키링을 사용하려면 먼저 브랜치 키를 관리하고 보호하는 Amazon DynamoDB 테이블인 키 스토어를 생성해야 합니다.

> **Important**
>
> 브랜치 키를 유지하는 DynamoDB 테이블을 삭제하지 마십시오. 이 테이블을 삭제하면 계층적 키링을 사용하여 암호화된 데이터를 해독할 수 없습니다.

파티션 키 및 정렬 키에 대해 다음과 같은 필수 문자열 값을 사용하여 Amazon DynamoDB 개발자 안내서의 테이블 생성 절차를 따릅니다.

| | 파티션 키 | 정렬 키 |
|---|---|---|
| 기본 테이블 | `branch-key-id` | `type` |

논리적 키 스토어 이름

키 스토어 역할을 하는 DynamoDB 테이블의 이름을 지정할 때는 키 스토어 작업을 구성할 때 지정할 논리적 키 스토어 이름을 신중하게 고려하는 것이 중요합니다. ??? 논리적 키 스토어 이름은 키 스토어의 식별자 역할을 하며 첫 번째 사용자가 처음 정의한 후에는 변경할 수 없습니다. 키 스토어 작업에 항상 동일한 논리적 키 스토어 이름을 지정해야 합니다.

DynamoDB 테이블 이름과 논리적 키 스토어 이름 사이에 one-to-one 매핑이 있어야 합니다. 논리적 키 스토어 이름은 테이블에 저장된 모든 데이터에 암호로 바인딩되어 DynamoDB 복원 작업을 간소화합니다. 논리적 키 스토어 이름은 DynamoDB 테이블 이름과 다를 수 있지만 DynamoDB 테이블 이름을 논리적 키 스토어 이름으로 지정하는 것이 좋습니다. 백업에서 DynamoDB 테이블을 복원한 후 테이블 이름이 변경되는 경우 논리적 키 스토어 이름을 새 DynamoDB 테이블 이름에 매핑하여 계층적 키링이 여전히 키 스토어에 액세스할 수 있도록 할 수 있습니다.

논리적 키 스토어 이름에 기밀 또는 민감한 정보를 포함하지 마세요. 논리적 키 스토어 이름은 AWS KMS CloudTrail 이벤트의 일반 텍스트로 로 표시됩니다`tablename`.

다음 단계

1. the section called “키 스토어 작업 구성”
2. the section called “브랜치 키 생성”
3. AWS KMS 계층적 키링 생성

# 키 스토어 작업 구성

키 스토어 작업은 사용자가 수행할 수 있는 작업과 AWS KMS 계층적 키링이 키 스토어에 나열된 KMS 키를 사용하는 방법을 결정합니다. 는 다음과 같은 키 스토어 작업 구성을 AWS Encryption SDK 지원합니다.

정적

키 스토어를 정적으로 구성하면 키 스토어는 키 스토어 작업을 구성할 `kmsConfiguration` 때에서 제공하는 KMS 키 ARN과 연결된 KMS 키만 사용할 수 있습니다. 브랜치 키를 생성, 버전 관리 또는 가져올 때 다른 KMS 키 ARN이 발생하는 경우 예외가 발생합니다.

에서 다중 리전 KMS 키를 지정할 수 `kmsConfiguration`있지만 리전을 포함한 키의 전체 ARN은 KMS 키에서 파생된 브랜치 키에 유지됩니다. 다른 리전에서 키를 지정할 수 없으며, 값이 일치하려면 정확히 동일한 다중 리전 키를 제공해야 합니다.

키 스토어 작업을 정적으로 구성하면 사용 작업(`GetActiveBranchKey`, , `GetBranchKeyVersionGetBeaconKey`) 및 관리 작업(`CreateKey` 및 )을 수행할 수 있습니다`VersionKey`. `CreateKey`는 키 스토어 허용 목록에 새 KMS 키 ARN을 추가할 수 있는 권한 있는 작업입니다. 이 KMS 키는 새 활성 브랜치 키를 생성할 수 있습니다. KMS 키가 키 스토어에 추가되면 삭제할 수 없으므로이 작업에 대한 액세스를 제한하는 것이 좋습니다.

Discovery

검색을 위해 키 스토어 작업을 구성할 때 키 스토어는 키 스토어에 허용 목록에 있는 모든 AWS KMS key ARN을 사용할 수 있습니다. 그러나 다중 리전 KMS 키가 발견되고 키의 ARN에 있는 리전이 사용 중인 AWS KMS 클라이언트의 리전과 일치하지 않는 경우 예외가 발생합니다.

검색을 위해 키 스토어를 구성할 때는 `CreateKey` 및와 같은 관리 작업을 수행할 수 없습니다`VersionKey`. 암호화, 복호화, 서명 및 확인 작업을 활성화하는 사용 작업만 수행할 수 있습니다. 자세한 내용은 the section called “최소 권한 구현” 단원을 참조하십시오.

## 키 스토어 작업 구성

키 스토어 작업을 구성하기 전에 다음 사전 조건을 충족하는지 확인합니다.

- 수행해야 할 작업을 결정합니다. 자세한 내용은 the section called “최소 권한 구현” 단원을 참조하십시오.
- 논리적 키 스토어 이름 선택

DynamoDB 테이블 이름과 논리적 키 스토어 이름 사이에 one-to-one 매핑이 있어야 합니다. 논리적 키 스토어 이름은 DynamoDB 복원 작업을 간소화하기 위해 테이블에 저장된 모든 데이터에 암호화 방식으로 바인딩되며, 첫 번째 사용자가 처음 정의한 후에는 변경할 수 없습니다. 키 스토어 작업에서 항상 동일한 논리적 키 스토어 이름을 지정해야 합니다. 자세한 내용은 logical key store name 단원을 참조하십시오.

### 정적 구성

다음 예제에서는 키 스토어 작업을 정적으로 구성합니다. 키 스토어 역할을 하는 DynamoDB 테이블의 이름, 키 스토어의 논리적 이름, 대칭 암호화 KMS 키를 식별하는 KMS 키 ARN을 지정해야 합니다.

> **Note**
>
> 키 스토어 서비스를 정적으로 구성할 때 지정하는 KMS 키 ARN을 신중하게 고려하세요. `CreateKey` 작업은 KMS 키 ARN을 브랜치 키 스토어 허용 목록에 추가합니다. KMS 키가 브랜치 키 스토어에 추가되면 삭제할 수 없습니다.

#### Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                  KeyStoreConfig.builder()
                          .ddbClient(DynamoDbClient.create())
                          .ddbTableName(keyStoreName)
                          .logicalKeyStoreName(logicalKeyStoreName)
                          .kmsClient(KmsClient.create())
                          .kmsConfiguration(KMSConfiguration.builder()
                                  .kmsKeyArn(kmsKeyArn)
                                  .build())
                          .build()).build();
```

#### C# / .NET

```
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);
```

### Python

```
keystore: KeyStore = KeyStore(
    config=KeyStoreConfig(
        ddb_client=ddb_client,
        ddb_table_name=key_store_name,
        logical_key_store_name=logical_key_store_name,
        kms_client=kms_client,
        kms_configuration=KMSConfigurationKmsKeyArn(
            value=kms_key_id
        ),
    )
)
```

### Rust

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let key_store_config = KeyStoreConfig::builder()
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .ddb_client(aws_sdk_dynamodb::Client::new(&sdk_config))
    .ddb_table_name(key_store_name)
    .logical_key_store_name(logical_key_store_name)
    .kms_configuration(KmsConfiguration::KmsKeyArn(kms_key_arn.to_string()))
    .build()?;

let keystore = keystore_client::Client::from_conf(key_store_config)?;
```

### Go

```
import (
 keystore "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygenerated"
 keystoretypes "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygeneratedtypes"
)
```

```
kmsConfig := keystoretypes.KMSConfigurationMemberkmsKeyArn{
    Value: kmsKeyArn,
}
keyStore, err := keystore.NewClient(keystoretypes.KeyStoreConfig{
    DdbTableName:        keyStoreTableName,
    KmsConfiguration:    &kmsConfig,
    LogicalKeyStoreName: logicalKeyStoreName,
    DdbClient:           ddbClient,
    KmsClient:           kmsClient,
})
if err != nil {
    panic(err)
}
```

### 검색 구성

다음 예제에서는 검색을 위한 키 스토어 작업을 구성합니다. 키 스토어 역할을 하는 DynamoDB 테이블의 이름과 논리적 키 스토어 이름을 지정해야 합니다.

#### Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                KeyStoreConfig.builder()
                        .ddbClient(DynamoDbClient.create())
                        .ddbTableName(keyStoreName)
                        .logicalKeyStoreName(logicalKeyStoreName)
                        .kmsClient(KmsClient.create())
                        .kmsConfiguration(KMSConfiguration.builder()
                                .discovery(Discovery.builder().build())
                                .build())
                        .build()).build();
```

#### C# / .NET

```
var keystoreConfig = new KeyStoreConfig
 {
     KmsClient = new AmazonKeyManagementServiceClient(),
     KmsConfiguration = new KMSConfiguration {Discovery = new Discovery()},
     DdbTableName = keyStoreName,
     DdbClient = new AmazonDynamoDBClient(),
     LogicalKeyStoreName = logicalKeyStoreName
 };
```

```
 var keystore = new KeyStore(keystoreConfig);
```

### Python

```
keystore: KeyStore = KeyStore(
    config=KeyStoreConfig(
        ddb_client=ddb_client,
        ddb_table_name=key_store_name,
        logical_key_store_name=logical_key_store_name,
        kms_client=kms_client,
        kms_configuration=KMSConfigurationDiscovery(
            value=Discovery()
        ),
    )
)
```

### Rust

```
let key_store_config = KeyStoreConfig::builder()
        .kms_client(kms_client)
        .ddb_client(ddb_client)
        .ddb_table_name(key_store_name)
        .logical_key_store_name(logical_key_store_name)

 .kms_configuration(KmsConfiguration::Discovery(Discovery::builder().build()?))
        .build()?;
```

### Go

```
import (
 keystore "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygenerated"
 keystoretypes "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygeneratedtypes"
)

kmsConfig := keystoretypes.KMSConfigurationMemberdiscovery{}
keyStore, err := keystore.NewClient(keystoretypes.KeyStoreConfig{
    DdbTableName:        keyStoreName,
    KmsConfiguration:    &kmsConfig,
    LogicalKeyStoreName: logicalKeyStoreName,
    DdbClient:           ddbClient,
```

```
    KmsClient:           kmsClient,
})
if err != nil {
    panic(err)
}
```

# 활성 브랜치 키 생성

브랜치 키는 AWS KMS 계층적 키링 AWS KMS key 이 호출 수를 줄이는 데 사용하는에서 파생된 데이터 키입니다 AWS KMS. 활성 브랜치 키는 최신 버전의 브랜치 키입니다. 계층적 키링은 모든 암호화 요청에 대해 고유한 데이터 키를 생성하고 활성 브랜치 키에서 파생된 고유한 래핑 키로 각 데이터 키를 암호화합니다.

새 활성 브랜치 키를 생성하려면 키 스토어 작업을 정적으로 구성해야 합니다. `CreateKey`는 키 스토어 작업 구성에 지정된 KMS 키 ARN을 키 스토어 허용 목록에 추가하는 권한 있는 작업입니다. 그런 다음 KMS 키를 사용하여 새 활성 브랜치 키를 생성합니다. KMS 키가 키 스토어에 추가되면 삭제할 수 없으므로이 작업에 대한 액세스를 제한하는 것이 좋습니다.

키 스토어에서 KMS 키 하나를 허용 목록에 추가하거나 키 스토어 작업 구성에서 지정한 KMS 키 ARN을 업데이트하고를 `CreateKey` 다시 호출하여 여러 KMS 키를 허용 목록에 추가할 수 있습니다. 여러 KMS 키를 허용 목록으로 표시하는 경우 키 스토어 사용자는 액세스 권한이 있는 키 스토어에서 허용 목록으로 지정된 키를 사용할 수 있도록 검색을 위해 키 스토어 작업을 구성해야 합니다. 자세한 내용은 the section called “키 스토어 작업 구성” 단원을 참조하십시오.

필요한 권한

브랜치 키를 생성하려면 키 스토어 작업에 지정된 KMS 키에 대한 kms:GenerateDataKeyWithoutPlaintext 및 kms:ReEncrypt 권한이 필요합니다.

브랜치 키 생성

다음 작업은 키 스토어 작업 구성에서 지정한 KMS 키를 사용하여 새 활성 브랜치 키를 생성하고 키 스토어 역할을 하는 DynamoDB 테이블에 활성 브랜치 키를 추가합니다.

`CreateKey`를 호출할 때 다음과 같은 선택적 값을 지정하도록 선택할 수 있습니다.

- `branchKeyIdentifier`: 사용자 지정 `branch-key-id`를 정의합니다.

  사용자 지정 `branch-key-id`를 만들려면 `encryptionContext` 파라미터에 추가 암호화 컨텍스트도 포함해야 합니다.

- `encryptionContext`: kms:GenerateDataKeyWithoutPlaintext 호출에 포함된 암호화 컨텍스트에서 추가 인증 데이터(AAD)를 제공하는 선택적 비보안 키-값 페어 세트를 정의합니다.

  이 추가 암호화 컨텍스트는 `aws-crypto-ec:` 접두사와 표시됩니다.

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();
```

C# / .NET

```
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
  });
```

Python

```
additional_encryption_context = {"Additional Encryption Context for": "custom branch
 key id"}

branch_key_id: str = keystore.create_key(
    CreateKeyInput(
        branch_key_identifier = "custom-branch-key-id", # OPTIONAL
        encryption_context = additional_encryption_context, # OPTIONAL
    )
)
```

Rust

```
let additional_encryption_context = HashMap::from([
    ("Additional Encryption Context for".to_string(), "custom branch key
 id".to_string())
]);

let branch_key_id = keystore.create_key()
    .branch_key_identifier("custom-branch-key-id") // OPTIONAL
    .encryption_context(additional_encryption_context) // OPTIONAL
    .send()
    .await?
    .branch_key_identifier
    .unwrap();
```

Go

```
encryptionContext := map[string]string{
    "Additional Encryption Context for": "custom branch key id",
}

branchKey, err := keyStore.CreateKey(context.Background(),
 keystoretypes.CreateKeyInput{
    BranchKeyIdentifier: &customBranchKeyId,
    EncryptionContext:   additional_encryption_context,
})
if err != nil {
    return "", err
}
```

먼저, `CreateKey` 작업은 다음 값을 생성합니다.

- `branch-key-id`의 버전 4 Universally Unique Identifier(UUID)(사용자 지정 `branch-key-id`를 지정하지 않은 경우).
- 브랜치 키 버전의 버전 4 UUID
- 협정 세계시(UTC)의 ISO 8601 날짜 및 시간 형식의 `timestamp`.

그런 다음 `CreateKey` 작업은 다음 요청을 사용하여 kms:GenerateDataKeyWithoutPlaintext를 호출합니다.

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

다음으로 CreateKey 작업은 kms:ReEncrypt를 호출하여 암호화 컨텍스트를 업데이트하고 브랜치 키에 대한 활성 레코드를 생성합니다.

마지막으로 CreateKey 작업은 ddb:TransactWriteItems를 호출하여 2단계에서 생성한 테이블의 브랜치 키를 유지할 새 항목을 작성합니다. 항목에는 다음 속성이 있습니다.

```
{
      "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"
 }
```

## 활성 브랜치 키 교체

각 브랜치 키에는 한 번에 하나의 활성 버전만 있을 수 있습니다. 일반적으로 각 활성 브랜치 키 버전은 여러 요청을 충족하는 데 사용됩니다. 하지만 활성 브랜치 키의 재사용 범위를 제어하고 활성 브랜치 키의 교체 빈도를 결정할 수 있습니다.

브랜치 키는 일반 텍스트 데이터 키를 암호화하는 데 사용되지 않습니다. 이들은 일반 텍스트 데이터 키를 암호화하는 고유한 래핑 키를 도출하는 데 사용됩니다. 래핑 키 추출 프로세스는 28바이트의 무작위성을 갖는 고유한 32바이트 래핑 키를 생성합니다. 즉, 브랜치 키는 암호화 마모가 발생하기 전에 79

옥틸리온($2^{96}$) 이상의 고유한 래핑 키를 도출할 수 있습니다. 이렇게 소진 위험은 매우 낮지만 비즈니스 또는 계약 규칙이나 정부 규정으로 인해 활성 브랜치 키를 교체해야 할 수도 있습니다.

브랜치 키의 활성 버전은 교체할 때까지 활성 상태로 유지됩니다. 이전 버전의 활성 브랜치 키는 암호화 작업을 수행하는 데 사용되지 않으며 새 래핑 키를 추출하는 데 사용될 수 없지만, 여전히 쿼리할 수 있으며 활성 상태에서 암호화한 데이터 키를 해독하는 래핑 키를 제공할 수 있습니다.

필수 권한

브랜치 키를 교체하려면 키 스토어 작업에 지정된 KMS 키에 대한 kms:GenerateDataKeyWithoutPlaintext 및 kms:ReEncrypt 권한이 필요합니다.

활성 브랜치 키 교체

`VersionKey` 작업을 사용하여 활성 브랜치 키를 교체합니다. 활성 브랜치 키를 교체하면 이하 버전을 대체하는 새 브랜치 키가 생성됩니다. 활성 브랜치 키를 교체해도 `branch-key-id`는 변경되지 않습니다. `VersionKey`를 호출할 때 현재 활성 브랜치 키를 식별하는 `branch-key-id`를 지정해야 합니다.

Java

```
keystore.VersionKey(
    VersionKeyInput.builder()
        .branchKeyIdentifier("branch-key-id")
        .build()
);
```

C# / .NET

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

Python

```
keystore.version_key(
    VersionKeyInput(
        branch_key_identifier=branch_key_id
    )
)
```

Rust

```
keystore.version_key()
```

```
        .branch_key_identifier(branch_key_id)
        .send()
        .await?;
```

Go

```
_, err = keyStore.VersionKey(context.Background(), keystoretypes.VersionKeyInput{
    BranchKeyIdentifier: branchKeyId,
})
if err != nil {
    return err
}
```

# 키링

지원되는 프로그래밍 언어 구현은 키링을 사용하여 봉투 암호화를 수행합니다. 키링은 데이터 키를 생성, 암호화 및 복호화합니다. 키링에 따라 각 메시지를 보호하는 고유한 데이터 키의 원본과 해당 데이터 키를 암호화하는 래핑 키가 결정됩니다. 암호화할 때 키링을 지정하고 암호를 복호화할 때는 동일하거나 다른 키링을 지정합니다. SDK에서 제공하는 키링을 사용하거나 호환되는 사용자 지정 키링을 직접 작성할 수 있습니다.

각 키링을 개별적으로 사용하거나 키링을 여러 개의 키링으로 결합할 수 있습니다. 대부분의 키링이 데이터 키를 생성, 암호화 및 복호화할 수 있지만, 데이터 키만 생성하는 키링과 같이 특정 작업 하나만 수행하는 키링을 만들고 해당 키링을 다른 키링과 조합하여 사용할 수 있습니다.

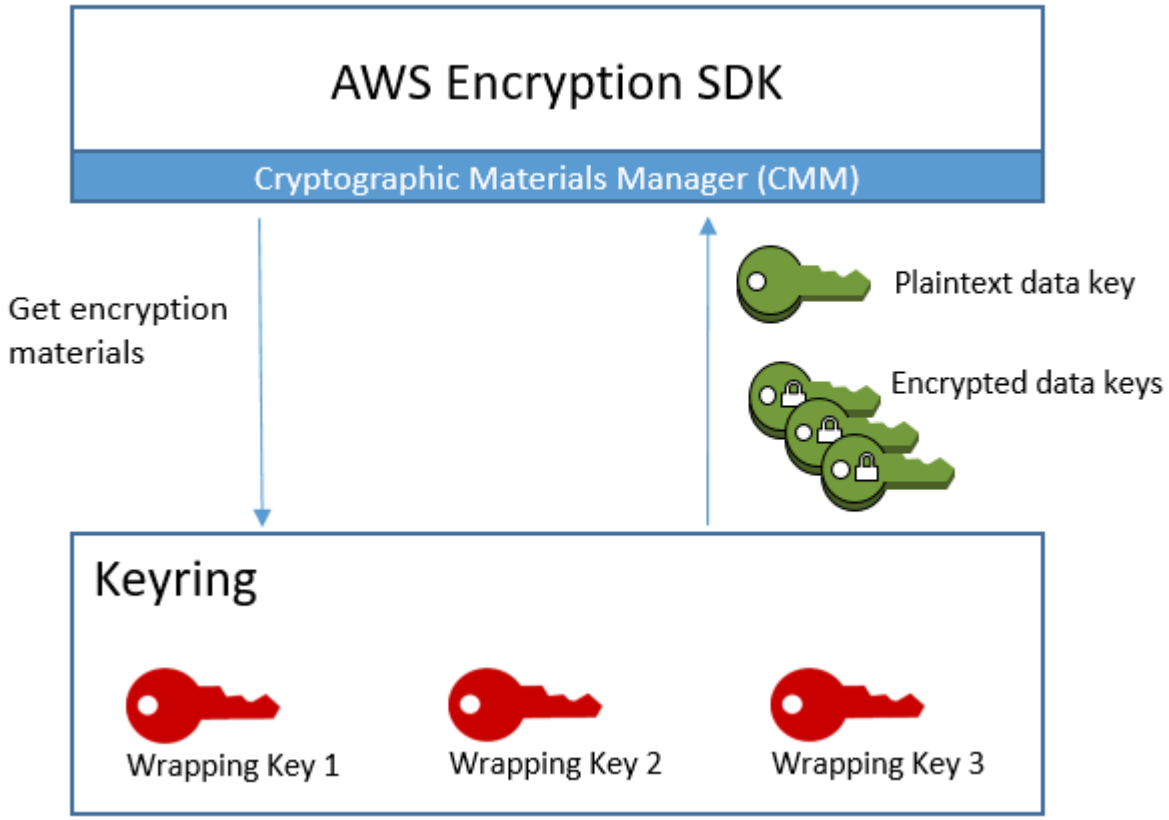
래핑 키를 보호하고 AWS Key Management Service (AWS KMS)를 암호화되지 않은 상태로 두지 않는를 사용하는 키링과 같은 보안 경계 내에서 암호화 작업을 수행하는 AWS KMS 키링 AWS KMS keys을 사용하는 것이 좋습니다. 하드웨어 보안 모듈(HSM)에 저장되거나 다른 마스터 키 서비스에서 보호하는 래핑 키를 사용하는 키링을 작성할 수도 있습니다. 자세한 내용은 AWS Encryption SDK 사양의 키링 인터페이스 항목을 참조하세요.

키링은 다른 프로그래밍 언어 구현에 사용되는 마스터 키 및 마스터 키 공급자의 역할을 합니다. AWS Encryption SDK 의 다른 언어 구현을 사용하여 데이터를 암호화하고 복호화하는 경우 호환되는 키링과 마스터 키 제공자를 사용해야 합니다. 자세한 내용은 키링 호환성을 참조하세요.

이 주제에서는의 키링 기능을 사용하는 방법과 키링을 선택하는 AWS Encryption SDK 방법을 설명합니다.

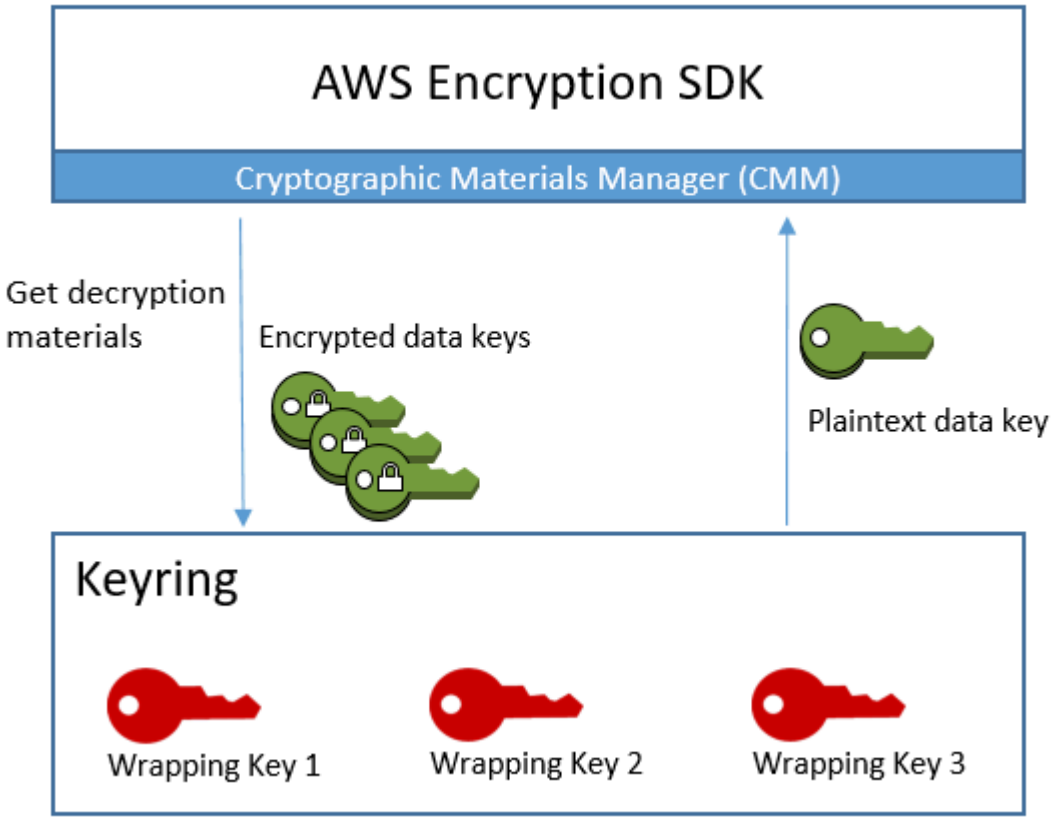
## 키링 작동 방식

데이터를 암호화할 때는 키링에 암호화 자료를 AWS Encryption SDK 요청합니다. 키링은 일반 텍스트 데이터 키와 키링의 각 래핑 키로 암호화된 데이터 키의 복사본을 반환합니다. 는 일반 텍스트 키를 AWS Encryption SDK 사용하여 데이터를 암호화한 다음 일반 텍스트 데이터 키를 삭제합니다. 그런 다음는 암호화된 데이터 키와 암호화된 데이터가 포함된 암호화된 메시지를 AWS Encryption SDK 반환합니다.




데이터를 복호화할 때 데이터를 암호화하는 데 사용한 것과 동일한 키링을 사용하거나 다른 키링을 사용할 수 있습니다. 데이터를 복호화하려면 복호화 키링에 암호화 키링에 래핑 키가 하나 이상 포함되거나 액세스 권한이 있어야 합니다.

는 암호화된 메시지의 암호화된 데이터 키를 키링으로 AWS Encryption SDK 전달하고 키링에 키링 중 하나를 복호화하도록 요청합니다. 키링은 해당 래핑 키를 사용하여 암호화된 데이터 키 중 하나를 암호화 해제하고 일반 텍스트 데이터 키를 반환합니다. AWS Encryption SDK 는 일반 텍스트 데이터 키를 사용하여 데이터를 복호화합니다. 키링에 있는 래핑 키 중 어느 것도 암호화된 데이터 키를 복호화할 수 없는 경우 복호화 작업이 실패합니다.




하나의 키링을 사용하거나, 동일한 유형 또는 여러 유형의 키링을 하나의 다중 키링에 조합할 수도 있습니다. 데이터를 암호화할 때 다중 키링은 다중 키링을 구성하는 모든 키링의 모든 래핑 키로 암호화된 데이터 키의 사본을 반환합니다. 다중 키링의 래핑 키 중 하나를 포함하는 키링을 사용하여 데이터를 복호화할 수 있습니다.

## 키링 호환성

의 다양한 언어 구현 AWS Encryption SDK 에는 몇 가지 아키텍처 차이가 있지만 언어 제약에 따라 완전히 호환됩니다. 한 언어 구현을 사용하여 데이터를 암호화하고 다른 언어 구현으로 복호화할 수 있습니다. 하지만 데이터 키를 암호화하고 복호화하려면 동일하거나 상응하는 래핑 키를 사용해야 합니다. 언어 제약 조건에 대한 자세한 내용은 주제에서와 같이 각 언어 구현the section called “호환성”에 대한 AWS Encryption SDK for JavaScript 주제를 참조하세요.

키링은 다음 프로그래밍 언어로 지원됩니다.

- AWS Encryption SDK for C
- AWS Encryption SDK for JavaScript

- AWS Encryption SDK .NET용
- 의 버전 3.x AWS Encryption SDK for Java
- 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
- AWS Encryption SDK Rust용
- AWS Encryption SDK Go용

## 암호화 키링에 대한 다양한 요구 사항

이외의 AWS Encryption SDK 언어 구현에서는 암호화 키링(또는 다중 키링) 또는 마스터 키 공급자의 AWS Encryption SDK for C모든 래핑 키가 데이터 키를 암호화할 수 있어야 합니다. 래핑 키가 암호화되지 않으면 암호화 메서드가 실패합니다. 따라서 호출자는 키링의 모든 키에 필요한 권한을 가지고 있어야 합니다. 검색 키링을 사용하여 단독 또는 다중 키링으로 데이터를 암호화하는 경우 암호화 작업이 실패합니다.

암호화 AWS Encryption SDK for C작업이 표준 검색 키링을 무시하지만 다중 리전 검색 키링을 단독으로 지정하거나 다중 키링에서 지정하는 경우 실패하는 경우는 예외입니다.

## 호환되는 키링 및 마스터 키 제공자

다음 표에는에서 AWS Encryption SDK 제공하는 키링과 호환되는 마스터 키 및 마스터 키 공급자가 나와 있습니다. 언어 제약 조건으로 인한 사소한 비호환성은 언어 구현에 대한 주제에 설명되어 있습니다.

| 키링: | 마스터 키 공급자: |
| --- | --- |
| AWS KMS 키링 | KMSMasterKey(Java) |
| | KMSMasterKeyProvider(Java) |
| | KMSMasterKey(Python) |
| | KMSMasterKeyProvider(Python) |

| 키링: | 마스터 키 공급자: |
| --- | --- |
| | **Note**<br>AWS Encryption SDK for Python 및 에는 AWS KMS 리전 검색 키링과 동일한 마스터 키 또는 마스터 키 공급자가 포함되지 AWS Encryption SDK for Java 않습니다. |
| AWS KMS 계층적 키링 | 다음 프로그래밍 언어 및 버전에서 지원됩니다.<br>• 의 버전 3.x AWS Encryption SDK for Java<br>• for .NET 버전 AWS Encryption SDK 4.x<br>• 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.<br>• AWS Encryption SDK for Rust 버전 1.x<br>• Go AWS Encryption SDK 용의 버전 0.1.x 이상 |
| AWS KMS ECDH 키링 | 다음 프로그래밍 언어 및 버전에서 지원됩니다.<br>• 의 버전 3.x AWS Encryption SDK for Java<br>• for .NET 버전 AWS Encryption SDK 4.x<br>• 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용되는 AWS Encryption SDK for Python경우 버전 4.x.<br>• AWS Encryption SDK for Rust 버전 1.x<br>• Go AWS Encryption SDK 용의 버전 0.1.x 이상 |
| Raw AES 키링 | 비대칭 암호화 키와 함께 사용하는 경우:<br>JceMasterKey(Java)<br><br>RawMasterKey(Python) |

| 키링: | 마스터 키 공급자: |
| --- | --- |
| Raw RSA 키링 | 비대칭 암호화 키와 함께 사용하는 경우: JceMasterKey(Java)<br><br>RawMasterKey(Python)<br><br>**Note**<br>Raw RSA 키링은 비대칭 KMS 키를 지원하지 않습니다. 비대칭 RSA KMS 키를 사용하려는 경우 for .NET 버전 AWS Encryption SDK 4.x는 대칭 암호화(`SYMMETRIC_DEFAULT`) 또는 비대칭 RSA를 사용하는 AWS KMS 키링을 지원합니다 AWS KMS keys. |
| 원시 ECDH 키링 | 다음 프로그래밍 언어 및 버전에서 지원됩니다.<br><br>• 의 버전 3.x AWS Encryption SDK for Java<br>• for .NET 버전 AWS Encryption SDK 4.x<br>• 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.<br>• AWS Encryption SDK for Rust 버전 1.x<br>• Go AWS Encryption SDK 용의 버전 0.1.x 이상 |

# AWS KMS 키링

AWS KMS 키링은 AWS KMS keys를 사용하여 데이터 키를 생성, 암호화 및 복호화합니다. AWS Key Management Service (AWS KMS)는 KMS 키를 보호하고 FIPS 경계 내에서 암호화 작업을 수행합니다. 가능하면 AWS KMS 키링 또는 유사한 보안 속성을 가진 키링을 사용할 것을 권장합니다.

키링을 지원하는 모든 프로그래밍 언어 구현은 대칭 암호화 KMS AWS KMS 키를 사용하는 키링을 지원합니다. 다음 프로그래밍 언어 구현은 비대칭 RSA KMS AWS KMS 키를 사용하는 키링도 지원합니다.

- 의 버전 3.x AWS Encryption SDK for Java
- for .NET 버전AWS Encryption SDK 4.x

- 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
- AWS Encryption SDK for Rust 버전 1.x
- Go AWS Encryption SDK 용의 버전 0.1.x 이상

다른 언어 구현의 암호화 키링에 비대칭 KMS 키를 포함하려고 하면 암호화 호출이 실패합니다. 복호화 키링에 포함하면 무시됩니다.

AWS 암호화 CLI의 버전 2.3.x AWS Encryption SDK 및 버전 3.0.x부터 AWS KMS 키링 또는 마스터 키 공급자에서 AWS KMS 다중 리전 키를 사용할 수 있습니다. multi-Region-aware 기호 사용에 대한 자세한 내용과 예제는 섹션을 참조하세요다중 리전 사용 AWS KMS keys. 다중 리전 키에 대한 자세한 내용은 AWS Key Management Service 개발자 가이드의 다중 리전 키 사용을 참조하세요.

> **Note**
>
> 의 KMS 키링에 AWS Encryption SDK 대한 모든 언급은 AWS KMS 키링을 참조합니다.

AWS KMS 키링에는 두 가지 유형의 래핑 키가 포함될 수 있습니다.

- 생성기 키: 일반 텍스트 데이터 키를 생성하고 암호화합니다. 데이터를 암호화하는 키링에는 하나의 생성기 키가 있어야 합니다.
- 추가 키: 생성기 키가 생성한 일반 텍스트 데이터 키를 암호화합니다. AWS KMS 키링에는 0개 이상의 추가 키가 있을 수 있습니다.

메시지를 암호화하려면에 생성기 키가 있어야 합니다. AWS KMS 키링에 KMS 키가 하나만 있는 경우 해당 키는 데이터 키를 생성하고 암호화하는 데 사용됩니다. 복호화할 때 생성기 키는 선택 사항이며 생성기 키와 추가 키의 구분은 무시됩니다.

모든 키링과 마찬가지로 AWS KMS 키링은 독립적으로 사용하거나 동일하거나 다른 유형의 다른 키링과 함께 다중 키링에서 사용할 수 있습니다.

주제

- AWS KMS 키링에 필요한 권한
- AWS KMS 키링 AWS KMS keys 에서 식별
- AWS KMS 키링 생성

- AWS KMS 검색 키링 사용
- AWS KMS 리전 검색 키링 사용

## AWS KMS 키링에 필요한 권한

AWS Encryption SDK 에는가 필요하지 AWS 계정 않으며에 종속되지 않습니다 AWS 서비스. 그러나 AWS KMS 키링을 사용하려면 키링의 AWS KMS keys 에 대해 AWS 계정 및 다음과 같은 최소 권한이 필요합니다.

- AWS KMS 키링으로 암호화하려면 생성기 키에 대한 kms:GenerateDataKey 권한이 필요합니다. AWS KMS 키링의 모든 추가 키에 대해 kms:Encrypt 권한이 필요합니다.
- AWS KMS 키링으로 복호화하려면 AWS KMS 키링에서 하나 이상의 키에 대한 kms:Decrypt 권한이 필요합니다.
- AWS KMS 키링으로 구성된 다중 키링으로 암호화하려면 생성기 키링의 생성기 키에 대한 kms:GenerateDataKey 권한이 필요합니다. 다른 모든 키링의 다른 모든 AWS KMS 키에 대한 kms:Encrypt 권한이 필요합니다.
- 비대칭 RSA AWS KMS 키링으로 암호화하려면 키링을 생성할 때 암호화에 사용할 퍼블릭 키 구성 요소를 지정해야 하므로 kms:GenerateDataKey 또는 kms:Encrypt가 필요하지 않습니다. 이 키링으로 암호화할 때는 AWS KMS 호출이 이루어지지 않습니다. 비대칭 RSA AWS KMS 키링으로 복호화하려면 kms:Decrypt 권한이 필요합니다.

권한에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 KMS 키 액세스 및 권한을 AWS KMS keys참조하세요.

## AWS KMS 키링 AWS KMS keys 에서 식별

AWS KMS 키링에는 하나 이상의가 포함될 수 있습니다 AWS KMS keys. AWS KMS 키링에서를 지정 AWS KMS key 하려면 지원되는 AWS KMS 키 식별자를 사용합니다. 키링 AWS KMS key 에서를 식별하는 데 사용할 수 있는 키 식별자는 작업 및 언어 구현에 따라 다릅니다. AWS KMS key의 키 식별자에 대한 자세한 내용은 AWS Key Management Service 개발자 가이드의 키 식별자를 참조하세요.

작업에 가장 적합한 키 식별자를 사용하는 것이 모범 사례입니다.

- 에 대한 암호화 키링에서 키 ARN 또는 별칭 ARN을 사용하여 KMS 키를 식별할 AWS Encryption SDK for C수 있습니다. 다른 모든 언어 구현에서는 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN을 사용하여 데이터를 암호화할 수 있습니다.

- 복호화 키링에서는 키 ARN을 사용하여 AWS KMS keys를 식별해야 합니다. 이 요구 사항은 AWS Encryption SDK의 모든 언어 구현에 적용됩니다. 자세한 내용은 래핑 키 선택을 참조하세요.
- 암호화 및 복호화에 사용되는 키 링에서는 키 ARN을 사용하여 AWS KMS keys를 식별해야 합니다. 이 요구 사항은 AWS Encryption SDK의 모든 언어 구현에 적용됩니다.

암호화 키링에서 KMS 키에 대해 별칭 이름 또는 별칭 ARN을 지정하는 경우 암호화 작업은 현재 별칭과 연결된 키 ARN을 암호화된 데이터 키의 메타데이터에 저장합니다. 별칭은 저장되지 않습니다. 별칭을 변경해도 암호화된 데이터 키를 복호화하는 데 사용되는 KMS 키에는 영향을 주지 않습니다.

## AWS KMS 키링 생성

동일하거나 다른 AWS 계정 및 AWS KMS keys 에서 단일 AWS KMS key 또는 여러 로 각 AWS KMS 키링을 구성할 수 있습니다 AWS 리전. 는 대칭 암호화 KMS 키(SYMMETRIC_DEFAULT) 또는 비대칭 RSA KMS 키여야 AWS KMS keys 합니다. 대칭 암호화 다중 리전 KMS 키를 사용할 수도 있습니다. 다중 키링에서도 하나 이상의 AWS KMS 키링을 사용할 수 있습니다.

데이터를 암호화 및 복호화하는 AWS KMS 키링을 생성하거나 암호화 또는 복호화 전용 AWS KMS 키링을 생성할 수 있습니다. 데이터를 암호화하는 AWS KMS 키링을 생성할 때는 생성기 키를 지정해야 합니다. 생성기 키는 일반 텍스트 데이터 키를 생성하고 암호화하는 데 AWS KMS key 사용되는 입니다. 데이터 키는 KMS 키와 수학적으로 관련이 없습니다. 그런 다음 선택한 경우 동일한 일반 텍스트 데이터 키를 암호화하는 추가 AWS KMS keys 를 지정할 수 있습니다. 이 키링으로 보호되는 암호화된 필드를 복호화하려면 사용하는 복호화 키링에 키링에 AWS KMS keys 정의된 중 하나 이상이 포함되거나 그렇지 않아야 합니다 AWS KMS keys. (이 없는 AWS KMS 키링 AWS KMS keys 을 AWS KMS 검색 키링이라고 합니다.)

이외의 AWS Encryption SDK 언어 구현에서는 암호화 키링 또는 다중 키링의 AWS Encryption SDK for C모든 래핑 키가 데이터 키를 암호화할 수 있어야 합니다. 래핑 키가 암호화되지 않으면 암호화 메서드가 실패합니다. 따라서 호출자는 키링의 모든 키에 필요한 권한을 가지고 있어야 합니다. 검색 키링을 사용하여 단독 또는 다중 키링으로 데이터를 암호화하는 경우 암호화 작업이 실패합니다. 암호화 AWS Encryption SDK for C작업이 표준 검색 키링을 무시하지만 다중 리전 검색 키링을 단독으로 지정하거나 다중 키링에서 지정하는 경우 실패하는 경우는 예외입니다.

다음 예제에서는 생성기 AWS KMS 키와 하나의 추가 키를 사용하여 키링을 생성합니다. 생성기 키와 추가 키는 모두 대칭 암호화 KMS 키입니다. 이 예제에서는 키 ARN을 사용하여 KMS 키를 식별합니다. 이는 암호화에 사용되는 AWS KMS 키링의 모범 사례이며 복호화에 사용되는 AWS KMS 키링의 요구 사항입니다. 자세한 내용은 AWS KMS 키링 AWS KMS keys 에서 식별을 참조하세요.

C

AWS KMS key 의 암호화 키링에서를 식별하려면 키 ARN 또는 별칭 ARN을 AWS Encryption SDK for C지정합니다. 복호화 키링에서는 키 ARN을 사용해야 합니다. 자세한 내용은 AWS KMS 키링 AWS KMS keys 에서 식별 섹션을 참조하세요.

전체 예를 보려면 string.cpp를 참조하세요.

```
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

AWS Encryption SDK for .NET에서 하나 이상의 KMS 키를 사용하여 키링을 생성하려면 `CreateAwsKmsMultiKeyring()` 메서드를 사용합니다. 이 예제에서는 두 개의 AWS KMS 키를 사용합니다. 한 개의 KMS 키를 지정하려면 `Generator` 파라미터만 사용합니다. 추가 KMS 키를 지정하는 `KmsKeyIds` 파라미터는 선택 사항입니다.

이 키링에 대한 입력은 AWS KMS 클라이언트를 가져오지 않습니다. 대신는 키링에서 KMS 키로 표시되는 각 리전에 기본 AWS KMS 클라이언트를 AWS Encryption SDK 사용합니다. 예를 들어 `Generator` 파라미터 값으로 식별되는 KMS 키가 미국 서부(오레곤) 리전()에 있는 경우 `us-west-2`는 `us-west-2` 리전에 대한 기본 AWS KMS 클라이언트를 AWS Encryption SDK 생성합니다. AWS KMS 클라이언트를 사용자 지정해야 하는 경우 `CreateAwsKmsKeyring()` 메서드를 사용합니다.

AWS Encryption SDK for .NET에서 암호화 키링 AWS KMS key 에 대해를 지정할 때 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN과 같은 유효한 키 식별자를 사용할 수 있습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

다음 예제에서는 for .NET 버전 AWS Encryption SDK 4.x와 `CreateAwsKmsKeyring()` 메서드를 사용하여 AWS KMS 클라이언트를 사용자 지정합니다.

```
// 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";
List<string> additionalKeys = new List<string> { "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321" };

// Instantiate the keyring input object
var createEncryptKeyringInput = new CreateAwsKmsMultiKeyringInput
{
    Generator = generatorKey,
    KmsKeyIds = additionalKeys
};

var kmsEncryptKeyring = mpl.CreateAwsKmsMultiKeyring(createEncryptKeyringInput);
```

JavaScript Browser

에서 암호화 키링에 AWS KMS key 대해를 지정할 때 키 ID AWS Encryption SDK for JavaScript, 키 ARN, 별칭 이름 또는 별칭 ARN과 같은 유효한 키 식별자를 사용할 수 있습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한`buildClient`할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_simple.ts를 참조하세요.

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

에서 암호화 키링에 AWS KMS key 대해를 지정할 때 키 ID AWS Encryption SDK for JavaScript, 키 ARN, 별칭 이름 또는 별칭 ARN과 같은 유효한 키 식별자를 사용할 수 있습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

다음 예제에서는 buildClient 함수를 사용하여 기본 커밋 정책인를 지정합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한buildClient할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

전체 예제는 GitHub의 AWS Encryption SDK for JavaScript 리포지토리에서 kms_simple.ts를 참조하세요.

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

하나 이상의 키가 있는 AWS KMS 키링을 생성하려면 `CreateAwsKmsMultiKeyring()` 메서드를 사용합니다. 이 예제에서는 두 개의 KMS 키를 사용합니다. 한 개의 KMS 키를 지정하려면 `generator` 파라미터만 사용합니다. 추가 KMS 키를 지정하는 `kmsKeyIds` 파라미터는 선택 사항입니다.

이 키링에 대한 입력은 AWS KMS 클라이언트를 가져오지 않습니다. 대신는 키링에서 KMS 키로 표시되는 각 리전에 기본 AWS KMS 클라이언트를 AWS Encryption SDK 사용합니다. 예를 들어 `Generator` 파라미터 값으로 식별되는 KMS 키가 미국 서부(오레곤) 리전()에 있는 경우 `us-west-2`는 `us-west-2` 리전에 대한 기본 AWS KMS 클라이언트를 AWS Encryption SDK 생성합니다. AWS KMS 클라이언트를 사용자 지정해야 하는 경우 `CreateAwsKmsKeyring()` 메서드를 사용합니다.

에서 암호화 키링에 AWS KMS key 대해를 지정할 때 키 ID AWS Encryption SDK for Java, 키 ARN, 별칭 이름 또는 별칭 ARN과 같은 유효한 키 식별자를 사용할 수 있습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

전체 예제는 GitHub의 AWS Encryption SDK for Java 리포지토리에서 BasicEncryptionKeyringExample.java를 참조하세요.

```
// 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 =
 materialProviders.CreateAwsKmsMultiKeyring(keyringInput);
```

Python

하나 이상의 키가 있는 AWS KMS 키링을 생성하려면 `create_aws_kms_multi_keyring()` 메서드를 사용합니다. 이 예제에서는 두 개의 KMS 키를 사용합니다. 한 개의 KMS 키를 지정하려면 `generator` 파라미터만 사용합니다. 추가 KMS 키를 지정하는 `kms_key_ids` 파라미터는 선택 사항입니다.

이 키링에 대한 입력은 AWS KMS 클라이언트를 가져오지 않습니다. 대신는 키링에서 KMS 키로 표시되는 각 리전에 기본 AWS KMS 클라이언트를 AWS Encryption SDK 사용합니다. 예를 들어 `generator` 파라미터 값으로 식별되는 KMS 키가 미국 서부(오레곤) 리전()에 있는 경우 `us-west-2`는 `us-west-2` 리전에 대한 기본 AWS KMS 클라이언트를 AWS Encryption SDK 생성합니다. AWS KMS 클라이언트를 사용자 지정해야 하는 경우 `create_aws_kms_keyring()` 메서드를 사용합니다.

에서 암호화 키링에 AWS KMS key 대해를 지정할 때 키 ID AWS Encryption SDK for Python, 키 ARN, 별칭 이름 또는 별칭 ARN과 같은 유효한 키 식별자를 사용할 수 있습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

다음 예시에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 전체 예제는 GitHub의 AWS Encryption SDK for Python 리포지토리에서 aws_kms_multi_keyring_example.py를 참조하세요.

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "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 material providers library
```

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS keyring
kms_multi_keyring_input: CreateAwsKmsMultiKeyringInput =
 CreateAwsKmsMultiKeyringInput(
    generator="arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    kms_key_ids="arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"
)

kms_multi_keyring: IKeyring = mat_prov.create_aws_kms_multi_keyring(
    input=kms_multi_keyring_input
)
```

Rust

하나 이상의 키가 있는 AWS KMS 키링을 생성하려면 `create_aws_kms_multi_keyring()` 메서드를 사용합니다. 이 예제에서는 두 개의 KMS 키를 사용합니다. 한 개의 KMS 키를 지정하려면 `generator` 파라미터만 사용합니다. 추가 KMS 키를 지정하는 `kms_key_ids` 파라미터는 선택 사항입니다.

이 키링에 대한 입력은 AWS KMS 클라이언트를 가져오지 않습니다. 대신는 키링에서 KMS 키로 표시되는 각 리전에 기본 AWS KMS 클라이언트를 AWS Encryption SDK 사용합니다. 예를 들어 `generator` 파라미터 값으로 식별되는 KMS 키가 미국 서부(오레곤) 리전()에 있는 경우 `us-west-2`는 `us-west-2` 리전에 대한 기본 AWS KMS 클라이언트를 AWS Encryption SDK 생성합니다. AWS KMS 클라이언트를 사용자 지정해야 하는 경우 `create_aws_kms_keyring()` 메서드를 사용합니다.

AWS Encryption SDK for Rust에서 암호화 키링 AWS KMS key 에 대해를 지정할 때 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN과 같은 유효한 키 식별자를 사용할 수 있습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

다음 예시에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 전체 예제는 GitHub의 aws-encryption-sdk 리포지토리 Rust 디렉터리에 있는 aws_kms_keyring_example.rs를 참조하세요. aws-encryption-sdk

```
// Instantiate the AWS Encryption SDK client
```

```
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create an AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create the AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

kms_multi_keyring: IKeyring = mpl.create_aws_kms_multi_keyring(
    input=kms_multi_keyring_input
)
```

Go

하나 이상의 키가 있는 AWS KMS 키링을 생성하려면 `create_aws_kms_multi_keyring()` 메서드를 사용합니다. 이 예제에서는 두 개의 KMS 키를 사용합니다. 한 개의 KMS 키를 지정하려면 `generator` 파라미터만 사용합니다. 추가 KMS 키를 지정하는 `kms_key_ids` 파라미터는 선택 사항입니다.

이 키링에 대한 입력은 AWS KMS 클라이언트를 가져오지 않습니다. 대신는 키링에서 KMS 키로 표시되는 각 리전에 기본 AWS KMS 클라이언트를 AWS Encryption SDK 사용합니다. 예를 들

어 generator 파라미터 값으로 식별되는 KMS 키가 미국 서부(오레곤) 리전()에 있는 경우 us-west-2는 us-west-2 리전에 대한 기본 AWS KMS 클라이언트를 AWS Encryption SDK 생성합니다. AWS KMS 클라이언트를 사용자 지정해야 하는 경우 create_aws_kms_keyring() 메서드를 사용합니다.

Go AWS KMS key 용에서 암호화 키링 AWS Encryption SDK 에 대해를 지정할 때 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN과 같은 유효한 키 식별자를 사용할 수 있습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

다음 예시에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT.

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create an encryption context
encryptionContext := map[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 material providers library
```

```
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create the AWS KMS keyring
awsKmsMultiKeyringInput := mpltypes.CreateAwsKmsMultiKeyringInput{
    Generator: "&arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    KmsKeyIds: []string{"arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"},
}
awsKmsMultiKeyring, err := matProv.CreateAwsKmsMultiKeyring(context.Background(),
 awsKmsMultiKeyringInput)
```

는 비대칭 RSA KMS AWS KMS 키를 사용하는 키링 AWS Encryption SDK 도 지원합니다. 비대칭 RSA AWS KMS 키링에는 키 페어가 하나만 포함될 수 있습니다.

비대칭 RSA AWS KMS 키링으로 암호화하려면 키링을 생성할 때 암호화에 사용할 퍼블릭 키 구성 요소를 지정해야 하므로 kms:GenerateDataKey 또는 kms:Encrypt가 필요하지 않습니다. 이 키링으로 암호화할 때는 AWS KMS 호출이 이루어지지 않습니다. 비대칭 RSA AWS KMS 키링으로 복호화하려면 kms:Decrypt 권한이 필요합니다.

> **Note**
>
> 비대칭 RSA KMS AWS KMS 키를 사용하는 키링을 생성하려면 다음 프로그래밍 언어 구현 중 하나를 사용해야 합니다.
>
> - 의 버전 3.x AWS Encryption SDK for Java
> - for .NET 버전AWS Encryption SDK 4.x
> - 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
> - AWS Encryption SDK for Rust 버전 1.x
> - Go AWS Encryption SDK 용의 버전 0.1.x 이상

다음 예제에서는 `CreateAwsKmsRsaKeyring` 메서드를 사용하여 비대칭 RSA KMS AWS KMS 키로 키링을 생성합니다. 비대칭 RSA AWS KMS 키링을 생성하려면 다음 값을 제공합니다.

- `kmsClient`: 새 AWS KMS 클라이언트 생성
- `kmsKeyID`: 비대칭 RSA KMS 키를 식별하는 키 ARN
- `publicKey`: 전달한 키의 퍼블릭 키를 나타내는 UTF-8 인코딩 PEM 파일의 ByteBuffer `kmsKeyID`
- `encryptionAlgorithm`: 암호화 알고리즘은 `RSAES_OAEP_SHA_256` 또는 이어야 합니다. `RSAES_OAEP_SHA_1`

C# / .NET

비대칭 RSA AWS KMS 키링을 생성하려면 비대칭 RSA KMS 키에서 퍼블릭 키와 프라이빗 키 ARN을 제공해야 합니다. 퍼블릭 키는 PEM으로 인코딩되어야 합니다. 다음 예제에서는 비대칭 RSA AWS KMS 키 페어를 사용하여 키링을 생성합니다.

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

var publicKey = new MemoryStream(Encoding.UTF8.GetBytes(AWS KMS RSA public key));

// Instantiate the keyring input object
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);
```

Java

비대칭 RSA AWS KMS 키링을 생성하려면 비대칭 RSA KMS 키에서 퍼블릭 키와 프라이빗 키 ARN을 제공해야 합니다. 퍼블릭 키는 PEM으로 인코딩되어야 합니다. 다음 예제에서는 비대칭 RSA AWS KMS 키 페어를 사용하여 키링을 생성합니다.

```
// 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("0x0114"),
        // ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA256("0x0146"),
        // ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256("0x0178")

 .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);
```

### Python

비대칭 RSA AWS KMS 키링을 생성하려면 비대칭 RSA KMS 키에서 퍼블릭 키와 프라이빗 키 ARN을 제공해야 합니다. 퍼블릭 키는 PEM으로 인코딩되어야 합니다. 다음 예제에서는 비대칭 RSA AWS KMS 키 페어를 사용하여 키링을 생성합니다.

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Optional: Create an encryption context
```

```
encryption_context: Dict[str, str] = {
    "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 material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS keyring
keyring_input: CreateAwsKmsRsaKeyringInput = CreateAwsKmsRsaKeyringInput(
    public_key="public_key",
    kms_key_id="kms_key_id",
    encryption_algorithm="RSAES_OAEP_SHA_256",
    kms_client=kms_client
)

kms_rsa_keyring: IKeyring = mat_prov.create_aws_kms_rsa_keyring(
    input=keyring_input
)
```

Rust

비대칭 RSA AWS KMS 키링을 생성하려면 비대칭 RSA KMS 키에서 퍼블릭 키와 프라이빗 키 ARN을 제공해야 합니다. 퍼블릭 키는 PEM으로 인코딩되어야 합니다. 다음 예제에서는 비대칭 RSA AWS KMS 키 페어를 사용하여 키링을 생성합니다.

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create an AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
```

```
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create the AWS KMS keyring
let kms_rsa_keyring = mpl
    .create_aws_kms_rsa_keyring()
    .kms_key_id(kms_key_id)
    .public_key(aws_smithy_types::Blob::new(public_key))

 .encryption_algorithm(aws_sdk_kms::types::EncryptionAlgorithmSpec::RsaesOaepSha256)
    .kms_client(kms_client)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
```

```
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[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 material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create the AWS KMS keyring
awsKmsRSAKeyringInput := mpltypes.CreateAwsKmsRsaKeyringInput{
    KmsClient:           kmsClient,
    KmsKeyId:            kmsKeyID,
    PublicKey:           kmsPublicKey,
    EncryptionAlgorithm: kmstypes.EncryptionAlgorithmSpecRsaesOaepSha256,
}
awsKmsRSAKeyring, err := matProv.CreateAwsKmsRsaKeyring(context.Background(),
 awsKmsRSAKeyringInput)
if err != nil {
    panic(err)
}
```

# AWS KMS 검색 키링 사용

복호화할 때는에서 사용할 AWS Encryption SDK 수 있는 래핑 키를 지정하는 것이 가장 좋습니다. 이 모범 사례를 따르려면 AWS KMS 래핑 키를 지정한 키로 제한하는 AWS KMS 복호화 키링을 사용합니다. 그러나 AWS KMS 검색 키링, 즉 래핑 키를 지정하지 않는 AWS KMS 키링을 생성할 수도 있습니다.

는 표준 AWS KMS 검색 키링과 AWS KMS 다중 리전 키에 대한 검색 키링을 AWS Encryption SDK 제공합니다. AWS Encryption SDK에서 다중 리전 키 사용에 관한 정보는 다중 리전 사용 AWS KMS keys 섹션을 참조하세요.

래핑 키를 지정하지 않기 때문에 검색 키링은 데이터를 암호화할 수 없습니다. 검색 키링을 사용하여 단독 또는 다중 키링으로 데이터를 암호화하는 경우 암호화 작업이 실패합니다. 암호화 AWS Encryption SDK for C작업이 표준 검색 키링을 무시하지만 다중 리전 검색 키링을 단독으로 지정하거나 다중 키링에서 지정하는 경우 실패하는 경우는 예외입니다.

복호화할 때 검색 키링을 사용하면 누가 소유하거나 액세스할 수 있는지에 관계없이에서 암호화된 데이터 키를 암호화 AWS KMS key 한를 사용하여 AWS KMS 복호화하도록 AWS Encryption SDK 요청할 수 있습니다 AWS KMS key. 호출자가 AWS KMS key에 대한 `kms:Decrypt` 권한이 있는 경우에만 호출이 성공합니다.

> **Important**
>
> 복호화 다중 키링에 AWS KMS 검색 키링을 포함하는 경우 검색 키링은 다중 키링의 다른 키링에서 지정한 모든 KMS 키 제한을 재정의합니다. ??? 다중 키링은 제한이 가장 적은 키링처럼 동작합니다. AWS KMS 검색 키링은 단독으로 사용되거나 다중 키링에 사용되는 경우 암호화에 영향을 주지 않습니다.

는 편의를 위해 AWS KMS 검색 키링을 AWS Encryption SDK 제공합니다. 단, 다음과 같은 이유로 가능하면 더 제한적인 키링을 사용하는 것이 좋습니다.

- 인증 - AWS KMS 검색 키링은 암호화된 메시지의 데이터 키를 암호화하는 데 AWS KMS key 사용된 모든를 사용할 수 있으므로 호출자는 이를 AWS KMS key 사용하여 복호화할 수 있는 권한이 있습니다. 호출 AWS KMS key 자가 사용하려는이 아닐 수 있습니다. 예를 들어 암호화된 데이터 키 중 하나가 누구나 사용할 수 AWS KMS key 있는 덜 안전한에서 암호화되었을 수 있습니다.
- 지연 시간 및 성능 - AWS KMS 가 다른 AWS 계정 및 리전 AWS KMS keys 에서 로 암호화된 키를 포함하여 암호화된 모든 데이터 키를 복호화하려고 AWS Encryption SDK 하고 AWS KMS keys 호출자에게 복호화에 사용할 권한이 없기 때문에 검색 키링이 다른 키링보다 느릴 수 있습니다.

검색 키링을 사용하는 경우 검색 필터를 사용하여 지정된 AWS 계정 및 파티션의 키로 사용할 수 있는 KMS 키를 제한하는 것이 좋습니다. 검색 필터는 AWS Encryption SDK버전 1.7.x 이상에서 지원됩니다. 계정 ID 및 파티션을 찾는 데 도움이 필요하면의 AWS 계정 식별자 및 ARN 형식을 참조하세요AWS 일반 참조.

다음 코드는가 사용할 AWS Encryption SDK 수 있는 KMS AWS KMS 키를 aws 파티션 및 111122223333 예제 계정의 키로 제한하는 검색 필터를 사용하여 검색 키링을 인스턴스화합니다.

이 코드를 사용하기 전에 예제 AWS 계정 및 파티션 값을 AWS 계정 및 파티션의 유효한 값으로 바꿉니다. KMS 키가 중국 리전에 있는 경우 aws-cn 파티션 값을 사용하세요. KMS 키가 AWS GovCloud (US) Regions에 있는 경우 aws-us-gov 파티션 값을 사용하세요. 다른 AWS 리전에 있는 경우 aws 파티션 값을 사용하세요.

C

전체 예제는 kms_discovery.cpp를 참조하세요.

```
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

다음 예제에서는 AWS Encryption SDK for .NET 4.x 버전을 사용합니다.

```
// 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" };

// 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 =
 mpl.CreateAwsKmsDiscoveryKeyring(kmsDiscoveryKeyringInput);
```

JavaScript Browser

JavaScript에서는 명시적으로 검색 속성을 지정해야 합니다.

다음 예제에서는 buildClient 함수를 사용하여 기본 커밋 정책인를 지정합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한buildClient할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

```
import {
  KmsKeyringBrowser,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const clientProvider = getClient(KMS, { credentials })

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

JavaScript Node.js

JavaScript에서는 명시적으로 검색 속성을 지정해야 합니다.

다음 예제에서는 buildClient 함수를 사용하여 기본 커밋 정책인를 지정합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 를 사용하여 암호화된 메시지의 암호화된 데이터 키

수를 제한buildClient할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const discovery = true

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

Java

```
// 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);
```

Python

```
# Instantiate the AWS Encryption SDK
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

```
# Create a boto3 client for AWS KMS
kms_client = boto3.client('kms', region_name=aws_region)

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "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 material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS discovery keyring
discovery_keyring_input: CreateAwsKmsDiscoveryKeyringInput =
 CreateAwsKmsDiscoveryKeyringInput(
    kms_client=kms_client,
    discovery_filter=DiscoveryFilter(
        account_ids=[aws_account_id],
        partition="aws"
    )
)

discovery_keyring: IKeyring = mat_prov.create_aws_kms_discovery_keyring(
    input=discovery_keyring_input
)
```

### Rust

```
// Instantiate the AWS Encryption SDK
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create a AWS KMS client.
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);
```

```
// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .account_ids(vec![aws_account_id.to_string()])
    .partition("aws".to_string())
    .build()?;

// Create the AWS KMS discovery keyring
let discovery_keyring = mpl
    .create_aws_kms_discovery_keyring()
    .kms_client(kms_client.clone())
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
```

```
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[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 material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create discovery filter
discoveryFilter := mpltypes.DiscoveryFilter{
    AccountIds: []string{kmsKeyAccountID},
    Partition:  "aws",
}
awsKmsDiscoveryKeyringInput := mpltypes.CreateAwsKmsDiscoveryKeyringInput{
    KmsClient:       kmsClient,
    DiscoveryFilter: &discoveryFilter,
}
awsKmsDiscoveryKeyring, err :=
 matProv.CreateAwsKmsDiscoveryKeyring(context.Background(),
 awsKmsDiscoveryKeyringInput)
if err != nil {
    panic(err)
}
```

# AWS KMS 리전 검색 키링 사용

AWS KMS 리전 검색 키링은 KMS 키의 ARN을 지정하지 않는 키링입니다. 대신가 특히 KMS 키만 사용하여 복호화 AWS Encryption SDK 할 수 있습니다 AWS 리전.

AWS KMS 리전 검색 키링을 사용하여 복호화할 때는 지정된 AWS KMS key 의에서 암호화된 모든 데이터 키를 AWS Encryption SDK 복호화합니다 AWS 리전. 성공하려면 호출자에게 데이터 키를 암호화 AWS 리전 한 지정된의 중 하나 이상 AWS KMS keys 에 대한 `kms:Decrypt` 권한이 있어야 합니다.

다른 검색 키링과 마찬가지로 리전 검색 키링은 암호화에 영향을 주지 않습니다. 암호화된 메시지를 복호화할 때만 작동합니다. 암호화 및 복호화에 사용되는 다중 키링에서 리전 검색 키링을 사용하는 경우 복호화 시에만 유효합니다. 다중 리전 검색 키링을 사용하여 단독 또는 다중 키링으로 데이터를 암호화하는 경우 암호화 작업이 실패합니다.

> **Important**
>
> 복호화 다중 키링에 AWS KMS 리전 검색 키링을 포함하는 경우 리전 검색 키링은 다중 키링의 다른 키링에서 지정한 모든 KMS 키 제한을 재정의합니다. ??? 다중 키링은 제한이 가장 적은 키링처럼 동작합니다. AWS KMS 검색 키링은 단독으로 사용되거나 다중 키링에 사용되는 경우 암호화에 영향을 주지 않습니다.

의 리전 검색 키링은 지정된 리전의 KMS 키로만 복호화를 AWS Encryption SDK for C 시도합니다. AWS Encryption SDK for JavaScript 및 AWS Encryption SDK for .NET에서 검색 키링을 사용하는 경우 AWS KMS 클라이언트에서 리전을 구성합니다. 이러한 AWS Encryption SDK 구현 AWS KMS 은 리전별로 KMS 키를 필터링하지 않지만 지정된 리전 외부의 KMS 키에 대한 복호화 요청에 실패합니다.

검색 키링을 사용하는 경우 검색 필터를 사용하여 복호화에 사용되는 KMS 키를 지정된 AWS 계정 및 파티션의 키로 제한하는 것이 좋습니다. 검색 필터는 AWS Encryption SDK버전 1.7.x 이상에서 지원됩니다.

예를 들어 다음 코드는 검색 필터를 사용하여 AWS KMS 리전 검색 키링을 생성합니다. 이 키링은 미국 서부(오레곤) 리전(us-west-2)의 계정 111122223333에서를 KMS 키로 제한 AWS Encryption SDK 합니다.

C

작동 예제에서 이 키링과 `create_kms_client` 메서드를 보려면 kms_discovery.cpp를 참조하세요.

```
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

AWS Encryption SDK for .NET에는 전용 리전 검색 키링이 없습니다. 단, 여러 기술을 사용하여 복호화할 때 사용되는 KMS 키를 특정 리전으로 제한할 수 있습니다.

검색 키링에서 리전을 제한하는 가장 효율적인 방법은 단일 리전 키만 사용하여 데이터를 암호화한 경우에도 다중 리전 인식 검색 키링을 사용하는 것입니다. 단일 리전 키가 발견되면 다중 리전 인식 키링은 다중 리전 기능을 사용하지 않습니다.

`CreateAwsKmsMrkDiscoveryKeyring()` 메서드에서 반환된 키링은 AWS KMS를 호출하기 전에 리전별로 KMS 키를 필터링합니다. 암호화된 데이터 키가 `CreateAwsKmsMrkDiscoveryKeyringInput` 객체의 `Region` 파라미터로 지정된 리전의 KMS 키로 암호화된 AWS KMS 경우에만에 복호화 요청을 보냅니다.

다음 예제에서는 AWS Encryption SDK for .NET 4.x 버전을 사용합니다.

```
// 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 =
 mpl.CreateAwsKmsMrkDiscoveryKeyring(regionalDiscoveryKeyringInput);
```

AWS KMS 클라이언트(AmazonKeyManagementServiceClient)의 인스턴스에 리전을 지정 AWS 리전 하여 KMS 키를 특정 로 제한할 수도 있습니다. 단, 이 구성은 multi-Region-aware 검색 키링을 사용하는 것보다 효율성이 떨어지고 비용이 더 많이 들 수 있습니다. 호출하기 전에 리전별로 KMS 키를 필터링하는 대신 AWS Encryption SDK for .NET AWS KMS은 암호화된 각 데이터 키에 AWS KMS 대해 (복호화할 때까지)를 호출하고 AWS KMS 를 사용하여 사용하는 KMS 키를 지정된 리전으로 제한합니다.

다음 예제에서는 AWS Encryption SDK for .NET 4.x 버전을 사용합니다.

```
// 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 =
 mlp.CreateAwsKmsDiscoveryKeyring(createRegionalDiscoveryKeyringInput);
```

### JavaScript Browser

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키

수를 제한buildClient할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

다음 예제에서는 buildClient 함수를 사용하여 기본 커밋 정책인를 지정합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한buildClient할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

이 키링과 limitRegions 함수를 보려면 실제 예제에서 kms_regional_discovery.ts를 참조하세요.

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
```

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

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);
```

Python

```
# Instantiate the AWS Encryption SDK
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Create a boto3 client for AWS KMS
kms_client = boto3.client('kms', region_name=aws_region)

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "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 material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
```

```
    config=MaterialProvidersConfig()
)

# Create the AWS KMS regional discovery keyring
regional_discovery_keyring_input: CreateAwsKmsMrkDiscoveryKeyringInput = \
    CreateAwsKmsMrkDiscoveryKeyringInput(
        kms_client=kms_client,
        region=mrk_replica_decrypt_region,
        discovery_filter=DiscoveryFilter(
            account_ids=[111122223333],
            partition="aws"
        )
)

    regional_discovery_keyring: IKeyring =
 mat_prov.create_aws_kms_mrk_discovery_keyring(
        input=regional_discovery_keyring_input
    )
```

Rust

```
// Instantiate the AWS Encryption SDK
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create an AWS KMS client
let decrypt_kms_config = aws_sdk_kms::config::Builder::from(&sdk_config)
```

```
    .region(Region::new(mrk_replica_decrypt_region.clone()))
    .build();
let decrypt_kms_client = aws_sdk_kms::Client::from_conf(decrypt_kms_config);


// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .account_ids(vec![aws_account_id.to_string()])
    .partition("aws".to_string())
    .build()?;

// Create the regional discovery keyring
let discovery_keyring = mpl
    .create_aws_kms_mrk_discovery_keyring()
    .kms_client(decrypt_kms_client)
    .region(mrk_replica_decrypt_region)
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
```

```
// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[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",
}

// Create discovery filter
discoveryFilter := mpltypes.DiscoveryFilter{
    AccountIds: []string{awsAccountID},
    Partition:  "aws",
}

// Create the regional discovery keyring
awsKmsMrkDiscoveryInput := mpltypes.CreateAwsKmsMrkDiscoveryKeyringInput{
    KmsClient:       kmsClient,
    Region:          alternateRegionMrkKeyRegion,
    DiscoveryFilter: &discoveryFilter,
}
awsKmsMrkDiscoveryKeyring, err :=
 matProv.CreateAwsKmsMrkDiscoveryKeyring(context.Background(),
 awsKmsMrkDiscoveryInput)
if err != nil {
    panic(err)
}
```

AWS Encryption SDK for JavaScript 또한는 Node.js 및 브라우저에 대한 `excludeRegions` 함수를 내보냅니다. 이 함수는 특정 AWS KMS 리전 AWS KMS keys 에서 생략되는 리전 검색 키링을 생성합니다. 다음 예제에서는 미국 동부(버지니아 북부)(us-east-1)를 AWS 리전 제외한 모든 AWS KMS keys 의 계정 111122223333에서를 사용할 수 있는 AWS KMS 리전 검색 키링을 생성합니다.

에는 유사한 메서 AWS Encryption SDK for C 드가 없지만 사용자 지정 ClientSupplier를 생성하여 구현할 수 있습니다.

이 예는 Node.js에 대한 코드를 보여줍니다.

```
const discovery = true
const clientProvider = excludeRegions(['us-east-1'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: [111122223333], partition: 'aws' }
})
```

# AWS KMS 계층적 키링

AWS KMS 계층적 키링을 사용하면 데이터를 암호화하거나 해독할 AWS KMS 때마다를 호출하지 않고도 대칭 암호화 KMS 키로 암호화 자료를 보호할 수 있습니다. 호출을 최소화해야 하는 애플리케이션 AWS KMS과 보안 요구 사항을 위반하지 않고 일부 암호화 자료를 재사용할 수 있는 애플리케이션에 적합합니다.

계층적 키링은 Amazon DynamoDB 테이블에 유지되는 AWS KMS 보호된 브랜치 키를 사용한 다음 암호화 및 복호화 작업에 사용되는 브랜치 키 자료를 로컬로 캐싱하여 AWS KMS 호출 수를 줄이는 암호화 자료 캐싱 솔루션입니다. DynamoDB 테이블은 브랜치 키를 관리하고 보호하는 키 스토어 역할을 합니다. 활성 브랜치 키와 모든 이하 버전의 브랜치 키를 저장합니다. 활성 브랜치 키는 최신 버전의 브랜치 키입니다. 계층적 키링은 고유한 데이터 키를 사용하여 각 메시지를 암호화하고 각 암호화 요청에 대한 각 데이터 암호화 키를 암호화하며 활성 브랜치 키에서 파생된 고유한 래핑 키로 각 데이터 암호화 키를 암호화합니다. 계층적 키링은 활성 브랜치 키와 파생된 래핑 키 사이에 설정된 계층 구조에 따라 달라집니다.

계층적 키링은 일반적으로 각 브랜치 키 버전을 사용하여 여러 요청을 충족합니다. 하지만 활성 브랜치 키의 재사용 범위를 제어하고 활성 브랜치 키의 교체 빈도를 결정할 수 있습니다. 브랜치 키의 활성 버전은 교체할 때까지 활성 상태로 유지됩니다. 이하 버전의 활성 브랜치 키는 암호화 작업을 수행하는 데 사용되지 않지만 여전히 쿼리를 통해 복호화 작업에 사용할 수 있습니다.

계층적 키링을 인스턴스화하면 로컬 캐시가 생성됩니다. 캐시 제한을 지정하고 브랜치 키 자료가 만료되어 캐시에서 제거되기 전에 로컬 캐시에 저장되는 최대 시간을 정의합니다. 계층적 키링은가 작업에 처음 `branch-key-id` 지정될 때 브랜치 키를 해독하고 브랜치 키 구성 요소를 어셈블하기 위해 한 AWS KMS 번 호출합니다. 그러면 브랜치 키 자료가 로컬 캐시에 저장되고 캐시 제한이 만료될 때까지 `branch-key-id`를 지정하는 모든 암호화 및 복호화 작업에 브랜치 키 자료가 재사용됩니다. 로컬 캐

시에 브랜치 키 구성 요소를 저장하면 AWS KMS 호출이 줄어듭니다. 예를 들어, 캐시 한도를 15분으로 가정해 보겠습니다. 해당 캐시 제한 내에서 10,000개의 암호화 작업을 수행하는 경우 기존 AWS KMS 키링은 10,000개의 암호화 작업을 충족하기 위해 10,000개의 AWS KMS 호출을 수행해야 합니다. 활성가 하나 있는 경우 계층적 키링은 10`branch-key-id`,000개의 암호화 작업을 충족하기 위해 한 번만 AWS KMS 호출하면 됩니다.

로컬 캐시는 암호화 자료를 복호화 자료와 분리합니다. 암호화 구성 요소는 활성 브랜치 키에서 수집되며 캐시 제한이 만료될 때까지 모든 암호화 작업에 재사용됩니다. 복호화 자료는 암호화된 필드의 메타데이터에서 식별된 브랜치 키 ID 및 버전에서 수집되며 캐시 제한이 만료될 때까지 브랜치 키 ID 및 버전과 관련된 모든 복호화 작업에 재사용됩니다. 로컬 캐시는 한 번에 여러 버전의 동일한 브랜치 키를 저장할 수 있습니다. 로컬 캐시가를 사용하도록 구성된 경우 branch key ID supplier한 번에 여러 활성 브랜치 키의 브랜치 키 구성 요소를 저장할 수도 있습니다.

> **Note**
>
> 의 계층적 키링에 대한 모든 언급은 AWS KMS 계층적 키링을 AWS Encryption SDK 참조합니다.

### 프로그래밍 언어 호환성

계층적 키링은 다음 프로그래밍 언어 및 버전에서 지원됩니다.

- 의 버전 3.x AWS Encryption SDK for Java
- for .NET 버전 AWS Encryption SDK 4.x
- 선택적 MPL 종속성과 함께 사용되는 AWS Encryption SDK for Python경우 버전 4.x.
- AWS Encryption SDK for Rust 버전 1.x
- Go AWS Encryption SDK 용의 버전 0.1.x 이상

### 주제

# 작동 방식

다음 연습에서는 계층적 키링이 암호화 및 복호화 자료를 조합하는 방법과 암호화 및 복호화 작업에 대해 키링이 수행하는 다양한 호출을 설명합니다. 래핑 키 파생 및 일반 텍스트 데이터 키 암호화 프로세스에 대한 기술 세부 정보는 AWS KMS 계층적 키링 기술 세부 정보를 참조하세요.

암호화 및 서명

다음 연습에서는 계층적 키링이 암호화 자료를 조합하고 고유한 래핑 키를 도출하는 방법을 설명합니다.

1. 암호화 메서드는 계층적 키링에 암호화 자료를 요청합니다. 키링은 일반 텍스트 데이터 키를 생성한 다음 로컬 캐시에 유효한 브랜치 자료가 있는지 확인하여 래핑 키를 생성합니다. 유효한 브랜치 키 구성 요소가 있는 경우 키링은 4단계로 진행됩니다.
2. 유효한 브랜치 키 구성 요소가 없는 경우 계층적 키링은 키 스토어에서 활성 브랜치 키를 쿼리합니다.

    a. 키 스토어는를 호출 AWS KMS 하여 활성 브랜치 키를 해독하고 일반 텍스트 활성 브랜치 키를 반환합니다. 활성 브랜치 키를 식별하는 데이터는 직렬화되어 AWS KMS복호화 호출 시 추가 인증 데이터(AAD)를 제공합니다.

    b. 키 스토어는 브랜치 키 버전과 같이 일반 텍스트 브랜치 키와 이를 식별하는 데이터를 반환합니다.
3. 계층적 키링은 브랜치 키 자료(일반 텍스트 브랜치 키 및 브랜치 키 버전)를 조합하여 로컬 캐시에 사본을 저장합니다.
4. 계층적 키링은 일반 텍스트 브랜치 키와 16바이트 무작위 솔트에서 고유한 래핑 키를 가져옵니다. 파생된 래핑 키를 사용하여 일반 텍스트 데이터 키의 사본을 암호화합니다.

암호화 메서드로 암호화 자료를 사용하여 데이터를 암호화합니다. 자세한 내용은 AWS Encryption SDK 가 데이터를 암호화하는 방법을 참조하세요.

복호화 및 확인

다음 안내에서는 계층적 키링이 복호화 자료를 조합하고 암호화된 데이터 키를 복호화하는 방법을 설명합니다.

1. 복호화 메서드는 암호화된 메시지에서 암호화된 데이터 키를 식별하여 계층적 키링에 전달합니다.

2. 계층적 키링은 브랜치 키 버전, 16바이트 솔트 및 데이터 키가 암호화된 방법을 설명하는 기타 정보 등 암호화된 데이터 키를 식별하는 데이터를 역직렬화합니다.

   자세한 내용은 AWS KMS 계층적 키링 기술 세부 정보 섹션을 참조하세요.

3. 계층적 키링은 2단계에서 식별한 브랜치 키 버전과 일치하는 유효한 브랜치 키 자료가 로컬 캐시에 있는지 확인합니다. 유효한 브랜치 키 자료가 있는 경우 키링은 6단계로 진행됩니다.

4. 유효한 브랜치 키 구성 요소가 없는 경우 계층적 키 링은 키 스토어에서 2단계에서 식별된 브랜치 키 버전과 일치하는 브랜치 키를 쿼리합니다.

   a. 키 스토어는를 호출 AWS KMS 하여 브랜치 키를 해독하고 일반 텍스트 활성 브랜치 키를 반환합니다. 활성 브랜치 키를 식별하는 데이터는 직렬화되어 AWS KMS복호화 호출 시 추가 인증 데이터(AAD)를 제공합니다.

   b. 키 스토어는 브랜치 키 버전과 같이 일반 텍스트 브랜치 키와 이를 식별하는 데이터를 반환합니다.

5. 계층적 키링은 브랜치 키 자료(일반 텍스트 브랜치 키 및 브랜치 키 버전)를 조합하여 로컬 캐시에 사본을 저장합니다.

6. 계층적 키링은 조합된 브랜치 키 자료와 2단계에서 식별한 16바이트 솔트를 사용하여 데이터 키를 암호화한 고유 래핑 키를 재현합니다.

7. 계층적 키링은 재생된 래핑 키를 사용하여 데이터 키를 복호화하고 일반 텍스트 데이터 키를 반환합니다.

복호화 메서드는 복호화 자료와 일반 텍스트 데이터 키를 사용하여 암호화된 메시지를 복호화합니다. 자세한 내용은 가 암호화된 메시지를 AWS Encryption SDK 해독하는 방법을 참조하세요.

## 사전 조건

계층적 키링을 생성하고 사용하기 전에 다음 사전 조건이 충족되는지 확인합니다.

- 사용자 또는 키 스토어 관리자가 키 스토어를 생성하고 하나 이상의 활성 브랜치 키를 생성했습니다.
- 키 스토어 작업을 구성했습니다.

> **Note**
>
> 키 스토어 작업을 구성하는 방법에 따라 수행할 수 있는 작업과 계층적 키 링에서 사용할 수 있는 KMS 키가 결정됩니다. 자세한 내용은 키 스토어 작업을 참조하세요.

- 키 스토어 및 브랜치 키에 액세스하고 사용하는 데 필요한 AWS KMS 권한이 있습니다. 자세한 내용은 the section called “필수 권한” 단원을 참조하십시오.
- 지원되는 캐시 유형을 검토하고 필요에 가장 적합한 캐시 유형을 구성했습니다. 자세한 내용은 the section called “캐시 선택” 섹션을 참조하세요.

## 필수 권한

AWS Encryption SDK 에는가 필요하지 AWS 계정 않으며에 종속되지 않습니다 AWS 서비스. 그러나 계층적 키링을 사용하려면 키 스토어의 대칭 암호화 AWS KMS key(들)에 대한 AWS 계정 및 다음과 같은 최소 권한이 필요합니다.

- 계층적 키링으로 데이터를 암호화하고 해독하려면 kms:Decrypt가 필요합니다.
- 브랜치 키를 생성하고 교체하려면 kms:GenerateDataKeyWithoutPlaintext 및 kms:ReEncrypt가 필요합니다.

브랜치 키 및 키 스토어에 대한 액세스 제어에 대한 자세한 내용은 섹션을 참조하세요the section called “최소 권한 구현”.

## 캐시 선택

계층적 키링은 암호화 및 복호화 작업에 사용되는 브랜치 키 자료를 로컬로 캐싱 AWS KMS 하여에 대한 호출 수를 줄입니다. 계층적 키링을 생성하기 전에 사용할 캐시 유형을 결정해야 합니다. 기본 캐시를 사용하거나 필요에 맞게 캐시를 사용자 지정할 수 있습니다.

계층적 키링은 다음 캐시 유형을 지원합니다.

- the section called “기본 캐시”
- the section called “MultiThreaded 캐시”
- the section called “StormTracking 캐시”
- the section called “공유 캐시”

> **Important**
>
> 지원되는 모든 캐시 유형은 멀티스레드 환경을 지원하도록 설계되었습니다.
> 그러나와 함께 사용하는 경우 AWS Encryption SDK for Python계층적 키링은 다중 스레드 환경을 지원하지 않습니다. 자세한 내용은 GitHub의 aws-cryptographic-material-providers-

library 리포지토리에 있는 Python README.rst 파일을 참조하세요. aws-cryptographic-material-providers-library

## 기본 캐시

대부분 사용자의 경우 기본 캐시로 스레딩 요구 사항을 충족합니다. 기본 캐시는 멀티스레드가 많은 환경을 지원하도록 설계되었습니다. 브랜치 키 자료 항목이 만료되면 기본 캐시는 브랜치 키 자료 항목이 10초 전에 만료될 것임을 한 스레드에 알림 AWS KMS 으로써 여러 스레드가를 호출하는 것을 방지합니다. 이렇게 하면 한 스레드만 AWS KMS 에 캐시 새로 고침 요청을 보냅니다.

기본 캐시와 StormTracking 캐시는 동일한 스레드 모델을 지원하지만 기본 캐시를 사용하려면 입력 용량만 지정하면 됩니다. 보다 세분화된 캐시 사용자 지정을 위해를 사용합니다the section called “StormTracking 캐시”.

로컬 캐시에 저장할 수 있는 브랜치 키 자료 항목의 수를 사용자 지정하려는 경우가 아니라면 계층적 키링을 생성할 때 캐시 유형을 지정할 필요가 없습니다. 캐시 유형을 지정하지 않으면 계층적 키링은 기본 캐시 유형을 사용하고 항목 용량을 1000으로 설정합니다.

기본 캐시를 사용자 지정하려면 다음 값을 지정합니다.

- 항목 용량: 로컬 캐시에 저장할 수 있는 브랜치 키 자료 항목의 수를 제한합니다.

Java

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

C# / .NET

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

Python

```
default_cache = CacheTypeDefault(
```

```
    value=DefaultCache(
        entry_capacity=100
    )
)
```

Rust

```
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);
```

Go

```
cache := mpltypes.CacheTypeMemberDefault{
  Value: mpltypes.DefaultCache{
   EntryCapacity: 100,
  },
 }
```

### MultiThreaded 캐시

MultiThreaded 캐시는 멀티스레드 환경에서 안전하게 사용할 수 있지만 AWS KMS 또는 Amazon DynamoDB 호출을 최소화하는 기능은 제공하지 않습니다. 따라서 브랜치 키 자료 입력이 만료되면 동시에 모든 스레드로 알림이 전송됩니다. 이로 인해 캐시를 새로 고치는 AWS KMS 호출이 여러 번 발생할 수 있습니다.

MultiThreaded 캐시를 사용하려면 다음 값을 지정합니다.

- 항목 용량: 로컬 캐시에 저장할 수 있는 브랜치 키 자료 항목의 수를 제한합니다.
- 항목 정리 테일 크기: 항목 용량에 도달한 경우 정리할 항목 수를 정의합니다.

Java

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

```
        .build())
```

### C# / .NET

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

### Python

```
multithreaded_cache = CacheTypeMultiThreaded(
    value=MultiThreadedCache(
        entry_capacity=100,
        entry_pruning_tail_size=1
    )
)
```

### Rust

```
CacheType::MultiThreaded(
            MultiThreadedCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .build()?)
```

### Go

```
var entryPruningTailSize int32 = 1
 cache := mpltypes.CacheTypeMemberMultiThreaded{
  Value: mpltypes.MultiThreadedCache{
   EntryCapacity:        100,
   EntryPruningTailSize: &entryPruningTailSize,
  },
 }
```

## StormTracking 캐시

StormTracking은 멀티스레드가 많은 환경을 지원하도록 설계되었습니다. 브랜치 키 자료 항목이 만료되면 StormTracking 캐시는 브랜치 키 자료 항목이 미리 만료될 것임을 한 스레드 AWS KMS 에 알려 여러 스레드가를 호출하는 것을 방지합니다. 이렇게 하면 한 스레드만 AWS KMS 에 캐시 새로 고침 요청을 보냅니다.

StormTracking 캐시를 사용하려면 다음 값을 지정합니다.

- 항목 용량: 로컬 캐시에 저장할 수 있는 브랜치 키 자료 항목의 수를 제한합니다.

  기본값: 항목 1,000개
- 항목 정리 테일 크기: 한 번에 정리할 브랜치 키 자료 항목의 수를 정의합니다.

  기본값: 항목 1개
- 유예 기간: 브랜치 키 자료를 새로 고치려는 시도가 만료되기까지 걸리는 시간(초)을 정의합니다.

  기본값: 10초
- 유예 간격: 브랜치 키 자료의 새로 고침 시도 간격(초)을 정의합니다.

  기본값: 1초
- 팬아웃: 브랜치 키 자료를 새로 고칠 수 있는 동시 시도 횟수를 정의합니다.

  기본값: 20회 시도
- 전송 유지 시간(TTL): 브랜치 키 자료를 새로 고치려는 시도가 제한 시간 초과될 때까지의 시간(초)을 정의합니다. `GetCacheEntry`에 대한 응답으로 캐시가 `NoSuchEntry`를 반환할 때마다 해당 브랜치 키는 `PutCache` 항목과 동일한 키가 기록될 때까지 전송 중인 것으로 간주됩니다.

  기본값: 10초
- 절전 모드:를 `fanOut` 초과하면 스레드가 절전 모드로 전환해야 하는 밀리초 수를 정의합니다.

  기본값: 20밀리초

Java

```
.cache(CacheType.builder()
        .StormTracking(StormTrackingCache.builder()
        .entryCapacity(100)
```

```
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(10)
        .sleepMilli(20)
        .build())
```

### C# / .NET

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

### Python

```
storm_tracking_cache = CacheTypeStormTracking(
    value=StormTrackingCache(
        entry_capacity=100,
        entry_pruning_tail_size=1,
        fan_out=20,
        grace_interval=1,
        grace_period=10,
        in_flight_ttl=10,
        sleep_milli=20
    )
)
```

### Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
```

```
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
                    .sleep_milli(20)
                    .build()?)
```

Go

```
var entryPruningTailSize int32 = 1
 cache := mpltypes.CacheTypeMemberStormTracking{
  Value: mpltypes.StormTrackingCache{
   EntryCapacity:        100,
   EntryPruningTailSize: &entryPruningTailSize,
   GraceInterval:        1,
   GracePeriod:          10,
   FanOut:               20,
   InFlightTTL:          10,
   SleepMilli:           20,
  },
 }
```

## 공유 캐시

기본적으로 계층적 키링은 키링을 인스턴스화할 때마다 새 로컬 캐시를 생성합니다. 그러나 공유 캐시를 사용하면 여러 계층적 키링에서 캐시를 공유할 수 있으므로 메모리를 절약할 수 있습니다. 공유 캐시는 인스턴스화하는 각 계층적 키링에 대해 새 암호화 자료 캐시를 생성하는 대신 메모리에 하나의 캐시만 저장하며, 이를 참조하는 모든 계층적 키링에서 사용할 수 있습니다. 공유 캐시는 키링 간에 암호화 자료가 중복되는 것을 방지하여 메모리 사용을 최적화하는 데 도움이 됩니다. 대신 계층적 키링은 동일한 기본 캐시에 액세스하여 전체 메모리 공간을 줄일 수 있습니다.

공유 캐시를 생성할 때도 캐시 유형을 정의합니다. the section called “기본 캐시”, the section called “MultiThreaded 캐시”또는를 캐시 유형the section called “StormTracking 캐시”으로 지정하거나 호환되는 사용자 지정 캐시를 대체할 수 있습니다.

### 파티션

여러 계층적 키링은 단일 공유 캐시를 사용할 수 있습니다. 공유 캐시를 사용하여 계층적 키링을 생성할 때 선택적 파티션 ID를 정의할 수 있습니다. 파티션 ID는 캐시에 쓰는 계층적 키링을 구분합니다. 두 계층적 키링이 동일한 파티션 ID, logical key store name및 브랜치 키 ID를 참조하는 경우 두 키링은 캐시에서 동일한 캐시 항목을 공유합니다. 동일한 공유 캐시와 다른 파티션 IDs로 두 개의 계층적 키링을 생성하는 경우 각 키링은 공유 캐시 내의 지정된 자체 파티션에서만 캐시 항목에 액세스합니다. 파티션은 공유 캐시 내에서 논리적 분할 역할을 하므로 각 계층적 키링이 다른 파티션에 저장된 데이터를 방해하지 않고 자체 지정된 파티션에서 독립적으로 작동할 수 있습니다.

파티션에서 캐시 항목을 재사용하거나 공유하려는 경우 고유한 파티션 ID를 정의해야 합니다. 파티션 ID를 계층적 키링에 전달하면 키링은 브랜치 키 자료를 다시 검색하고 다시 승인할 필요 없이 공유 캐시에 이미 있는 캐시 항목을 재사용할 수 있습니다. 파티션 ID를 지정하지 않으면 계층적 키링을 인스턴스화할 때마다 고유한 파티션 ID가 키링에 자동으로 할당됩니다.

다음 절차에서는 기본 캐시 유형으로 공유 캐시를 생성하고 계층적 키링에 전달하는 방법을 보여줍니다.

1. 재료 공급자 라이브러리CryptographicMaterialsCache(MPL)를 사용하여 (CMC)를 생성합니다. https://github.com/aws/aws-cryptographic-material-providers-library

   Java

   ```
   // Instantiate the MPL
   final MaterialProviders matProv =
       MaterialProviders.builder()
           .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
           .build();

   // Create a CacheType object for the Default cache
   final CacheType cache =
       CacheType.builder()
           .Default(DefaultCache.builder().entryCapacity(100).build())
           .build();

   // Create a CMC using the default cache
   final CreateCryptographicMaterialsCacheInput cryptographicMaterialsCacheInput =
       CreateCryptographicMaterialsCacheInput.builder()
           .cache(cache)
           .build();

   final ICryptographicMaterialsCache sharedCryptographicMaterialsCache =
       matProv.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
   ```

### C# / .NET

```
// Instantiate the MPL
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create a CacheType object for the Default cache
var cache = new CacheType { Default = new DefaultCache{EntryCapacity = 100} };

// Create a CMC using the default cache
var cryptographicMaterialsCacheInput = new
 CreateCryptographicMaterialsCacheInput {Cache = cache};

var sharedCryptographicMaterialsCache =
 materialProviders.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

### Python

```
# Instantiate the MPL
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create a CacheType object for the default cache
cache: CacheType = CacheTypeDefault(
    value=DefaultCache(
        entry_capacity=100,
    )
)

# Create a CMC using the default cache
cryptographic_materials_cache_input = CreateCryptographicMaterialsCacheInput(
    cache=cache,
)

shared_cryptographic_materials_cache =
 mat_prov.create_cryptographic_materials_cache(
    cryptographic_materials_cache_input
)
```

### Rust

```
// Instantiate the MPL
```

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create a CacheType object for the default cache
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);

// Create a CMC using the default cache
let shared_cryptographic_materials_cache: CryptographicMaterialsCacheRef = mpl.
    create_cryptographic_materials_cache()
    .cache(cache)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
)

// Instantiate the MPL
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create a CacheType object for the default cache
cache := mpltypes.CacheTypeMemberDefault{
    Value: mpltypes.DefaultCache{
        EntryCapacity: 100,
    },
}

// Create a CMC using the default cache
cmcCacheInput := mpltypes.CreateCryptographicMaterialsCacheInput{
```

```
    Cache: &cache,
}
sharedCryptographicMaterialsCache, err :=
 matProv.CreateCryptographicMaterialsCache(context.Background(), cmcCacheInput)
if err != nil {
    panic(err)
}
```

2. 공유 캐시에 대한 CacheType 객체를 생성합니다.

   1단계에서 sharedCryptographicMaterialsCache 생성한를 새 CacheType 객체에 전달합니다.

   Java

   ```
   // Create a CacheType object for the sharedCryptographicMaterialsCache
   final CacheType sharedCache =
       CacheType.builder()
           .Shared(sharedCryptographicMaterialsCache)
           .build();
   ```

   C# / .NET

   ```
   // Create a CacheType object for the sharedCryptographicMaterialsCache
   var sharedCache = new CacheType { Shared = sharedCryptographicMaterialsCache };
   ```

   Python

   ```
   # Create a CacheType object for the shared_cryptographic_materials_cache
   shared_cache: CacheType = CacheTypeShared(
       value=shared_cryptographic_materials_cache
   )
   ```

   Rust

   ```
   // Create a CacheType object for the shared_cryptographic_materials_cache
   let shared_cache: CacheType =
    CacheType::Shared(shared_cryptographic_materials_cache);
   ```

Go

```
// Create a CacheType object for the shared_cryptographic_materials_cache
shared_cache :=
 mpltypes.CacheTypeMemberShared{sharedCryptographicMaterialsCache}
```

3. 2단계에서 계층적 키링으로 sharedCache 객체를 전달합니다.

공유 캐시를 사용하여 계층적 키링을 생성할 때 선택적으로를 정의partitionID하여 여러 계층적 키링에서 캐시 항목을 공유할 수 있습니다. 파티션 ID를 지정하지 않으면 계층적 키링이 자동으로 키링에 고유한 파티션 ID를 할당합니다.

> **Note**
>
> 동일한 파티션 ID, logical key store name및 브랜치 키 ID를 참조하는 두 개 이상의 키링을 생성하는 경우 계층적 키링은 공유 캐시에서 동일한 캐시 항목을 공유합니다. 여러 키링이 동일한 캐시 항목을 공유하지 않도록 하려면 각 계층적 키링에 고유한 파티션 ID를 사용해야 합니다.

다음 예제에서는 branch key ID supplier, 캐시 제한 600초로 계층적 키링을 생성합니다. 다음 계층적 키링 구성에 정의된 값에 대한 자세한 내용은 섹션을 참조하세요the section called “계층적 키링 생성”.

Java

```
// Create the Hierarchical keyring
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
    CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(sharedCache)
        .partitionID(partitionID)
        .build();
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
// Create the Hierarchical keyring
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   Cache = sharedCache,
   TtlSeconds = 600,
   PartitionId = partitionID
};
var keyring =
 materialProviders.CreateAwsKmsHierarchicalKeyring(createKeyringInput);
```

Python

```
# Create the Hierarchical keyring
keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id_supplier=branch_key_id_supplier,
    ttl_seconds=600,
    cache=shared_cache,
    partition_id=partition_id
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
    // CryptographicMaterialsCacheRef is an Rc (Reference Counted), so if you
 clone it to
    // pass it to different Hierarchical Keyrings, it will still point to the
 same
    // underlying cache, and increment the reference count accordingly.
```

```
    .cache(shared_cache.clone())
    .ttl_seconds(600)
    .partition_id(partition_id.clone())
    .send()
    .await?;
```

Go

```
// Create the Hierarchical keyring
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:    keyStore1,
    BranchKeyId: &branchKeyId,
    TtlSeconds:  600,
    Cache:       &shared_cache,
    PartitionId: &partitionId,
}
keyring, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

## 계층적 키링 생성

계층적 키링을 생성하려면 다음 값을 제공해야 합니다.

- 키 스토어 이름

  사용자 또는 키 스토어 관리자가 키 스토어 역할을 하도록 생성한 DynamoDB 테이블의 이름입니다.

- 

  캐시 제한 유지 시간(TTL)

  로컬 캐시 내의 브랜치 키 자료 항목이 만료되기 전에 사용할 수 있는 시간(초) 입니다. 캐시 제한 TTL은 클라이언트가를 호출 AWS KMS 하여 브랜치 키 사용을 승인하는 빈도를 지정합니다. 이 값은 0보다 커야 합니다. 캐시 제한 TTL이 만료된 후에는 항목이 제공되지 않으며 로컬 캐시에서 제거됩니다.

- 브랜치 키 식별자

키 스토어에서 단일 활성 브랜치 키를 `branch-key-id` 식별하는를 정적으로 구성하거나 브랜치 키 ID 공급자를 제공할 수 있습니다.

브랜치 키 ID 공급자는 암호화 컨텍스트에 저장된 필드를 사용하여 레코드를 복호화하는 데 필요한 브랜치 키를 결정합니다.

각 테넌트에 자체 브랜치 키가 있는 멀티테넌트 데이터베이스에는 브랜치 키 ID 공급자를 사용하는 것이 좋습니다. 브랜치 키 ID 공급자를 사용하여 브랜치 키 IDs에 대한 기억하기 쉬운 이름을 생성하여 특정 테넌트에 대한 올바른 브랜치 키 ID를 쉽게 인식할 수 있습니다. 예를 들어 친숙한 이름을 사용하면 브랜치 키를 `b3f61619-4d35-48ad-a275-050f87e15122` 대신 `tenant1`로 참조할 수 있습니다.

복호화 작업의 경우 단일 계층적 키링을 정적으로 구성하여 복호화을 단일 테넌트로 제한하거나 브랜치 키 ID 공급자를 사용하여 레코드 복호화을 담당하는 테넌트를 식별할 수 있습니다.

- (선택 사항) 캐시

  캐시 유형이나 로컬 캐시에 저장할 수 있는 브랜치 키 자료 항목 수를 사용자 지정하려면 키링을 초기화할 때 캐시 유형과 항목 용량을 지정하세요.

  계층적 키링은 기본, MultiThreaded, StormTracking 및 공유 캐시 유형을 지원합니다. 각 캐시 유형을 정의하는 방법을 보여주는 자세한 내용과 예제는 섹션을 참조하세요the section called “캐시 선택”.

  캐시를 지정하지 않으면 계층적 키링은 자동으로 기본 캐시 유형을 사용하고 항목 용량을 1,000으로 설정합니다.

- (선택 사항) 파티션 ID

  를 지정하는 경우 선택적으로 파티션 ID를 정의할 the section called “공유 캐시”수 있습니다. 파티션 ID는 캐시에 쓰는 계층적 키링을 구분합니다. 파티션에서 캐시 항목을 재사용하거나 공유하려는 경우 고유한 파티션 ID를 정의해야 합니다. 파티션 ID에 대한 문자열을 지정할 수 있습니다. 파티션 ID를 지정하지 않으면 생성 시 고유한 파티션 ID가 키링에 자동으로 할당됩니다.

  자세한 내용은 Partitions 단원을 참조하십시오.

> **Note**
>
> 동일한 파티션 ID, logical key store name및 브랜치 키 ID를 참조하는 두 개 이상의 키링을 생성하는 경우 계층적 키링은 공유 캐시에서 동일한 캐시 항목을 공유합니다. 여러 키링이 동일한 캐시 항목을 공유하지 않도록 하려면 각 계층적 키링에 고유한 파티션 ID를 사용해야 합니다.

- (선택 사항) 권한 부여 토큰 목록

  권한 부여를 통해 계층적 키링의 KMS 키에 대한 액세스를 제어하는 경우 키링을 초기화할 때 필요한 모든 권한 부여 토큰을 제공해야 합니다.

정적 브랜치 키 ID를 사용하여 계층적 키링 생성

다음 예제에서는 정적 브랜치 키 ID, the section called “기본 캐시”및 캐시 제한 TTL이 600초인 계층적 키링을 생성하는 방법을 보여줍니다.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyId(branch-key-id)
        .ttlSeconds(600)
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyId = branch-key-id,
   TtlSeconds = 600
};
```

```
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

### Python

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id=branch_key_id,
    ttl_seconds=600
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

### Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
        .create_aws_kms_hierarchical_keyring()
        .key_store(key_store.clone())
        .branch_key_id(branch_key_id)
        .ttl_seconds(600)
        .send()
        .await?;
```

### Go

```
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:    keyStore,
    BranchKeyId: &branchKeyID,
    TtlSeconds:  600,
```

```
}
hKeyRing, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

### 브랜치 키 ID 공급자를 사용하여 계층적 키링 생성

다음 절차에서는 브랜치 키 ID 공급자를 사용하여 계층적 키링을 생성하는 방법을 보여줍니다.

1. 브랜치 키 ID 공급자 생성

   다음 예제에서는 두 개의 브랜치 키에 대한 표시 이름을 생성하고를 호출`CreateDynamoDbEncryptionBranchKeyIdSupplier`하여 브랜치 키 ID 공급자를 생성합니다.

   Java

```
// Create friendly names for each branch-key-id
class ExampleBranchKeyIdSupplier implements IDynamoDbKeyBranchKeyIdSupplier {
    private static String branchKeyIdForTenant1;
    private static String branchKeyIdForTenant2;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this.branchKeyIdForTenant1 = tenant1Id;
        this.branchKeyIdForTenant2 = tenant2Id;
    }
// Create the branch key ID supplier
final DynamoDbEncryption ddbEnc = DynamoDbEncryption.builder()
        .DynamoDbEncryptionConfig(DynamoDbEncryptionConfig.builder().build())
        .build();
final BranchKeyIdSupplier branchKeyIdSupplier =
 ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    CreateDynamoDbEncryptionBranchKeyIdSupplierInput.builder()
            .ddbKeyBranchKeyIdSupplier(new ExampleBranchKeyIdSupplier(branch-
key-ID-tenant1, branch-key-ID-tenant2))
            .build()).branchKeyIdSupplier();
```

### C# / .NET

```
// Create friendly names for each branch-key-id
 class ExampleBranchKeyIdSupplier : DynamoDbKeyBranchKeyIdSupplierBase {
    private String _branchKeyIdForTenant1;
    private String _branchKeyIdForTenant2;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this._branchKeyIdForTenant1 = tenant1Id;
        this._branchKeyIdForTenant2 = tenant2Id;
    }
// Create the branch key ID supplier
var ddbEnc = new DynamoDbEncryption(new DynamoDbEncryptionConfig());
var branchKeyIdSupplier = ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    new CreateDynamoDbEncryptionBranchKeyIdSupplierInput
    {
        DdbKeyBranchKeyIdSupplier = new ExampleBranchKeyIdSupplier(branch-key-
ID-tenant1, branch-key-ID-tenant2)
    }).BranchKeyIdSupplier;
```

### Python

```
# Create branch key ID supplier that maps the branch key ID to a friendly name
branch_key_id_supplier: IBranchKeyIdSupplier = ExampleBranchKeyIdSupplier(
        tenant_1_id=branch_key_id_a,
        tenant_2_id=branch_key_id_b,
)
```

### Rust

```
// Create branch key ID supplier that maps the branch key ID to a friendly name
let branch_key_id_supplier = ExampleBranchKeyIdSupplier::new(
    &branch_key_id_a,
    &branch_key_id_b
);
```

### Go

```
// Create branch key ID supplier that maps the branch key ID to a friendly name
keySupplier := branchKeySupplier{branchKeyA: branchKeyA, branchKeyB: branchKeyB}
```

2. 계층적 키링 생성

다음 예제에서는 1단계에서 생성된 브랜치 키 ID 공급자, 600초의 캐시 제한 TLL, 1000의 최대 캐시 크기로 계층적 키링을 초기화합니다.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                .entryCapacity(100)
                .build())
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
   {
        Default = new DefaultCache { EntryCapacity = 100 }
   }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Python

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
```

```
)

keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id_supplier=branch_key_id_supplier,
    ttl_seconds=600,
    cache=CacheTypeDefault(
        value=DefaultCache(
            entry_capacity=100
        )
    ),
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store.clone())
    .branch_key_id_supplier(branch_key_id_supplier)
    .ttl_seconds(600)
    .send()
    .await?;
```

Go

```
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:            keyStore,
    BranchKeyIdSupplier: &keySupplier,
    TtlSeconds:          600,
}
hKeyRing, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
```

```
}
```

# AWS KMS ECDH 키링

AWS KMS ECDH 키링은 비대칭 키 계약을 사용하여 두 당사자 간에 공유 대칭 래핑 키를 도출AWS KMS keys합니다. 먼저 키링은 Elliptic Curve Diffie-Hellman(ECDH) 키 계약 알고리즘을 사용하여 발신자의 KMS 키 페어와 수신자의 퍼블릭 키에 있는 프라이빗 키에서 공유 암호를 도출합니다. 그런 다음 키링은 공유 보안 암호를 사용하여 데이터 암호화 키를 보호하는 공유 래핑 키를 파생합니다. 가(KDF_CTR_HMAC_SHA384)를 AWS Encryption SDK 사용하여 공유 래핑 키를 파생하는 키 파생 함수는 키 파생에 대한 NIST 권장 사항을 준수합니다.

키 파생 함수는 64바이트의 키 지정 구성 요소를 반환합니다. 양 당사자가 올바른 키 구성 요소를 사용하도록 하기 위해는 처음 32바이트를 커밋 키로 사용하고 마지막 32바이트를 공유 래핑 키로 AWS Encryption SDK 사용합니다. 복호화 시 키링이 메시지 헤더 사이퍼텍스트에 저장된 동일한 커밋 키와 공유 래핑 키를 재현할 수 없는 경우 작업이 실패합니다. 예를 들어 Alice의 프라이빗 키와 Bob의 퍼블릭 키로 구성된 키링으로 데이터를 암호화하는 경우 Bob의 프라이빗 키와 Alice의 퍼블릭 키로 구성된 키링은 동일한 커밋 키와 공유 래핑 키를 재현하고 데이터를 해독할 수 있습니다. Bob의 퍼블릭 키가 KMS 키 페어가 아닌 경우 Bob은 원시 ECDH 키링을 생성하여 데이터를 해독할 수 있습니다.

AWS KMS ECDH 키링은 AES-GCM을 사용하여 대칭 키로 데이터를 암호화합니다. 그런 다음 AES-GCM을 사용하여 파생된 공유 래핑 키로 데이터 키를 봉투 암호화합니다. 각 AWS KMS ECDH 키링에는 공유 래핑 키가 하나만 있을 수 있지만, 다중 키링에는 단독으로 또는 다른 키링과 함께 여러 AWS KMS ECDH 키링을 포함할 수 있습니다.

프로그래밍 언어 호환성

AWS KMS ECDH 키링은 암호화 자료 공급자 라이브러리(MPL) 버전 1.5.0에 도입되었으며 다음 프로그래밍 언어 및 버전에서 지원됩니다.

- 의 버전 3.x AWS Encryption SDK for Java
- for .NET 버전 AWS Encryption SDK 4.x
- 선택적 MPL 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
- AWS Encryption SDK for Rust 버전 1.x
- Go AWS Encryption SDK 용의 버전 0.1.x 이상

주제

## AWS KMS ECDH 키링에 필요한 권한

AWS Encryption SDK 에는 AWS 계정이 필요하지 않으며 서비스에 AWS 의존하지 않습니다. 그러나 AWS KMS ECDH 키링을 사용하려면 키링 AWS KMS keys 의에 대한 AWS 계정과 다음과 같은 최소 권한이 필요합니다. 권한은 사용하는 키 계약 스키마에 따라 달라집니다.

- `KmsPrivateKeyToStaticPublicKey` 키 계약 스키마를 사용하여 데이터를 암호화하고 해독하려면 발신자의 비대칭 KMS 키 페어에 kms:GetPublicKey 및 kms:DeriveSharedSecret이 필요합니다. 키링을 인스턴스화할 때 발신자의 DER 인코딩 퍼블릭 키를 직접 제공하는 경우 발신자의 비대칭 KMS 키 페어에 대한 kms:DeriveSharedSecret 권한만 있으면 됩니다.
- `KmsPublicKeyDiscovery` 키 계약 스키마를 사용하여 데이터를 복호화하려면 지정된 비대칭 KMS 키 페어에 대한 kms:DeriveSharedSecret 및 kms:GetPublicKey 권한이 필요합니다.

## AWS KMS ECDH 키링 생성

데이터를 암호화하고 해독하는 AWS KMS ECDH 키링을 생성하려면 `KmsPrivateKeyToStaticPublicKey` 키 계약 스키마를 사용해야 합니다. `KmsPrivateKeyToStaticPublicKey` 키 계약 스키마를 사용하여 AWS KMS ECDH 키링을 초기화하려면 다음 값을 제공합니다.

- 발신자 AWS KMS key ID

  `KeyUsage` 값이 인 비대칭 NIST 권장 타원 곡선(ECC) KMS 키 페어를 식별해야 합니다`KEY_AGREEMENT`. 발신자의 프라이빗 키는 공유 암호를 도출하는 데 사용됩니다.

- (선택 사항) 발신자의 퍼블릭 키

  RFC 5280에 정의된 대로 `SubjectPublicKeyInfo` (SPKI)라고도 하는 DER 인코딩 X.509 퍼블릭 키여야 합니다. https://tools.ietf.org/html/rfc5280

  AWS KMS GetPublicKey 작업은 비대칭 KMS 키 페어의 퍼블릭 키를 필요한 DER 인코딩 형식으로 반환합니다.

키링이 수행하는 AWS KMS 호출 수를 줄이기 위해 발신자의 퍼블릭 키를 직접 제공할 수 있습니다. 발신자의 퍼블릭 키에 값이 제공되지 않은 경우 키링은를 호출 AWS KMS 하여 발신자의 퍼블릭 키를 검색합니다.

- 수신자의 퍼블릭 키

  RFC 5280에 정의된 대로 `SubjectPublicKeyInfo` (SPKI)라고도 하는 수신자의 DER 인코딩 X.509 퍼블릭 키를 제공해야 합니다. https://tools.ietf.org/html/rfc5280

  AWS KMS GetPublicKey 작업은 비대칭 KMS 키 페어의 퍼블릭 키를 필요한 DER 인코딩 형식으로 반환합니다.

- 곡선 사양

  지정된 키 페어에서 타원 곡선 사양을 식별합니다. 발신자와 수신자의 키 페어 모두 곡선 사양이 동일해야 합니다.

  유효한 값: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

- (선택 사항) 권한 부여 토큰 목록

  권한 부여를 사용하여 AWS KMS ECDH 키링의 KMS 키에 대한 액세스를 제어하는 경우 키링을 초기화할 때 필요한 모든 권한 부여 토큰을 제공해야 합니다.

C# / .NET

다음 예제에서는 발신자의 KMS 키, 발신자의 퍼블릭 키 및 수신자의 퍼블릭 키를 사용하여 AWS KMS 로 ECDH 키링을 생성합니다. 이 예제에서는 선택적 `SenderPublicKey` 파라미터를 사용하여 발신자의 퍼블릭 키를 제공합니다. 발신자의 퍼블릭 키를 제공하지 않으면 키링이를 호출 AWS KMS 하여 발신자의 퍼블릭 키를 검색합니다. 발신자와 수신자의 키 페어가 모두 `ECC_NIST_P256` 곡선에 있습니다.

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Must be DER-encoded X.509 public keys
var BobPublicKey = new MemoryStream(new byte[] { });
var AlicePublicKey = new MemoryStream(new byte[] { });

// Create the AWS KMS ECDH static keyring
var staticConfiguration = new KmsEcdhStaticConfigurations
{
```

```
    KmsPrivateKeyToStaticPublicKey = new KmsPrivateKeyToStaticPublicKeyInput
    {
        SenderKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        SenderPublicKey = BobPublicKey,
        RecipientPublicKey = AlicePublicKey
    }
};

var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = staticConfiguration
};

var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

다음 예제에서는 발신자의 KMS 키, 발신자의 퍼블릭 키 및 수신자의 퍼블릭 키를 사용하여 AWS KMS 로 ECDH 키링을 생성합니다. 이 예제에서는 선택적 `senderPublicKey` 파라미터를 사용하여 발신자의 퍼블릭 키를 제공합니다. 발신자의 퍼블릭 키를 제공하지 않으면 키링이를 호출 AWS KMS 하여 발신자의 퍼블릭 키를 검색합니다. 발신자와 수신자의 키 페어가 모두 `ECC_NIST_P256` 곡선에 있습니다.

```
// Retrieve public keys
// Must be DER-encoded X.509 public keys
ByteBuffer BobPublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab");
        ByteBuffer AlicePublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");

// Create the AWS KMS ECDH static keyring
        final CreateAwsKmsEcdhKeyringInput senderKeyringInput =
          CreateAwsKmsEcdhKeyringInput.builder()
            .kmsClient(KmsClient.create())
            .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
            .KeyAgreementScheme(
              KmsEcdhStaticConfigurations.builder()
                .KmsPrivateKeyToStaticPublicKey(
                  KmsPrivateKeyToStaticPublicKeyInput.builder()
```

```
                    .senderKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
                    .senderPublicKey(BobPublicKey)
                    .recipientPublicKey(AlicePublicKey)
                    .build()).build()).build();
```

Python

다음 예제에서는 발신자의 KMS 키, 발신자의 퍼블릭 키 및 수신자의 퍼블릭 키를 사용하여 AWS KMS 로 ECDH 키링을 생성합니다. 이 예제에서는 선택적 `senderPublicKey` 파라미터를 사용하여 발신자의 퍼블릭 키를 제공합니다. 발신자의 퍼블릭 키를 제공하지 않으면 키링이를 호출 AWS KMS 하여 발신자의 퍼블릭 키를 검색합니다. 발신자와 수신자의 키 페어가 모두 `ECC_NIST_P256` 곡선에 있습니다.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
    KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey,
    KmsPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Retrieve public keys
# Must be DER-encoded X.509 public keys
bob_public_key = get_public_key_bytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
alice_public_key = get_public_key_bytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321")

# Create the AWS KMS ECDH static keyring
sender_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey(
        KmsPrivateKeyToStaticPublicKeyInput(
```

```
            sender_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            sender_public_key = bob_public_key,
            recipient_public_key = alice_public_key,

        )
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(sender_keyring_input)
```

Rust

다음 예제에서는 발신자의 KMS 키, 발신자의 퍼블릭 키 및 수신자의 퍼블릭 키를 사용하여 AWS KMS 로 ECDH 키링을 생성합니다. 이 예제에서는 선택적 sender_public_key 파라미터를 사용하여 발신자의 퍼블릭 키를 제공합니다. 발신자의 퍼블릭 키를 제공하지 않으면 키링이를 호출 AWS KMS 하여 발신자의 퍼블릭 키를 검색합니다.

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create the AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Retrieve public keys
// Must be DER-encoded X.509 keys
let public_key_file_content_sender =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_SENDER))?;
let parsed_public_key_file_content_sender = parse(public_key_file_content_sender)?;
let public_key_sender_utf8_bytes = parsed_public_key_file_content_sender.contents();
```

```
let public_key_file_content_recipient =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content_recipient =
 parse(public_key_file_content_recipient)?;
let public_key_recipient_utf8_bytes =
 parsed_public_key_file_content_recipient.contents();

// Create KmsPrivateKeyToStaticPublicKeyInput
let kms_ecdh_static_configuration_input =
    KmsPrivateKeyToStaticPublicKeyInput::builder()
        .sender_kms_identifier(arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab)
        // Must be a UTF8 DER-encoded X.509 public key
        .sender_public_key(public_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let kms_ecdh_static_configuration =
 KmsEcdhStaticConfigurations::KmsPrivateKeyToStaticPublicKey(kms_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH keyring
let kms_ecdh_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client)
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
```

```
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[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",
}

// Retrieve public keys
// Must be DER-encoded X.509 keys
publicKeySender, err := utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameSender)
if err != nil {
    panic(err)
}
publicKeyRecipient, err :=
 utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameRecipient)
if err != nil {
    panic(err)
```

```
}

// Create KmsPrivateKeyToStaticPublicKeyInput
kmsEcdhStaticConfigurationInput := mpltypes.KmsPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey:  publicKeyRecipient,
    SenderKmsIdentifier: arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    SenderPublicKey:     publicKeySender,
}
kmsEcdhStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPrivateKeyToStaticPublicKey{
    Value: kmsEcdhStaticConfigurationInput,
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create AWS KMS ECDH keyring
awsKmsEcdhKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhStaticConfiguration,
    KmsClient:          kmsClient,
}
awsKmsEcdhKeyring, err := matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

## AWS KMS ECDH 검색 키링 생성

복호화할 때에서 사용할 AWS Encryption SDK 수 있는 키를 지정하는 것이 가장 좋습니다. 이 모범 사례를 따르려면 `KmsPrivateKeyToStaticPublicKey` 키 계약 스키마와 함께 AWS KMS ECDH 키링을 사용합니다. 그러나 AWS KMS ECDH 검색 키링, 즉 지정된 KMS 키 페어의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있는 AWS KMS ECDH 키링을 생성할 수도 있습니다.

> **Important**
>
> `KmsPublicKeyDiscovery` 키 계약 스키마를 사용하여 메시지를 복호화할 때 누가 소유하든 모든 퍼블릭 키를 수락합니다.

`KmsPublicKeyDiscovery` 키 계약 스키마를 사용하여 AWS KMS ECDH 키링을 초기화하려면 다음 값을 제공합니다.

- 수신자 ID AWS KMS key

  `KeyUsage` 값이 인 비대칭 NIST 권장 타원 곡선(ECC) KMS 키 페어를 식별해야 합니다`KEY_AGREEMENT`.
- 곡선 사양

  수신자의 KMS 키 페어에서 타원 곡선 사양을 식별합니다.

  유효한 값: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`
- (선택 사항) 권한 부여 토큰 목록

  권한 부여를 사용하여 AWS KMS ECDH 키링의 KMS 키에 대한 액세스를 제어하는 경우 키링을 초기화할 때 필요한 모든 권한 부여 토큰을 제공해야 합니다.

C# / .NET

다음 예제에서는 `ECC_NIST_P256` 곡선에 KMS 키 페어가 있는 AWS KMS ECDH 검색 키링을 생성합니다. 지정된 KMS 키 페어에 대해 kms:GetPublicKey 및 kms:DeriveSharedSecret 권한이 있어야 합니다. 이 키링은 지정된 KMS 키 페어의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있습니다.

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create the AWS KMS ECDH discovery keyring
var discoveryConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPublicKeyDiscovery = new KmsPublicKeyDiscoveryInput
    {
        RecipientKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"
```

```
    }

};
var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = discoveryConfiguration
};
var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

다음 예제에서는 ECC_NIST_P256 곡선에 KMS 키 페어가 있는 AWS KMS ECDH 검색 키링을 생성합니다. 지정된 KMS 키 페어에 대해 kms:GetPublicKey 및 kms:DeriveSharedSecret 권한이 있어야 합니다. 이 키링은 지정된 KMS 키 페어의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있습니다.

```
// Create the AWS KMS ECDH discovery keyring
final CreateAwsKmsEcdhKeyringInput recipientKeyringInput =
  CreateAwsKmsEcdhKeyringInput.builder()
    .kmsClient(KmsClient.create())
    .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
    .KeyAgreementScheme(
      KmsEcdhStaticConfigurations.builder()
        .KmsPublicKeyDiscovery(
          KmsPublicKeyDiscoveryInput.builder()
            .recipientKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321").build()
        ).build())
    .build();
```

Python

다음 예제에서는 ECC_NIST_P256 곡선에 KMS 키 페어가 있는 AWS KMS ECDH 검색 키링을 생성합니다. 지정된 KMS 키 페어에 대해 kms:GetPublicKey 및 kms:DeriveSharedSecret 권한이 있어야 합니다. 이 키링은 지정된 KMS 키 페어의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있습니다.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
```

```
    KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery,
    KmsPublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS ECDH discovery keyring
create_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery(
        KmsPublicKeyDiscoveryInput(
            recipient_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321",
        )
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(create_keyring_input)
```

### Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create the AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
```

```
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Create KmsPublicKeyDiscoveryInput
let kms_ecdh_discovery_static_configuration_input =
    KmsPublicKeyDiscoveryInput::builder()
        .recipient_kms_identifier(ecc_recipient_key_arn)
        .build()?;

let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH discovery keyring
let kms_ecdh_discovery_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client.clone())
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_discovery_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)
```

```
// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[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",
}

// Create KmsPublicKeyDiscoveryInput
kmsEcdhDiscoveryStaticConfigurationInput := mpltypes.KmsPublicKeyDiscoveryInput{
    RecipientKmsIdentifier: eccRecipientKeyArn,
}
kmsEcdhDiscoveryStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPublicKeyDiscovery{
    Value: kmsEcdhDiscoveryStaticConfigurationInput,
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create AWS KMS ECDH discovery keyring
awsKmsEcdhDiscoveryKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhDiscoveryStaticConfiguration,
    KmsClient:          kmsClient,
```

```
}
awsKmsEcdhDiscoveryKeyring, err :=
 matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhDiscoveryKeyringInput)
if err != nil {
    panic(err)
}
```

# Raw AES 키링

를 AWS Encryption SDK 사용하면 데이터 키를 보호하는 래핑 키로 제공하는 AES 대칭 키를 사용할 수 있습니다. 가급적이면 하드웨어 보안 모듈(HSM) 또는 키 관리 시스템에서 키 자료를 생성, 저장 및 보호해야 합니다. 래핑 키를 제공하고 로컬 또는 오프라인에서 데이터 키를 암호화해야 하는 경우 Raw AES 키링을 사용하세요.

Raw AES 키링은 데이터를 암호화하기 위해 바이트 배열로 지정하는 AES-GCM 알고리즘 및 래핑 키를 사용합니다. 각 Raw AES 키링에는 래핑 키를 하나만 지정할 수 있지만, 여러 개의 Raw AES 키링을 단독으로 또는 다른 키링과 함께 다중 키링에 포함할 수 있습니다.

원시 AES 키링은의 JceMasterKey 클래스 및 AES 암호화 키와 함께 사용되는 AWS Encryption SDK for Python 경우의 RawMasterKey 클래스와 동일 AWS Encryption SDK for Java 하며 상호 작용합니다. 한 구현으로 데이터를 암호화하고 다른 구현으로는 동일한 래핑 키를 사용하여 데이터를 복호화할 수 있습니다. 자세한 내용은 키링 호환성 섹션을 참조하세요.

키 네임스페이스 및 이름

키링에서 AES 키를 식별하기 위해 Raw AES 키링은 사용자가 제공한 키 네임스페이스와 키 이름을 사용합니다. 이 값은 비밀이 아닙니다. 암호화 작업이 반환하는 암호화된 메시지 헤더에 일반 텍스트로 나타납니다. HSM 또는 키 관리 시스템의 키 네임스페이스와 해당 시스템에서 AES 키를 식별하는 키 이름을 사용하는 것이 좋습니다.

> **Note**
>
> 키 네임스페이스와 키 이름은 `JceMasterKey` 및 `RawMasterKey`의 공급자 ID(또는 공급자) 및 키 ID 필드와 동일합니다.
>
> .NET AWS Encryption SDK 용 AWS Encryption SDK for C 및는 KMS `aws-kms` 키의 키 네임스페이스 값을 예약합니다. 이 라이브러리의 Raw AES 키링 또는 Raw RSA 키링에는 해당 네임스페이스 값을 사용하지 마세요.

특정 메시지를 암호화하고 복호화하기 위해 서로 다른 키링을 구성하는 경우 네임스페이스와 이름 값이 중요합니다. 복호화 키링의 키 네임스페이스와 키 이름이 대/소문자를 구분하여 암호화 키링의 키 네임스페이스와 키 이름이 정확히 일치하지 않으면 키 자료 바이트가 동일하더라도 복호화 키링이 사용되지 않습니다.

예를 들어 키 네임스페이스 `HSM_01`과 키 이름 `AES_256_012`를 사용하여 Raw AES 키링을 정의할 수 있습니다. 그런 다음 해당 키링을 사용하여 일부 데이터를 암호화합니다. 해당 데이터를 복호화하려면 동일한 키 네임스페이스, 키 이름 및 키 자료를 사용하여 Raw AES 키링을 구성하세요.

다음 예제에서는 Raw AES 키링을 생성하는 방법을 보여줍니다. `AESWrappingKey` 변수는 사용자가 제공하는 키 자료를 나타냅니다.

C

에서 원시 AES 키링을 인스턴스화하려면를 AWS Encryption SDK for C사용합니다`aws_cryptosdk_raw_aes_keyring_new()`. 전체 예제를 보려면 raw_aes_keyring.c를 참조하세요.

```
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

AWS Encryption SDK for .NET에서 원시 AES 키링을 생성하려면 `materialProviders.CreateRawAesKeyring()` 메서드를 사용합니다. 전체 예제를 보려면 RawAESKeyringExample.cs를 참조하세요.

다음 예제에서는 AWS Encryption SDK for .NET 4.x 버전을 사용합니다.

```
// 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.ALG_AES256_GCM_IV12_TAG16
};

var keyring = materialProviders.CreateRawAesKeyring(createKeyringInput);
```

JavaScript Browser

브라우저 AWS Encryption SDK for JavaScript 의는 WebCrypto API에서 암호화 프리미티브를 가져옵니다. 키링을 구성하기 전에 WebCrypto 백엔드로 원시 키 자료를 가져오는 데 `RawAesKeyringWebCrypto.importCryptoKey()`를 사용해야 합니다. 이렇게 하면 WebCrypto에 대한 모든 호출이 비동기식이어도 키링이 완료됩니다.

다음으로 Raw AES 키링을 인스턴스화하려면 `RawAesKeyringWebCrypto()` 메서드를 사용하세요. 키 자료의 길이에 따라 AES 래핑 알고리즘(“래핑 제품군)을 지정해야 합니다. 전체 예제를 보려면 aes_simple.ts(JavaScript 브라우저)를 참조하세요.

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한`buildClient`할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

```
import {
  RawAesWrappingSuiteIdentifier,
  RawAesKeyringWebCrypto,
  synchronousRandomValues,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'
```

```
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

Node.js AWS Encryption SDK for JavaScript 용에서 원시 AES 키링을 인스턴스화하려면 RawAesKeyringNode 클래스의 인스턴스를 생성합니다. 키 자료의 길이에 따라 AES 래핑 알고리즘("래핑 제품군")을 지정해야 합니다. 전체 예제를 보려면 aes_simple.ts(JavaScript Node.js)를 참조하세요.

다음 예제에서는 buildClient 함수를 사용하여 기본 커밋 정책인를 지정합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한buildClient할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

```
import {
  RawAesKeyringNode,
  buildClient,
  CommitmentPolicy,
  RawAesWrappingSuiteIdentifier,
} from '@aws-crypto/client-node'
```

```
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

const wrappingSuite =
  RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

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

Java

에서 원시 AES 키링을 인스턴스화하려면를 AWS Encryption SDK for Java사용합니다matProv.CreateRawAesKeyring().

```
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);
```

Python

다음 예제에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 전체 예제는 GitHub의 AWS Encryption SDK for Python 리포지토리에서 raw_aes_keyring_example.py를 참조하세요.

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
```

```
)

# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "AES_256_012"

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "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 material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create Raw AES keyring
keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)

raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=keyring_input
)
```

### Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "AES_256_012";

// Optional: Create an encryption context
```

```
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create Raw AES keyring
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

Go

```
import (
    mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
    client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
    esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)
//Instantiate the AWS Encryption SDK client.
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
// Define the key namespace and key name
var keyNamespace = "A managed aes keys"
```

```
var keyName = "My 256-bit AES wrapping key"

// Optional: Create an encryption context
encryptionContext := map[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 material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
// Create Raw AES keyring
aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  aesWrappingKey,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
if err != nil {
    panic(err)
}
```

# Raw RSA 키링

Raw RSA 키링은 제공한 RSA 퍼블릭 및 프라이빗 키를 사용하여 로컬 메모리에서 데이터 키의 비대칭 암호화 및 복호화를 수행합니다. 가급적이면 하드웨어 보안 모듈(HSM) 또는 키 관리 시스템에서 프라이빗 키를 생성, 저장 및 보호해야 합니다. 암호화 기능은 RSA 퍼블릭 키로 데이터 키를 암호화합니다. 복호화 함수는 프라이빗 키를 사용하여 데이터 키를 복호화합니다. 여러 RSA 패딩 모드 중에서 선택할 수 있습니다.

암호화 및 복호화하는 Raw RSA 키링에는 비대칭 퍼블릭 키 페어와 프라이빗 키 페어가 포함되어야 합니다. 단, 퍼블릭 키만 있는 Raw RSA 키링을 사용하여 데이터를 암호화할 수 있으며, 프라이빗 키만 있는 Raw RSA 키링을 사용하여 데이터를 복호화할 수 있습니다. 다중 키링에 Raw RSA 키링을 포함시킬 수 있습니다. 퍼블릭 키와 프라이빗 키로 Raw RSA 키링을 구성하는 경우 두 키링이 동일한 키 페

어에 속하는지 확인하세요. 의 일부 언어 구현 AWS Encryption SDK 은 서로 다른 페어의 키를 사용하여 Raw RSA 키링을 구성하지 않습니다. 다른 구현에서는 키가 동일한 키 페어에서 나온 것인지 사용자가 확인해야 합니다.

Raw RSA 키링은 RSA 비대칭 암호화 키와 함께 사용되는 AWS Encryption SDK for Python 경우의 JceMasterKey AWS Encryption SDK for Java 및의 RawMasterKey와 동일하고 상호 작동합니다. 한 구현으로 데이터를 암호화하고 다른 구현으로는 동일한 래핑 키를 사용하여 데이터를 복호화할 수 있습니다. 자세한 내용은 키링 호환성을 참조하세요.

> **Note**
>
> Raw RSA 키링은 비대칭 KMS 키를 지원하지 않습니다. 비대칭 RSA KMS 키를 사용하려면 다음 프로그래밍 언어가 비대칭 RSA를 사용하는 AWS KMS 키링을 지원합니다 AWS KMS keys.
>
> - 의 버전 3.x AWS Encryption SDK for Java
> - for .NET 버전AWS Encryption SDK 4.x
> - 선택적 암호화 자료 공급자 라이브러리(MPL) 종속성과 함께 사용하는 AWS Encryption SDK for Python경우 버전 4.x.
> - Go AWS Encryption SDK 용의 버전 0.1.x 이상
>
> RSA KMS 키의 퍼블릭 키가 포함된 Raw RSA 키링으로 데이터를 암호화하는 경우 AWS Encryption SDK 도 복호화할 AWS KMS 수 없습니다. AWS KMS 비대칭 KMS 키의 프라이빗 키는 Raw RSA 키링으로 내보낼 수 없습니다. AWS KMS 복호화 작업은이 AWS Encryption SDK 반환하는 암호화된 메시지를 복호화할 수 없습니다.

에서 Raw RSA 키링을 구성할 때는 각 키를 경로 또는 파일 이름이 아닌 null로 종료된 C-문자열로 포함하는 PEM 파일의 내용을 제공해야 AWS Encryption SDK for C합니다. JavaScript에서 Raw RSA 키링을 구성하면 다른 언어 구현과 호환되지 않을 수 있습니다.

네임스페이스 및 이름

키링에서 RSA 키 자료를 식별하기 위해 Raw RSA 키링은 사용자가 제공한 키 네임스페이스와 키 이름을 사용합니다. 이 값은 비밀이 아닙니다. 암호화 작업이 반환하는 암호화된 메시지 헤더에 일반 텍스트로 나타납니다. HSM 또는 키 관리 시스템에서 RSA 키 페어(또는 프라이빗 키)를 식별하는 키 네임스페이스와 키 이름을 사용하는 것이 좋습니다.

> **Note**
>
> 키 네임스페이스와 키 이름은 `JceMasterKey` 및 `RawMasterKey`의 공급자 ID(또는 공급자) 및 키 ID 필드와 동일합니다.
> 는 KMS `aws-kms` 키에 대한 키 네임스페이스 값을 AWS Encryption SDK for C 예약합니다. Raw AES 키링 또는 Raw RSA 키링을 AWS Encryption SDK for C와 함께 사용하지 마세요.

특정 메시지를 암호화하고 복호화하기 위해 서로 다른 키링을 구성하는 경우 네임스페이스와 이름 값이 중요합니다. 복호화 키링의 키 네임스페이스와 키 이름이 대/소문자를 구분하여 암호화 키링의 키 네임스페이스와 키 이름이 정확히 일치하지 않으면 키가 동일한 키 페어에 속하더라도 복호화 키링이 사용되지 않습니다.

암호화 및 복호화 키링에 있는 키 자료의 키 네임스페이스와 키 이름은 키링에 RSA 퍼블릭 키, RSA 프라이빗 키 또는 키 페어의 두 키가 모두 포함되어 있는지 여부에 관계없이 동일해야 합니다. 예를 들어 키 네임스페이스 `HSM_01` 및 키 이름 `RSA_2048_06`이 있는 RSA 퍼블릭 키에 대한 Raw RSA 키링으로 데이터를 암호화한다고 가정해 보겠습니다. 해당 데이터를 복호화하려면 프라이빗 키(또는 키 페어)와 동일한 키 네임스페이스 및 이름을 사용하여 Raw RSA 키링을 구성하세요.

패딩 모드

암호화 및 복호화에 사용되는 Raw RSA 키링의 패딩 모드를 지정하거나 해당 패딩 모드를 지정하는 언어 구현 기능을 사용해야 합니다.

는 각 언어의 제약 조건에 따라 다음과 같은 패딩 모드를 AWS Encryption SDK 지원합니다. OAEP 패딩 모드, 특히 SHA-256을 사용하는 OAEP와 SHA-256 패딩을 사용하는 MGF1을 추천합니다. PKCS1 패딩 모드는 이하 버전과의 호환성을 위해서만 지원됩니다.

- SHA-1 패딩 모드가 있는 OAEP 및 MGF1
- SHA-256 패딩 모드가 있는 OAEP 및 MGF1
- SHA-384 패딩 모드가 있는 OAEP 및 MGF1
- SHA-512 패딩 모드가 있는 OAEP 및 MGF1
- PKCS1 v1.5 패딩

다음 예제는 SHA-256 패딩 모드가 있는 MGF1과 RSA 키 쌍의 공개 및 개인 키를 사용하여 원시 RSA 키링을 생성하는 방법을 보여줍니다. `RSAPublicKey` 및 `RSAPrivateKey` 변수는 사용자가 제공하는 키 자료를 나타냅니다.

C

에서 Raw RSA 키링을 생성하려면를 AWS Encryption SDK for C사용합니다`aws_cryptosdk_raw_rsa_keyring_new`.

에서 Raw RSA 키링을 구성할 때는 각 키를 경로 또는 파일 이름이 아닌 null로 종료된 C-문자열로 포함하는 PEM 파일의 내용을 제공해야 AWS Encryption SDK for C합니다. 전체 예제를 보려면 raw_rsa_keyring.c를 참조하세요.

```
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

AWS Encryption SDK for .NET에서 Raw RSA 키링을 인스턴스화하려면 `materialProviders.CreateRawRsaKeyring()` 메서드를 사용합니다. 전체 예제는 RawRSAKeyringExample.cs를 참조하세요.

다음 예제에서는 AWS Encryption SDK for .NET 4.x 버전을 사용합니다.

```
// 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 = "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

브라우저 AWS Encryption SDK for JavaScript 의는 WebCrypto 라이브러리에서 암호화 프리미티브를 가져옵니다. 키링을 구성하기 전에 WebCrypto 백엔드로 원시 키 자료를 가져오는 데 `importPublicKey()` 또는 `importPrivateKey()`를 사용해야 합니다. 이렇게 하면 WebCrypto에 대한 모든 호출이 비동기식이어도 키링이 완료됩니다. 가져오기 메서드가 사용하는 객체에는 래핑 알고리즘과 해당 패딩 모드가 포함됩니다.

키 자료를 가져온 후 `RawRsaKeyringWebCrypto()` 메서드를 사용하여 키링을 인스턴스화하세요. JavaScript에서 Raw RSA 키링을 구성하면 다른 언어 구현과 호환되지 않을 수 있습니다.

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한`buildClient`할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

전체 예제는 rsa_simple.ts(JavaScript 브라우저)를 참조하세요.

```
import {
  RsaImportableKey,
  RawRsaKeyringWebCrypto,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

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

const publicKey = await RawRsaKeyringWebCrypto.importPublicKey(
  publicRsaJwKKey
)

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'

const keyring = new RawRsaKeyringWebCrypto({
  keyName,
  keyNamespace,
  publicKey,
  privateKey,
})
```

JavaScript Node.js

Node.js AWS Encryption SDK for JavaScript 용에서 Raw RSA 키링을 인스턴스화하려면 `RawRsaKeyringNode` 클래스의 새 인스턴스를 생성합니다. `wrapKey` 파라미터는 퍼블릭 키를 보유합니다. `unwrapKey`파라미터는 프라이빗 키를 보유합니다. 기본 패딩 모드를 지정할 수는 있지만 `RawRsaKeyringNode` 생성자가 기본 패딩 모드를 자동으로 계산합니다.

JavaScript에서 Raw RSA 키링을 구성하면 다른 언어 구현과 호환되지 않을 수 있습니다.

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한`buildClient`할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

전체 예제를 보려면 rsa_simple.ts(JavaScript Node.js)를 참조하세요.

```
import {
  RawRsaKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
```

```
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'

const keyring = new RawRsaKeyringNode({ keyName, keyNamespace, rsaPublicKey,
 rsaPrivateKey})
```

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())
        .build();
IKeyring rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

Python

다음 예시에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 전체 예제는 GitHub의 AWS Encryption SDK for Python 리포지토리에서 raw_rsa_keyring_example.py를 참조하세요.

```
# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "RSA_2048_06"

# Instantiate the material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create Raw RSA keyring
keyring_input: CreateRawRsaKeyringInput = CreateRawRsaKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
```

```
    padding_scheme=PaddingScheme.OAEP_SHA256_MGF1,
    public_key=RSAPublicKey,
    private_key=RSAPrivateKey
)

raw_rsa_keyring: IKeyring = mat_prov.create_raw_rsa_keyring(
    input=keyring_input
)
```

### Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "RSA_2048_06";

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create Raw RSA keyring
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
    .public_key(aws_smithy_types::Blob::new(RSAPublicKey))
    .private_key(aws_smithy_types::Blob::new(RSAPrivateKey))
    .send()
    .await?;
```

Go

```
// Instantiate the material providers library
matProv, err :=
 awscryptographymaterialproviderssmithygenerated.NewClient(awscryptographymaterialproviderss

// Create Raw RSA keyring
rsaKeyRingInput :=
 awscryptographymaterialproviderssmithygeneratedtypes.CreateRawRsaKeyringInput{
 KeyName:       "rsa",
 KeyNamespace:  "rsa-keyring",
 PaddingScheme:
 awscryptographymaterialproviderssmithygeneratedtypes.PaddingSchemePkcs1,
 PublicKey:     pem.EncodeToMemory(publicKeyBlock),
 PrivateKey:    pem.EncodeToMemory(privateKeyBlock),
}

rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create an encryption context
encryptionContext := map[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",
}

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "RSA_2048_06"

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create Raw RSA keyring
rsaKeyRingInput := mpltypes.CreateRawRsaKeyringInput{
    KeyName:       keyName,
    KeyNamespace:  keyNamespace,
    PaddingScheme: mpltypes.PaddingSchemeOaepSha512Mgf1,
    PublicKey:     (RSAPublicKey),
    PrivateKey:    (RSAPrivateKey),
}
rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
if err != nil {
    panic(err)
}
```

# 원시 ECDH 키링

원시 ECDH 키링은 사용자가 제공하는 타원 곡선 퍼블릭-프라이빗 키 페어를 사용하여 두 당사자 간에 공유 래핑 키를 도출합니다. 먼저 키링은 발신자의 프라이빗 키, 수신자의 퍼블릭 키 및 ECDH(Elliptic Curve Diffie-Hellman) 키 계약 알고리즘을 사용하여 공유 보안 암호를 추출합니다. 그런 다음 키링은 공유 보안 암호를 사용하여 데이터 암호화 키를 보호하는 공유 래핑 키를 파생합니다. 가 (KDF_CTR_HMAC_SHA384)를 AWS Encryption SDK 사용하여 공유 래핑 키를 파생하는 키 파생 함수는 키 파생에 대한 NIST 권장 사항을 준수합니다.

키 파생 함수는 64바이트의 키 구성 요소를 반환합니다. 양 당사자가 올바른 키 구성 요소를 사용하도록 하기 위해는 처음 32바이트를 커밋 키로 사용하고 마지막 32바이트를 공유 래핑 키로 AWS Encryption SDK 사용합니다. 복호화 시 키링이 메시지 헤더 사이퍼텍스트에 저장된 동일한 커밋 키와 공유 래핑 키를 재현할 수 없는 경우 작업이 실패합니다. 예를 들어 Alice의 프라이빗 키와 Bob의 퍼블릭 키로 구성된 키링으로 데이터를 암호화하는 경우 Bob의 프라이빗 키와 Alice의 퍼블릭 키로 구성된 키링은 동일한 커밋 키와 공유 래핑 키를 재현하고 데이터를 해독할 수 있습니다. Bob의 퍼블릭 키가 AWS KMS key 페어에서 가져온 경우 Bob은 AWS KMS ECDH 키링을 생성하여 데이터를 해독할 수 있습니다.

원시 ECDH 키링은 AES-GCM을 사용하여 대칭 키로 데이터를 암호화합니다. 그런 다음 AES-GCM을 사용하여 파생된 공유 래핑 키로 데이터 키를 봉투 암호화합니다. 각 Raw ECDH 키링에는 공유 래핑 키가 하나만 있을 수 있지만, 단독으로 또는 다른 키링과 함께 여러 Raw ECDH 키링을 다중 키링에 포함할 수 있습니다.

사용자는 가급적이면 하드웨어 보안 모듈(HSM) 또는 키 관리 시스템에서 프라이빗 키를 생성, 저장 및 보호할 책임이 있습니다. 발신자와 수신자의 키 페어는 동일한 타원 곡선에 많이 있습니다. 는 다음과 같은 타원 곡선 사양을 AWS Encryption SDK 지원합니다.

- `ECC_NIST_P256`
- `ECC_NIST_P384`
- `ECC_NIST_P512`

프로그래밍 언어 호환성

원시 ECDH 키링은 암호화 자료 공급자 라이브러리(MPL) 버전 1.5.0에 도입되었으며 다음 프로그래밍 언어 및 버전에서 지원됩니다.

- 의 버전 3.x AWS Encryption SDK for Java
- for .NET 버전 AWS Encryption SDK 4.x
- 선택적 MPL 종속성과 함께 사용할 AWS Encryption SDK for Python경우 버전 4.x.
- AWS Encryption SDK for Rust 버전 1.x
- Go AWS Encryption SDK 용의 버전 0.1.x 이상

## 원시 ECDH 키링 생성

원시 ECDH 키링은 `RawPrivateKeyToStaticPublicKey`, 및 `EphemeralPrivateKeyToStaticPublicKey`의 세 가지 주요 계약 스키마를 지원합니

다PublicKeyDiscovery. 선택하는 키 계약 스키마에 따라 수행할 수 있는 암호화 작업과 키 구성 요소가 조합되는 방식이 결정됩니다.

주제

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

### RawPrivateKeyToStaticPublicKey

RawPrivateKeyToStaticPublicKey 키 계약 스키마를 사용하여 키링에서 발신자의 프라이빗 키와 수신자의 퍼블릭 키를 정적으로 구성합니다. 이 키 계약 스키마는 데이터를 암호화하고 해독할 수 있습니다.

RawPrivateKeyToStaticPublicKey 키 계약 스키마를 사용하여 원시 ECDH 키링을 초기화하려면 다음 값을 제공합니다.

- 발신자의 프라이빗 키

  RFC 5958에 정의된 대로 발신자의 PEM 인코딩 프라이빗 키(PKCS #8 PrivateKeyInfo 구조)를 제공해야 합니다.
- 수신자의 퍼블릭 키

  RFC 5280에 정의된 대로 SubjectPublicKeyInfo (SPKI)라고도 하는 수신자의 DER 인코딩 X.509 퍼블릭 키를 제공해야 합니다. https://tools.ietf.org/html/rfc5280

  비대칭 키 계약 KMS 키 페어의 퍼블릭 키 또는 외부에서 생성된 키 페어의 퍼블릭 키를 지정할 수 있습니다 AWS.
- 곡선 사양

  지정된 키 페어에서 타원 곡선 사양을 식별합니다. 발신자와 수신자의 키 페어 모두 곡선 사양이 동일해야 합니다.

  유효한 값: ECC_NIST_P256, ECC_NIS_P384, ECC_NIST_P512

C# / .NET

```
// Instantiate material providers
```

```
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var BobPrivateKey = new MemoryStream(new byte[] { });
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH static keyring
     var staticConfiguration = new RawEcdhStaticConfigurations()
     {
      RawPrivateKeyToStaticPublicKey = new RawPrivateKeyToStaticPublicKeyInput
      {
       SenderStaticPrivateKey = BobPrivateKey,
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = staticConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

다음 Java 예제에서는 RawPrivateKeyToStaticPublicKey 키 계약 스키마를 사용하여 발신자의 프라이빗 키와 수신자의 퍼블릭 키를 정적으로 구성합니다. 두 키 페어 모두 ECC_NIST_P256 곡선에 있습니다.

```
private static void StaticRawKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair senderKeys = GetRawEccKey();
    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH static keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
```

```
          RawEcdhStaticConfigurations.builder()
            .RawPrivateKeyToStaticPublicKey(
                RawPrivateKeyToStaticPublicKeyInput.builder()
                  // Must be a PEM-encoded private key

 .senderStaticPrivateKey(ByteBuffer.wrap(senderKeys.getPrivate().getEncoded()))
                  // Must be a DER-encoded X.509 public key

 .recipientPublicKey(ByteBuffer.wrap(recipient.getPublic().getEncoded()))
                  .build()
            )
            .build()
        ).build();

    final IKeyring staticKeyring =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Python

다음 Python 예제에서는 RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey 키 계약 스키마를 사용하여 발신자의 프라이빗 키와 수신자의 퍼블릭 키를 정적으로 구성합니다. 두 키 페어 모두 ECC_NIST_P256 곡선에 있습니다.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey,
    RawPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Must be a PEM-encoded private key
bob_private_key = get_private_key_bytes()
# Must be a DER-encoded X.509 public key
alice_public_key = get_public_key_bytes()
```

```
# Create the raw ECDH static keyring
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey(
        RawPrivateKeyToStaticPublicKeyInput(
            sender_static_private_key = bob_private_key,
            recipient_public_key = alice_public_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

다음 Python 예제에서는 `raw_ecdh_static_configuration` 키 계약 스키마를 사용하여 발신자의 프라이빗 키와 수신자의 퍼블릭 키를 정적으로 구성합니다. 두 키 페어 모두 동일한 곡선에 있어야 합니다.

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Create keyring input
let raw_ecdh_static_configuration_input =
    RawPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .sender_static_private_key(private_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;
```

```
let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH static keyring
let raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create your encryption context
encryptionContext := map[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",
}

// Create keyring input
rawEcdhStaticConfigurationInput := mpltypes.RawPrivateKeyToStaticPublicKeyInput{
    SenderStaticPrivateKey: privateKeySender,
    RecipientPublicKey:     publicKeyRecipient,
}
rawECDHStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberRawPrivateKeyToStaticPublicKey{
    Value: rawEcdhStaticConfigurationInput,
}
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: rawECDHStaticConfiguration,
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create raw ECDH static keyring
rawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

## EphemeralPrivateKeyToStaticPublicKey

키 계약 스키마로 구성된 EphemeralPrivateKeyToStaticPublicKey 키링은 로컬에서 새 키 페어를 생성하고 각 암호화 호출에 대해 고유한 공유 래핑 키를 도출합니다.

이 키 계약 스키마는 메시지만 암호화할 수 있습니다.
EphemeralPrivateKeyToStaticPublicKey 키 계약 스키마로 암호화된 메시지를 복호화하려면 동일한 수신자의 퍼블릭 키로 구성된 검색 키 계약 스키마를 사용해야 합니다. 복호화하려면 PublicKeyDiscovery 키 계약 알고리즘과 함께 원시 ECDH 키링을 사용하거나 수신자의 퍼블릭 키가 비대칭 키 계약 KMS 키 페어에서 가져온 경우 KmsPublicKeyDiscovery 키 계약 스키마와 함께 AWS KMS ECDH 키링을 사용할 수 있습니다.

EphemeralPrivateKeyToStaticPublicKey 키 계약 스키마를 사용하여 원시 ECDH 키링을 초기화하려면 다음 값을 제공합니다.

- 수신자의 퍼블릭 키

  RFC 5280에 정의된 대로 SubjectPublicKeyInfo (SPKI)라고도 하는 수신자의 DER 인코딩 X.509 퍼블릭 키를 제공해야 합니다. https://tools.ietf.org/html/rfc5280

  비대칭 키 계약 KMS 키 페어의 퍼블릭 키 또는 외부에서 생성된 키 페어의 퍼블릭 키를 지정할 수 있습니다 AWS.

- 곡선 사양

  지정된 퍼블릭 키에서 타원 곡선 사양을 식별합니다.

  암호화 시 키링은 지정된 곡선에 새 키 페어를 생성하고 새 프라이빗 키와 지정된 퍼블릭 키를 사용하여 공유 래핑 키를 도출합니다.

  유효한 값: ECC_NIST_P256, ECC_NIS_P384, ECC_NIST_P512

C# / .NET

다음 예시에서는 EphemeralPrivateKeyToStaticPublicKey 키 계약 스키마를 사용하여 Raw ECDH 키링을 생성합니다. 암호화 시 키링은 지정된 ECC_NIST_P256 곡선에 로컬로 새 키 페어를 생성합니다.

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH ephemeral keyring
     var ephemeralConfiguration = new RawEcdhStaticConfigurations()
     {
      EphemeralPrivateKeyToStaticPublicKey = new
 EphemeralPrivateKeyToStaticPublicKeyInput
      {
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
```

```
        CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
        KeyAgreementScheme = ephemeralConfiguration
      };

      var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

다음 예시에서는 EphemeralPrivateKeyToStaticPublicKey 키 계약 스키마를 사용하여 Raw ECDH 키링을 생성합니다. 암호화 시 키링은 지정된 ECC_NIST_P256 곡선에 로컬로 새 키 페어를 생성합니다.

```
private static void EphemeralRawEcdhKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    ByteBuffer recipientPublicKey = getPublicKeyBytes();

    // Create the Raw ECDH ephemeral keyring
    final CreateRawEcdhKeyringInput ephemeralInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .EphemeralPrivateKeyToStaticPublicKey(
              EphemeralPrivateKeyToStaticPublicKeyInput.builder()
                .recipientPublicKey(recipientPublicKey)
                .build()
            )
            .build()
        ).build();

    final IKeyring ephemeralKeyring =
 materialProviders.CreateRawEcdhKeyring(ephemeralInput);
}
```

Python

다음 예시에서는 RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey 키 계약 스

키마를 사용하여 원시 ECDH 키링을 생성합니다. 암호화 시 키링은 지정된 ECC_NIST_P256 곡선에 로컬로 새 키 페어를 생성합니다.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey,
    EphemeralPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Your get_public_key_bytes must return a DER-encoded X.509 public key
recipient_public_key = get_public_key_bytes()

# Create the raw ECDH ephemeral private key keyring
ephemeral_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey(
        EphemeralPrivateKeyToStaticPublicKeyInput(
            recipient_public_key = recipient_public_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(ephemeral_input)
```

Rust

다음 예시에서는 키 계약 스키마를 사용하여 원시 ECDH ephemeral_raw_ecdh_static_configuration 키링을 생성합니다. 암호화 시 키링은 지정된 곡선에 로컬로 새 키 페어를 생성합니다.

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;
```

```
// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Load public key from UTF-8 encoded PEM files into a DER encoded public key.
let public_key_file_content =
 std::fs::read_to_string(Path::new(EXAMPLE_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content = parse(public_key_file_content)?;
let public_key_recipient_utf8_bytes = parsed_public_key_file_content.contents();

// Create EphemeralPrivateKeyToStaticPublicKeyInput
let ephemeral_raw_ecdh_static_configuration_input =
    EphemeralPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
    .send()
    .await?;
```

## Go

```
import (
    "context"
```

```
 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create your encryption context
encryptionContext := map[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",
}

// Load public key from UTF-8 encoded PEM files into a DER encoded public key
publicKeyRecipient, err := LoadPublicKeyFromPEM(eccPublicKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Create EphemeralPrivateKeyToStaticPublicKeyInput
ephemeralRawEcdhStaticConfigurationInput :=
 mpltypes.EphemeralPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey: publicKeyRecipient,
}
ephemeralRawECDHStaticConfiguration :=
    mpltypes.RawEcdhStaticConfigurationsMemberEphemeralPrivateKeyToStaticPublicKey{
        Value: ephemeralRawEcdhStaticConfigurationInput,
    }

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
```

```
if err != nil {
    panic(err)
}

// Create raw ECDH ephemeral private key keyring
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: &ephemeralRawECDHStaticConfiguration,
}
ecdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

### PublicKeyDiscovery

복호화할 때는에서 사용할 AWS Encryption SDK 수 있는 래핑 키를 지정하는 것이 가장 좋습니다. 이 모범 사례를 따르려면 발신자의 프라이빗 키와 수신자의 퍼블릭 키를 모두 지정하는 ECDH 키링을 사용합니다. 그러나 원시 ECDH 검색 키링, 즉 지정된 키의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있는 원시 ECDH 키링을 생성할 수도 있습니다. 이 키 계약 스키마는 메시지만 복호화할 수 있습니다.

> **Important**
>
> `PublicKeyDiscovery` 키 계약 스키마를 사용하여 메시지를 복호화하는 경우 누가 소유하든 모든 퍼블릭 키를 수락합니다.

`PublicKeyDiscovery` 키 계약 스키마를 사용하여 원시 ECDH 키링을 초기화하려면 다음 값을 제공합니다.

- 수신자의 정적 프라이빗 키

  RFC 5958에 정의된 대로 수신자의 PEM 인코딩 프라이빗 키(PKCS #8 PrivateKeyInfo 구조)를 제공해야 합니다.
- 곡선 사양

지정된 프라이빗 키에서 타원 곡선 사양을 식별합니다. 발신자와 수신자의 키 페어 모두 곡선 사양이 동일해야 합니다.

유효한 값: ECC_NIST_P256, ECC_NIS_P384, ECC_NIST_P512

### C# / .NET

다음 예시에서는 PublicKeyDiscovery 키 계약 스키마를 사용하여 원시 ECDH 키링을 생성합니다. 이 키링은 지정된 프라이빗 키의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있습니다.

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePrivateKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH discovery keyring
     var discoveryConfiguration = new RawEcdhStaticConfigurations()
     {
      PublicKeyDiscovery = new PublicKeyDiscoveryInput
      {
       RecipientStaticPrivateKey = AlicePrivateKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = discoveryConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

### Java

다음 예시에서는 PublicKeyDiscovery 키 계약 스키마를 사용하여 원시 ECDH 키링을 생성합니다. 이 키링은 지정된 프라이빗 키의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있습니다.

```
private static void RawEcdhDiscovery() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
```

```
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH discovery keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .PublicKeyDiscovery(
              PublicKeyDiscoveryInput.builder()
                // Must be a PEM-encoded private key

 .recipientStaticPrivateKey(ByteBuffer.wrap(sender.getPrivate().getEncoded()))
                .build()
            )
            .build()
        ).build();

    final IKeyring publicKeyDiscovery  =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Python

다음 예시에서는 RawEcdhStaticConfigurationsPublicKeyDiscovery 키 계약 스키마를 사용하여 Raw ECDH 키링을 생성합니다. 이 키링은 지정된 프라이빗 키의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있습니다.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsPublicKeyDiscovery,
    PublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
```

```
    config=MaterialProvidersConfig()
)

# Your get_private_key_bytes must return a PEM-encoded private key
recipient_private_key = get_private_key_bytes()

# Create the raw ECDH discovery keyring
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = RawEcdhStaticConfigurationsPublicKeyDiscovery(
        PublicKeyDiscoveryInput(
            recipient_static_private_key = recipient_private_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

다음 예시에서는 discovery_raw_ecdh_static_configuration 키 계약 스키마를 사용하여 원시 ECDH 키링을 생성합니다. 이 키링은 지정된 프라이빗 키의 퍼블릭 키가 메시지 사이퍼텍스트에 저장된 수신자의 퍼블릭 키와 일치하는 모든 메시지를 복호화할 수 있습니다.

```
// Instantiate the AWS Encryption SDK client and material providers library
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;


// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Load keys from UTF-8 encoded PEM files.
```

```
let mut file = File::open(Path::new(EXAMPLE_ECC_PRIVATE_KEY_FILENAME_RECIPIENT))?;
let mut private_key_recipient_utf8_bytes = Vec::new();
file.read_to_end(&mut private_key_recipient_utf8_bytes)?;

// Create PublicKeyDiscoveryInput
let discovery_raw_ecdh_static_configuration_input =
    PublicKeyDiscoveryInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .recipient_static_private_key(private_key_recipient_utf8_bytes)
        .build()?;

let discovery_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::PublicKeyDiscovery(discovery_raw_ecdh_static_configuration_inp

// Create raw ECDH discovery private key keyring
let discovery_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(discovery_raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
```

```
// Optional: Create your encryption context
encryptionContext := map[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",
}

// Load keys from UTF-8 encoded PEM files.
privateKeyRecipient, err := os.ReadFile(eccPrivateKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create PublicKeyDiscoveryInput
discoveryRawEcdhStaticConfigurationInput := mpltypes.PublicKeyDiscoveryInput{
    RecipientStaticPrivateKey: privateKeyRecipient,
}

discoveryRawEcdhStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberPublicKeyDiscovery{
    Value: discoveryRawEcdhStaticConfigurationInput,
}

// Create raw ECDH discovery private key keyring
discoveryRawEcdhKeyringInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: discoveryRawEcdhStaticConfiguration,
}

discoveryRawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 discoveryRawEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

# 다중 키링

키링을 여러 개의 키링으로 결합할 수 있습니다. 다중 키링은 유형이 같거나 다른 하나 이상의 개별 키링으로 구성된 키링입니다. 이 효과는 여러 개의 키링을 연속으로 사용하는 것과 같습니다. 다중 키링을 사용하여 데이터를 암호화하는 경우 해당 키링의 모든 래핑 키로 해당 데이터를 복호화할 수 있습니다.

다중 키링을 생성하여 데이터를 암호화하는 경우, 키링 중 하나를 생성기 키링으로 지정하세요. 다른 모든 키링은 하위 키링이라고 합니다. 생성기 키링은 일반 텍스트 데이터 키를 생성하고 암호화합니다. 그러면 모든 하위 키링의 모든 래핑 키가 동일한 일반 텍스트 데이터 키를 암호화합니다. 다중 키링은 다중 키링의 각 래핑 키에 대해 일반 텍스트 키와 암호화된 데이터 키 하나를 반환합니다. 생성기 키링이 KMS 키링인 경우 AWS KMS 의 생성기 키가 일반 텍스트 키를 생성하고 암호화합니다. 그런 다음 AWS KMS 키링의 모든 추가 AWS KMS keys 키와 다중 키링의 모든 하위 키링의 모든 래핑 키는 동일한 일반 텍스트 키를 암호화합니다.

생성기 키링 없이 다중 키링을 생성하는 경우 자체적으로 사용하여 데이터를 복호화할 수 있지만 암호화할 수는 없습니다. 또는 암호화 작업에서 genertor 키링이 없는 다중 키링을 사용하려면 다른 다중 키링에서 하위 키링으로 지정할 수 있습니다. 생성기 키링이 없는 다중 키링은 다른 다중 키링에서 생성기 키링으로 지정할 수 없습니다.

복호화 시 AWS Encryption SDK 는 키링을 사용하여 암호화된 데이터 키 중 하나를 복호화하려고 시도합니다. 키링은 다중 키링에 지정된 순서대로 호출됩니다. 모든 키 링의 모든 키가 암호화된 데이터 키를 복호화할 수 있는 즉시 처리가 중지됩니다.

버전 1.7.x부터 암호화된 데이터 키가 AWS Key Management Service (AWS KMS) 키링(또는 마스터 키 공급자)으로 암호화되면는 AWS Encryption SDK 항상의 키 ARN을 AWS KMS Decrypt 작업의 `KeyId` 파라미터 AWS KMS key 로 전달합니다. 이는 사용하려는 래핑 키로 암호화된 데이터 키를 해독하도록 보장하는 AWS KMS 모범 사례입니다.

다중 키링의 작동 예제를 보려면 다음을 참조하세요.

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

- Python: multi_keyring_example.py

다중 키링을 만들려면 먼저 하위 키링을 인스턴스화하세요. 이 예제에서는 AWS KMS 키링과 원시 AES 키링을 사용하지만 지원되는 모든 키링을 다중 키링에 결합할 수 있습니다.

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

다음 예제에서는 buildClient 함수를 사용하여 기본 커밋 정책인를 지정합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한buildClient할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

```
import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  RawAesKeyringWebCrypto,
  RawAesWrappingSuiteIdentifier,
  MultiKeyringWebCrypto,
  buildClient,
  CommitmentPolicy,
  synchronousRandomValues,
```

```
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한`buildClient`할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

```
import {
  MultiKeyringNode,
  KmsKeyringNode,
  RawAesKeyringNode,
  RawAesWrappingSuiteIdentifier,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

// 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

```
// 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();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

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

Python

다음 예시에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT.

```
# Create the AWS KMS keyring
kms_client = boto3.client('kms', region_name="us-west-2")

mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

kms_keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
    generator=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    kms_client=kms_client
)

kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
    input=kms_keyring_input
```

```
)

# Create Raw AES keyring
key_name_space = "HSM_01"
key_name = "AES_256_012"

raw_aes_keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)

raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=raw_aes_keyring_input
)
```

### Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create the AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create an AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

// Create a Raw AES keyring
let key_namespace: &str = "my-key-namespace";
let key_name: &str = "my-aes-key-name";
```

```
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Instantiate the material providers library
```

```
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS keyring
awsKmsKeyringInput := mpltypes.CreateAwsKmsKeyringInput{
    KmsClient: kmsClient,
    KmsKeyId:  kmsKeyId,
}
awsKmsKeyring, err := matProv.CreateAwsKmsKeyring(context.Background(),
 awsKmsKeyringInput)
if err != nil {
    panic(err)
}

// Create a Raw AES keyring
var keyNamespace = "my-key-namespace"
var keyName = "my-aes-key-name"

aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  AESWrappingKey,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
```

그런 다음 다중 키링을 만들고 생성기 키링(있는 경우)을 지정합니다. 이 예제에서는 AWS KMS 키링이 생성기 키링이고 AES 키링이 하위 키링인 다중 키링을 생성합니다.

C

C의 다중 키링 생성자에서는 생성기 키링만 지정합니다.

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

다중 키링에 하위 키링을 추가하려면 `aws_cryptosdk_multi_keyring_add_child` 메서드를 사용합니다. 추가하는 각 하위 키링에 대해 메서드를 한 번 호출해야 합니다.

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

C# / .NET

.NET `CreateMultiKeyringInput` 생성자를 사용하면 생성기 키링과 자식 키링을 정의할 수 있습니다. 결과 `CreateMultiKeyringInput` 객체는 변경할 수 없습니다.

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

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

JavaScript Browser

JavaScript 다중 키링은 변경할 수 없습니다. JavaScript 다중 키링 생성자를 사용하면 생성기 키링과 여러 하위 키링을 지정할 수 있습니다.

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

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

JavaScript Node.js

JavaScript 다중 키링은 변경할 수 없습니다. JavaScript 다중 키링 생성자를 사용하면 생성기 키링과 여러 하위 키링을 지정할 수 있습니다.

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

Java

Java `CreateMultiKeyringInput` 생성자를 사용하면 생성기 키링과 하위 키링을 정의할 수 있습니다. 결과 `createMultiKeyringInput` 객체는 변경할 수 없습니다.

```
final CreateMultiKeyringInput createMultiKeyringInput =
 CreateMultiKeyringInput.builder()
```

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

Python

```
multi_keyring_input: CreateMultiKeyringInput = CreateMultiKeyringInput(
    generator=kms_keyring,
    child_keyrings=[raw_aes_keyring]
)

multi_keyring: IKeyring = mat_prov.create_multi_keyring(
    input=multi_keyring_input
)
```

Rust

```
let multi_keyring = mpl
    .create_multi_keyring()
    .generator(kms_keyring.clone())
    .child_keyrings(vec![raw_aes_keyring.clone()])
    .send()
    .await?;
```

Go

```
createMultiKeyringInput := mpltypes.CreateMultiKeyringInput{
  Generator:     awsKmsKeyring,
  ChildKeyrings: []mpltypes.IKeyring{rawAESKeyring},
 }
 multiKeyring, err := matProv.CreateMultiKeyring(context.Background(),
 createMultiKeyringInput)
 if err != nil {
  panic(err)
 }
```

이제 다중 키링을 사용하여 데이터를 암호화 및 복호화할 수 있습니다.

# AWS Encryption SDK 프로그래밍 언어

AWS Encryption SDK 는 다음 프로그래밍 언어에 사용할 수 있습니다. 모든 언어 구현은 상호 연동이 가능합니다. 하나의 언어 구현으로 암호화하고 다른 언어 구현으로 복호화할 수 있습니다. 상호 연동성에는 언어 제약 조건이 적용될 수 있습니다. 이 경우 이러한 제약 조건은 언어 구현에 대한 주제에 설명되어 있습니다. 또한 암호화 및 복호화를 수행할 때는 호환되는 키링이나 마스터 키 및 마스터 키 공급자를 사용해야 합니다. 자세한 내용은 the section called “키링 호환성” 섹션을 참조하세요.

주제



## AWS Encryption SDK for C

는 C로 애플리케이션을 작성하는 개발자를 위한 클라이언트 측 암호화 라이브러리를 AWS Encryption SDK for C 제공합니다. 또한 AWS Encryption SDK 상위 수준 프로그래밍 언어로를 구현하기 위한 기반 역할을 합니다.

의 모든 구현과 마찬가지로 AWS Encryption SDK는 고급 데이터 보호 기능을 AWS Encryption SDK for C 제공합니다. 이러한 기능에는 봉투 암호화, 추가 인증 데이터(AAD), 안전하고 인증된 대칭 키 알고리즘 제품군(예: 키 유도 및 서명을 사용하는 256비트 AES-GCM)이 포함됩니다.

의 모든 언어별 구현 AWS Encryption SDK 은 완전히 상호 운용 가능합니다. 예를 들어,를 사용하여 데이터를 암호화 AWS Encryption SDK for C 하고 AWS Encryption CLI를 포함하여 지원되는 언어 구현으로 데이터를 해독할 수 있습니다.

를 AWS Encryption SDK for C 사용하려면가 AWS Key Management Service ()와 상호 작용 AWS SDK for C++ 해야 합니다AWS KMS. 선택 사항인 AWS KMS 키링을 사용하는 경우에만 이를 사용해

야 합니다. 그러나 AWS Encryption SDK 에는 AWS KMS 또는 다른 AWS 서비스가 필요하지 않습니다.

자세히 알아보기

- 를 사용한 프로그래밍에 대한 자세한 내용은 C 예제, GitHubhttps://github.com/aws/aws-encryption-sdk-c/tree/master/examples의 aws-encryption-sdk-c 리포지토리 예제 및 AWS Encryption SDK for C API 설명서를 AWS Encryption SDK for C참조하세요.
- 여러에서 데이터를 해독할 수 있도록 AWS Encryption SDK for C 를 사용하여 데이터를 암호화하는 방법에 대한 자세한 내용은 AWS 보안 블로그의 C에서를 사용하여 여러 리전의 사이퍼텍스트를 해독하는 방법을 AWS 리전참조하세요. AWS Encryption SDK

주제

- 설치 AWS Encryption SDK for C
- 사용 AWS Encryption SDK for C
- AWS Encryption SDK for C 예제

## 설치 AWS Encryption SDK for C

AWS Encryption SDK for C의 최신 버전을 설치합니다.

> **Note**
>
> 2.0.0 AWS Encryption SDK for C 이전의 모든 버전은 end-of-support 단계에 있습니다. 코드나 데이터를 변경하지 않고 버전 2.0.x 이상에서 AWS Encryption SDK for C 의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.0.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 1.7.x 이하 버전에서 2.0.x 이상 버전으로 업데이트하려면 먼저 AWS Encryption SDK for C의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.

설치 및 빌드에 대한 자세한 지침은 aws-encryption-sdk-c 리포지토리의 README 파일 AWS Encryption SDK for C 에서 확인할 수 있습니다. 여기에는 Amazon Linux, Ubuntu, macOS, Windows 플랫폼에서 빌드하는 방법에 대한 지침이 포함되어 있습니다.

시작하기 전에 AWS Encryption SDK에서 AWS KMS 키링을 사용할지 여부를 결정합니다. AWS KMS 키링을 사용하는 경우를 설치해야 합니다 AWS SDK for C++. AWS SDK는 AWS Key Management

Service ()와 상호 작용하는 데 필요합니다AWS KMS. AWS KMS 키링을 사용하는 경우는를 AWS Encryption SDK AWS KMS 사용하여 데이터를 보호하는 암호화 키를 생성하고 보호합니다.

원시 AES 키링, 원시 RSA 키링 또는 키링이 포함되지 않은 다중 키링과 같은 다른 AWS KMS 키링 유형을 사용하는 AWS SDK for C++ 경우를 설치할 필요가 없습니다. 하지만 원시 키링 유형을 사용하는 경우 원시 래핑 키를 직접 생성하고 보호해야 합니다.

설치에 문제가 있는 경우 `aws-encryption-sdk-c` 리포지토리에 문제를 제출하거나 이 페이지에 있는 피드백 링크를 사용하세요.

## 사용 AWS Encryption SDK for C

이 주제에서는 다른 프로그래밍 언어 구현에서 지원되지 AWS Encryption SDK for C 않는의 일부 기능에 대해 설명합니다.

이 섹션의 예제에서는 2.0.x 이상 버전의 AWS Encryption SDK for C를 사용하는 방법을 보여줍니다. 이하 버전을 사용하는 예제는 GitHub의 aws-encryption-sdk-c 리포지토리 리포지토리의 릴리스 목록에서 해당하는 릴리스를 찾을 수 있습니다.

를 사용한 프로그래밍에 대한 자세한 내용은 C 예제, GitHubhttps://github.com/aws/aws-encryption-sdk-c/tree/master/examples의 aws-encryption-sdk-c 리포지토리 예제 및 AWS Encryption SDK for C API 설명서를 AWS Encryption SDK for C참조하세요.

또한 키링 섹션도 참조하세요.

주제

- 데이터 암호화 및 복호화 패턴
- 참조 카운트

### 데이터 암호화 및 복호화 패턴

를 사용하는 경우 다음과 유사한 패턴을 AWS Encryption SDK for C따릅니다. 키링 생성, 키링을 사용하는 CMM 생성, CMM(및 키링)을 사용하는 세션 생성, 세션 처리.

1. 오류 문자열을 로드합니다.

   C 또는 C++ 코드에서 `aws_cryptosdk_load_error_strings()` 메서드를 호출합니다. 이 메서드는 디버깅에 매우 유용한 오류 정보를 로드합니다.

   `main` 메서드에서와 같이 한 번만 호출하면 됩니다.

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

2. 키링을 생성합니다.

   데이터 키를 암호화하는 데 사용할 래핑 키로 키링을 구성합니다. 이 예제에서는 AWS KMS 키링을 1과 함께 사용하지만 AWS KMS key그 자리에 모든 유형의 키링을 사용할 수 있습니다.

   AWS KMS key 의 암호화 키링에서를 식별하려면 키 ARN 또는 별칭 ARN을 AWS Encryption SDK for C지정합니다. 복호화 키링에서는 키 ARN을 사용해야 합니다. 자세한 내용은 AWS KMS 키링 AWS KMS keys 에서 식별을 참조하세요.

```
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. 세션을 생성합니다.

   에서 세션을 AWS Encryption SDK for C사용하여 크기에 관계없이 단일 일반 텍스트 메시지를 암호화하거나 단일 사이퍼텍스트 메시지를 해독합니다. 세션은 처리 과정 내내 메시지 상태를 유지합니다.

   할당자, 키링, 모드(AWS_CRYPTOSDK_ENCRYPT 또는 AWS_CRYPTOSDK_DECRYPT)를 사용하여 세션을 구성합니다. 세션 모드를 변경해야 하는 경우 aws_cryptosdk_session_reset 메서드를 사용합니다.

   키링으로 세션을 생성하면가 AWS Encryption SDK for C 자동으로 기본 암호화 자료 관리자(CMM)를 생성합니다. 이 객체를 만들거나 유지 관리하거나 삭제할 필요가 없습니다.

   예를 들어 다음 세션은 1단계에서 정의한 키링과 할당자를 사용합니다. 데이터를 암호화할 때 모드는 AWS_CRYPTOSDK_ENCRYPT입니다.

```
struct aws_cryptosdk_session * session =
 aws_cryptosdk_session_new_from_keyring_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);
```

4. 데이터를 암호화 또는 복호화합니다.

   세션에서 데이터를 처리하려면 aws_cryptosdk_session_process 메서드를 사용합니다. 입력 버퍼가 전체 일반 텍스트를 수용할 수 있을 만큼 크고 출력 버퍼가 전체 사이퍼

텍스트를 수용할 수 있을 만큼 큰 경우, `aws_cryptosdk_session_process_full`을 호출할 수 있습니다. 그러나 스트리밍 데이터를 처리해야 하는 경우 루프에서 `aws_cryptosdk_session_process`를 호출할 수 있습니다. 예를 들어 file_streaming.cpp 예제를 참조하세요. `aws_cryptosdk_session_process_full`는 AWS Encryption SDK 버전 1.9.x 및 2.2.x에 도입되었습니다.

세션이 데이터를 암호화하도록 구성된 경우 일반 텍스트 필드는 입력을 설명하고 사이퍼텍스트 필드는 출력을 설명합니다. `plaintext` 필드에는 암호화하려는 메시지가 들어 있고 `ciphertext` 필드는 암호화 메서드가 반환하는 암호화된 메시지를 가져옵니다.

```
/* Encrypting data */
aws_cryptosdk_session_process_full(session,
                                   ciphertext,
                                   ciphertext_buffer_size,
                                   &ciphertext_length,
                                   plaintext,
                                   plaintext_length)
```

세션이 데이터를 복호화하도록 구성된 경우 사이퍼텍스트 필드는 입력을 설명하고 일반 텍스트 필드는 출력을 설명합니다. `ciphertext` 필드에는 암호화 메서드가 반환한 암호화된 메시지가 들어 있고 `plaintext` 필드는 복호화 메서드가 반환하는 일반 텍스트 메시지를 가져옵니다.

데이터를 복호화하려면 `aws_cryptosdk_session_process_full` 메서드를 호출합니다.

```
/* Decrypting data */
aws_cryptosdk_session_process_full(session,
                                   plaintext,
                                   plaintext_buffer_size,
                                   &plaintext_length,
                                   ciphertext,
                                   ciphertext_length)
```

## 참조 카운트

메모리 누수를 방지하려면 생성하는 모든 객체의 사용을 마친 후 그에 대한 참조를 릴리스해야 합니다. 그러지 않으면 메모리 누수가 발생합니다. SDK는 이 작업을 더 쉽게 수행할 수 있는 방법을 제공합니다.

다음 하위 객체 중 하나를 사용하여 상위 객체를 만들 때마다 상위 객체는 다음과 같이 하위 객체에 대한 참조를 가져오고 유지합니다.

- 키링(예: 키링으로 세션 만들기)
- 기본 암호화 구성 요소 관리자(CMM)(예: 기본 CMM을 사용하여 세션 또는 사용자 지정 CMM 만들기)
- 데이터 키 캐시(예: 키링 및 캐시가 있는 캐싱 CMM 생성)

하위 객체에 대한 독립적인 참조가 필요하지 않은 한, 상위 객체를 만드는 즉시 하위 객체에 대한 참조를 릴리스할 수 있습니다. 상위 객체가 삭제되면 하위 객체에 대한 나머지 참조가 릴리스됩니다. 이 패턴을 사용하면 각 개체에 대한 참조를 필요한 기간 동안만 유지할 수 있으며 릴리스되지 않은 참조로 인해 메모리가 누수되는 것을 방지할 수 있습니다.

명시적으로 만드는 하위 객체에 대한 참조만 릴리스하면 됩니다. 사용자에게는 SDK가 생성하는 객체에 대한 참조를 관리할 책임이 없습니다. SDK가 `aws_cryptosdk_caching_cmm_new_from_keyring` 메서드가 세션에 추가하는 기본 CMM과 같은 객체를 만드는 경우 SDK는 객체와 해당 참조의 생성 및 삭제를 관리합니다.

다음 예제에서 키링이 있는 세션을 만들면 세션은 키링에 대한 참조를 가져오고 세션이 삭제될 때까지 해당 참조를 유지합니다. 키링에 대한 추가 참조를 유지할 필요가 없는 경우, 세션이 만들어지는 즉시 `aws_cryptosdk_keyring_release` 메서드를 사용하여 키링 객체를 릴리스할 수 있습니다. 이 메서드를 사용하면 키링의 참조 횟수가 줄어듭니다. 키링에 대한 세션의 참조는 `aws_cryptosdk_session_destroy`를 호출하여 세션을 삭제할 때 릴리스됩니다.

```
// 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);
```

여러 세션에 대해 키링을 재사용하거나 CMM에서 알고리즘 제품군을 지정하는 등 복잡한 작업의 경우, 객체에 대한 독립적인 참조를 유지해야 할 수 있습니다. 이 경우 릴리스 메서드를 즉시 호출하지 마세요. 대신, 세션을 삭제하는 것 외에도 객체를 더 이상 사용하지 않을 때 참조를 릴리스하세요.

이 참조 카운트 기술은 데이터 키 캐싱을 위한 캐싱 CMM과 같은 대체 CMM을 사용할 때도 사용할 수 있습니다. 캐시와 키링에서 캐싱 CMM을 만들면 캐싱 CMM이 두 객체 모두에 대한 참조를 가져옵니다.

다른 작업에 필요하지 않은 한, 캐싱 CMM이 만들어지는 즉시 캐시 및 키링에 대한 독립적인 참조를 릴리스할 수 있습니다. 그런 다음 캐싱 CMM으로 세션을 만들 때, 캐싱 CMM에 대한 참조를 릴리스할 수 있습니다.

명시적으로 생성하는 객체에 대한 참조만 릴리스하면 됩니다. 캐싱 CMM의 기반이 되는 기본 CMM과 같이 메서드가 만드는 객체는 메서드에 의해 관리됩니다.

```
/ 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_cryptosdk_session_destroy(session);
```

## AWS Encryption SDK for C 예제

다음 예제에서는를 사용하여 데이터를 암호화하고 복호화 AWS Encryption SDK for C 하는 방법을 보여줍니다.

이 섹션의 예제에서는 2.0.x 이상 버전의 AWS Encryption SDK for C를 사용하는 방법을 보여줍니다. 이하 버전을 사용하는 예제는 GitHub의 aws-encryption-sdk-c 리포지토리 리포지토리의 릴리스 목록에서 해당하는 릴리스를 찾을 수 있습니다.

를 설치하고 빌드하면 이러한 예제 및 기타 예제 AWS Encryption SDK for C의 소스 코드가 `examples` 하위 디렉터리에 포함되고 디렉터리에 컴파일 및 빌드됩니다`build`. GitHub의 aws-encryption-sdk-c 리포지토리의 예제 하위 디렉터리에서도 해당 예제를 찾을 수 있습니다.

주제

- 문자열 암호화 및 복호화

## 문자열 암호화 및 복호화

다음 예제에서는를 사용하여 문자열 AWS Encryption SDK for C 을 암호화하고 복호화하는 방법을 보여줍니다.

이 예제는 AWS Key Management Service (AWS KMS) AWS KMS key 의를 사용하여 데이터 키를 생성하고 암호화하는 키링 AWS KMS유형인 키링을 특징으로 합니다. 이 예제에는 C++로 작성된 코드가 포함되어 있습니다. AWS Encryption SDK for C 에서는 AWS KMS 키링을 사용할 AWS KMS 때를 호출 AWS SDK for C++ 해야 합니다. 원시 AES 키링 AWS KMS, 원시 RSA 키링 또는 키링이 포함되지 않은 다중 키링과 같이와 상호 작용하지 않는 AWS KMS 키링을 사용하는 경우 AWS SDK for C++ 는 필요하지 않습니다.

생성에 대한 도움말은 AWS Key Management Service 개발자 안내서의 키 생성을 AWS KMS key참조하세요. https://docs.aws.amazon.com/kms/latest/developerguide/create-keys.html AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

전체 코드 샘플 보기: string.cpp

주제

- 문자열 암호화
- 문자열 복호화

### 문자열 암호화

이 예제의 첫 번째 부분에서는 AWS KMS 키링을 1과 함께 사용하여 일반 텍스트 문자열을 암호화 AWS KMS key 합니다.

1단계: 오류 문자열을 로드합니다.

C 또는 C++ 코드에서 `aws_cryptosdk_load_error_strings()` 메서드를 호출합니다. 이 메서드는 디버깅에 매우 유용한 오류 정보를 로드합니다.

`main` 메서드에서와 같이 한 번만 호출하면 됩니다.

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

2단계: 키링을 구성합니다.

암호화를 위한 AWS KMS 키링을 생성합니다. 이 예제의 키링은 1로 구성 AWS KMS key되지만 서로 다른 계정 AWS 리전 과 다른 계정을 AWS KMS keys포함하여 여러 로 AWS KMS 키링 AWS KMS keys 을 구성할 수 있습니다.

AWS KMS key 의 암호화 키링에서를 식별하려면 키 ARN 또는 별칭 ARN을 AWS Encryption SDK for C지정합니다. 복호화 키링에서는 키 ARN을 사용해야 합니다. 자세한 내용은 AWS KMS 키링 AWS KMS keys 에서 식별을 참조하세요.

AWS KMS 키링 AWS KMS keys 에서 식별

여러 로 키링을 생성할 때 일반 텍스트 데이터 키를 생성하고 암호화하는 데 AWS KMS key 사용되는와 동일한 일반 텍스트 데이터 키를 AWS KMS keys 암호화하는 추가의 선택적 배열을 AWS KMS keys지정합니다. 이 경우 생성기만 지정합니다 AWS KMS key.

이 코드를 실행하기 전에 예제 키 ARN을 유효한 키로 바꿉니다.

```
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);
```

3단계: 세션을 생성합니다.

할당자, 모드 열거자, 키링을 사용하여 세션을 생성합니다.

모든 세션에는 모드가 필요합니다. 암호화하려면 AWS_CRYPTOSDK_ENCRYPT, 복호화하려면 AWS_CRYPTOSDK_DECRYPT 중 하나를 선택해야 합니다. 기존 세션의 모드를 변경하려면 aws_cryptosdk_session_reset 메서드를 사용합니다.

키링으로 세션을 생성한 후, SDK가 제공하는 메서드를 사용하여 키링에 대한 참조를 릴리스할 수 있습니다. 세션은 수명 기간 동안 키링 객체에 대한 참조를 유지합니다. 세션을 삭제하면 키링 및 세션 객체에 대한 참조가 릴리스됩니다. 이 참조 카운트 기술은 메모리 누수를 방지하고 객체가 사용 중일 때 객체가 릴리스되지 않도록 도와줍니다.

```
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);
```

4단계: 암호화 컨텍스트를 설정합니다.

암호화 컨텍스트는 비밀이 아닌 임의의 추가 인증 데이터입니다. 암호화 시 암호화 컨텍스트를 제공하면 AWS Encryption SDK 는 암호화 컨텍스트를 사이퍼텍스트에 암호화 방식으로 바인딩하여 데이터를 복호화하는 데 동일한 암호화 컨텍스트가 필요합니다. 암호화 컨텍스트를 사용하는 것은 선택 사항이지만 권장되는 모범 사례입니다.

먼저 암호화 컨텍스트 문자열을 포함하는 해시 테이블을 생성합니다.

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

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

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

세션에서 암호화 컨텍스트에 대한 변경 가능한 포인터를 가져옵니다. 그런 다음 `aws_cryptosdk_enc_ctx_clone` 함수를 사용하여 암호화 컨텍스트를 세션에 복사합니다. 이 복사본은 데이터를 복호화한 후 값을 검증할 수 있도록 `my_enc_ctx`에 보관됩니다.

암호화 컨텍스트는 세션 프로세스 함수에 전달되는 파라미터가 아니라 세션의 일부입니다. 이렇게 하면 전체 메시지를 암호화하기 위해 세션 프로세스 함수를 여러 번 호출하더라도 메시지의 모든 세그먼트에 동일한 암호화 컨텍스트가 사용되도록 합니다.

```
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)
```

5단계: 문자열을 암호화합니다.

일반 텍스트 문자열을 암호화하려면 세션이 암호화 모드인 상태에서 `aws_cryptosdk_session_process_full` 메서드를 사용합니다. AWS Encryption SDK 버전

1.9.x 및 2.2.x에 도입된이 방법은 비스트리밍 암호화 및 복호화를 위해 설계되었습니다. 스트리밍 데이터를 처리하려면 aws_cryptosdk_session_process를 루프에서 호출합니다.

암호화할 때 일반 텍스트 필드는 입력 필드이고 사이퍼텍스트 필드는 출력 필드입니다. 처리가 완료되면 실제 사이퍼텍스트, 암호화된 데이터 키, 암호화 컨텍스트를 비롯한 암호화된 메시지가 ciphertext_output 필드에 포함됩니다. 지원되는 프로그래밍 언어에 AWS Encryption SDK 대해를 사용하여이 암호화된 메시지를 해독할 수 있습니다.

```
/* 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;
}
```

6단계: 세션을 정리합니다.

마지막 단계에서는 CMM 및 키링에 대한 참조를 포함하여 세션을 삭제합니다.

원하는 경우 세션을 삭제하는 대신 동일한 키링과 CMM으로 세션을 재사용하여 문자열을 복호화하거나 다른 메시지를 암호화 또는 복호화할 수 있습니다. 세션을 복호화에 사용하려면 aws_cryptosdk_session_reset 메서드를 사용하여 모드를 AWS_CRYPTOSDK_DECRYPT로 변경합니다.

### 문자열 복호화

이 예제의 두 번째 부분에서는 원본 문자열의 사이퍼텍스트가 포함된 암호화된 메시지를 복호화합니다.

1단계: 오류 문자열을 로드합니다.

C 또는 C++ 코드에서 aws_cryptosdk_load_error_strings() 메서드를 호출합니다. 이 메서드는 디버깅에 매우 유용한 오류 정보를 로드합니다.

main 메서드에서와 같이 한 번만 호출하면 됩니다.

```
/* Load error strings for debugging */
```

```
aws_cryptosdk_load_error_strings();
```

2단계: 키링을 구성합니다.

에서 데이터를 복호화하면 암호화 API가 반환한 암호화된 메시지를 AWS KMS전달합니다. Decrypt API는를 입력 AWS KMS key 으로 사용하지 않습니다. 대신 동일한를 AWS KMS 사용하여 암호화 AWS KMS key 에 사용한 사이퍼텍스트를 복호화합니다. 그러나를 AWS Encryption SDK 사용하면 암호화 및 복호화 AWS KMS keys 시를 사용하여 AWS KMS 키링을 지정할 수 있습니다.

복호화 시 암호화된 메시지를 복호화 AWS KMS keys 하는 데 사용할 로만 키링을 구성할 수 있습니다. 예를 들어 조직의 특정 역할에서 AWS KMS key 사용하는 만 사용하여 키링을 생성할 수 있습니다. 는 AWS Encryption SDK 복호화 키링에 나타나지 않는 AWS KMS key 한를 사용하지 않습니다. 제공된 키링 AWS KMS keys 에서를 사용하여 암호화된 데이터 키를 복호화할 수 없는 경우 키링 AWS KMS keys 의가 데이터 키를 암호화하는 데 사용되지 않았거나 호출자가 키링 AWS KMS keys 의를 사용하여 복호화할 권한이 없기 때문에 복호화 호출이 실패합니다.

복호화 키링에 AWS KMS key 대해를 지정할 때는 해당 키 ARN을 사용해야 합니다. 별칭 ARN은 암호화 키링에서만 허용됩니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요AWS KMS 키링 AWS KMS keys 에서 식별.

이 예제에서는 문자열을 암호화하는 데 AWS KMS key 사용된 것과 동일한 로 구성된 키링을 지정합니다. 이 코드를 실행하기 전에 예제 키 ARN을 유효한 키로 바꿉니다.

```
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);
```

3단계: 세션을 생성합니다.

할당자와 키링을 사용하여 세션을 생성합니다. 복호화를 위한 세션을 구성하려면 AWS_CRYPTOSDK_DECRYPT 모드를 사용하여 세션을 구성합니다.

키링으로 세션을 생성한 후, SDK가 제공하는 메서드를 사용하여 키링에 대한 참조를 릴리스할 수 있습니다. 세션은 수명 동안 키링 객체에 대한 참조를 유지하며, 세션과 키링은 세션을 삭제할 때 릴리스됩니다. 이 참조 카운트 기술은 메모리 누수를 방지하고 객체가 사용 중일 때 객체가 릴리스되지 않도록 도와줍니다.

```
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);
```

4단계: 문자열을 복호화합니다.

문자열을 복호화하려면 복호화를 위해 구성된 세션에서 `aws_cryptosdk_session_process_full` 메서드를 사용합니다. 이 메서드는 AWS Encryption SDK 버전 1.9.x 및 2.2.x에 도입되었으며, 비스트리밍 암호화 및 복호화를 위해 설계되었습니다. 스트리밍 데이터를 처리하려면 `aws_cryptosdk_session_process`를 루프에서 호출합니다.

복호화 시 사이퍼텍스트 필드는 입력 필드이고 일반 텍스트 필드는 출력 필드입니다. `ciphertext_input` 필드에는 암호화 메서드가 반환한 암호화된 메시지가 있습니다. 처리가 완료되면 `plaintext_output` 필드에 일반 텍스트(복호화된) 문자열이 포함됩니다.

```
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;
}
```

5단계: 암호화 컨텍스트를 확인합니다.

메시지를 복호화하는 데 사용된 실제 암호화 컨텍스트에 메시지를 암호화할 때 제공한 암호화 컨텍스트가 포함되어 있는지 확인하세요. 암호화 구성 요소 관리자(CMM)가 메시지를 암호화하기 전에 제공된 암호화 컨텍스트에 페어를 추가할 수 있으므로 실제 암호화 컨텍스트에 추가 페어가 포함될 수 있습니다.

에서는 암호화 컨텍스트가 SDK가 반환하는 암호화된 메시지에 포함되므로 복호화할 때 암호화 컨텍스트를 제공할 필요가 AWS Encryption SDK for C없습니다. 하지만 복호화 함수는 일반 텍스트 메시지를 반환하기 전에 제공된 암호화 컨텍스트의 모든 페어가 메시지를 복호화하는 데 사용된 암호화 컨텍스트에 나타나는지 확인해야 합니다.

먼저 세션의 해시 테이블에 대한 읽기 전용 포인터를 가져옵니다. 이 해시 테이블에는 메시지를 복호화하는 데 사용된 암호화 컨텍스트가 포함되어 있습니다.

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

그런 다음 암호화할 때 복사한 `my_enc_ctx` 해시 테이블에서 암호화 컨텍스트를 반복합니다. 암호화에 사용된 `my_enc_ctx` 해시 테이블의 각 페어가 복호화에 사용된 `session_enc_ctx` 해시 테이블에 나타나는지 확인합니다. 누락된 키가 있거나 해당 키의 값이 다른 경우 처리를 중지하고 오류 메시지를 작성합니다.

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

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

6단계: 세션을 정리합니다.

암호화 컨텍스트를 확인한 후 세션을 삭제하거나 재사용할 수 있습니다. 세션을 재구성해야 하는 경우 `aws_cryptosdk_session_reset` 메서드를 사용합니다.

```
aws_cryptosdk_session_destroy(session);
```

# AWS Encryption SDK .NET용

AWS Encryption SDK for .NET은 C# 및 기타 .NET 프로그래밍 언어로 애플리케이션을 작성하는 개발자를 위한 클라이언트 측 암호화 라이브러리입니다. 이는 Windows, macOS, Linux에서 지원됩니다.

> **Note**
>
> AWS Encryption SDK for .NET 버전 4.0.0은 AWS Encryption SDK 메시지 사양을 벗어납니다. 따라서 버전 4.0.0으로 암호화된 메시지는 for .NET 버전 4.0.0 이상으로만 복호화할 수 AWS Encryption SDK 있습니다. 다른 프로그래밍 언어 구현으로는 복호화할 수 없습니다.
> AWS Encryption SDK for .NET 버전 4.0.1은 AWS Encryption SDK 메시지 사양에 따라 메시지를 작성하고 다른 프로그래밍 언어 구현과 상호 운용할 수 있습니다. 기본적으로 버전 4.0.1은 버전 4.0.0으로 암호화된 메시지를 읽을 수 있습니다. 그러나 버전 4.0.0으로 암호화된 메시지를 복호화하지 않으려면 클라이언트가 이러한 메시지를 읽지 못하도록 `NetV4_0_0_RetryPolicy` 속성을 지정합니다. 자세한 내용은 GitHub의 aws-encryption-sdk 리포지토리에서 v4.0.1 릴리스 정보를 참조하세요.

AWS Encryption SDK for .NET은 다음과 AWS Encryption SDK 같은 방식으로의 다른 프로그래밍 언어 구현과 다릅니다.

- 데이터 키 캐싱이 지원되지 않음

  > **Note**
  >
  > for .NET 버전 AWS Encryption SDK 4.x는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 지원합니다.

- 스트리밍 데이터가 지원되지 않음
- AWS Encryption SDK for .NET에서 로깅 또는 스택 추적이 없음
- 가 필요합니다. AWS SDK for .NET

AWS Encryption SDK for .NET에는의 다른 언어 구현 버전 2.0.x 이상에 도입된 모든 보안 기능이 포함되어 있습니다 AWS Encryption SDK. 그러나 AWS Encryption SDK for .NET을 사용하여의 다른 언어 구현인 2.0.x 이전 버전으로 암호화된 데이터를 해독 AWS Encryption SDK하는 경우 커밋 정책을 조정해야 할 수 있습니다. 자세한 내용은 커밋 정책 설정 방법을 참조하세요.

AWS Encryption SDK for .NET은 사양을 작성하는 공식 확인 언어인 Dafny AWS Encryption SDK 의 제품이며, 이를 구현할 코드와 이를 테스트하기 위한 증거입니다. 그 결과, 기능적 정확성을 보장하는 프레임워크에서 AWS Encryption SDK 의 기능을 구현하는 라이브러리가 탄생했습니다.

자세히 알아보기

- 대체 알고리즘 제품군 지정 AWS Encryption SDK, 암호화된 데이터 키 제한, AWS KMS 다중 리전 키 사용 등에서 옵션을 구성하는 방법을 보여주는 예제는 섹션을 참조하세요구성 AWS Encryption SDK.
- AWS Encryption SDK for .NET을 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-encryption-sdk 리포지토리 `aws-encryption-sdk-net` 디렉터리를 참조하세요.

주제



## AWS Encryption SDK for .NET 설치

AWS Encryption SDK for .NET은 NuGet에서 `AWS.Cryptography.EncryptionSDK` 패키지로 사용할 수 있습니다. AWS Encryption SDK for .NET 설치 및 빌드에 대한 자세한 내용은 `aws-encryption-sdk-net`리포지토리의 README.md 파일을 참조하세요.

버전 3.x

AWS Encryption SDK for .NET 버전 3.x는 Windows에서만 .NET Framework 4.5.2~4.8을 지원합니다. 지원되는 모든 운영 체제에서 .NET Core 3.0 이상 및 .NET 5.0 이상을 지원합니다.

버전 4.x

AWS Encryption SDK for .NET 버전 4.x는 .NET 6.0 및 .NET Framework net48 이상을 지원합니다. 버전 4.x에는 .NET v3용 AWS SDK가 필요합니다.

AWS Encryption SDK for .NET에는 AWS Key Management Service (AWS KMS) 키를 사용하지 않더라도 SDK for .NET 가 필요합니다. NuGet 패키지와 함께 설치됩니다. 그러나 AWS KMS 키를 사용하지 않는 한 AWS Encryption SDK for .NET에는 AWS 계정자격 AWS 증명 또는 AWS 서비스와의 상호 작용이 필요하지 않습니다. 필요한 경우 AWS 계정 설정에 대한 도움말은 섹션을 참조하세요와 AWS Encryption SDK 함께 사용 AWS KMS.

## AWS Encryption SDK for .NET 디버깅

AWS Encryption SDK for .NET은 로그를 생성하지 않습니다. AWS Encryption SDK for .NET의 예외는 예외 메시지를 생성하지만 스택 추적은 생성하지 않습니다.

디버깅에 도움이 되도록 SDK for .NET에서 로그인을 활성화해야 합니다. 의 로그 및 오류 메시지는에서 발생하는 오류를 AWS Encryption SDK for .NET의 오류 SDK for .NET 와 구별하는 데 도움이 될 SDK for .NET 수 있습니다. SDK for .NET 로깅에 대한 도움말은 AWS SDK for .NET 개발자 안내서의 AWSLogging을 참조하세요. (이 주제를 보려면 .NET Framework 콘텐츠를 열어서 보기 섹션을 확장하세요.)

## AWS Encryption SDK for .NET 예제

다음 예제에서는 for AWS Encryption SDK .NET을 사용하여 프로그래밍할 때 사용하는 기본 코딩 패턴을 보여줍니다. 특히 AWS Encryption SDK 및 재료 공급자 라이브러리를 인스턴스화합니다. 그런 다음 각 메서드를 호출하기 전에 메서드의 입력을 정의하는 객체를 인스턴스화합니다. 이는 SDK for .NET에서 사용하는 코딩 패턴과 매우 비슷합니다.

대체 알고리즘 제품군 지정 AWS Encryption SDK, 암호화된 데이터 키 제한, AWS KMS 다중 리전 키 사용 등에서 옵션을 구성하는 방법을 보여주는 예제는 섹션을 참조하세요구성 AWS Encryption SDK.

for .NET을 사용한 프로그래밍에 AWS Encryption SDK 대한 자세한 예제는 GitHub의 `aws-encryption-sdk`리포지토리 `aws-encryption-sdk-net` 디렉터리에 있는 예제를 참조하세요.

### AWS Encryption SDK for .NET의 데이터 암호화

이 예제에서는 데이터를 암호화하는 기본 패턴을 보여줍니다. 하나의 AWS KMS 래핑 키로 보호되는 데이터 키로 작은 파일을 암호화합니다.

1단계: AWS Encryption SDK 및 재료 공급자 라이브러리를 인스턴스화합니다.

먼저 AWS Encryption SDK 및 재료 공급자 라이브러리를 인스턴스화합니다. 의 메서드를 사용하여 데이터를 암호화하고 해독 AWS Encryption SDK 합니다. 구성 요소 공급자 라이브러리의 메서드를 사용하여, 데이터를 보호하는 키를 지정하는 키링을 만들 수 있습니다.

AWS Encryption SDK 및 재료 공급자 라이브러리를 인스턴스화하는 방법은 for . AWS Encryption SDK NET 버전 3.x와 4.x 간에 다릅니다. 다음 단계는 for . AWS Encryption SDK NET 버전 3.x와 4.x에서 모두 동일합니다.

Version 3.x

```
// 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());
```

2단계: 키링에 대한 입력 객체를 생성합니다.

키링을 만드는 각 메서드에는 해당하는 입력 객체 클래스가 있습니다. 예를 들어, `CreateAwsKmsKeyring()` 메서드의 입력 객체를 만들려면 `CreateAwsKmsKeyringInput` 클래스의 인스턴스를 생성합니다.

이 키링의 입력에 생성기 키를 지정하지 않더라도 `KmsKeyId` 파라미터로 지정된 단일 KMS 키는 생성기 키입니다. 데이터를 암호화하는 데이터 키를 생성하고 암호화합니다.

이 입력 객체에는 KMS 키 AWS 리전 의에 대한 AWS KMS 클라이언트가 필요합니다. AWS KMS 클라이언트를 생성하려면에서 `AmazonKeyManagementServiceClient` 클래스를 인스턴스화합니다 SDK for .NET. 파라미터 없이 `AmazonKeyManagementServiceClient()` 생성자를 호출하면 기본값으로 클라이언트가 만들어집니다.

AWS Encryption SDK for .NET을 사용한 암호화에 사용되는 AWS KMS 키링에서 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN을 사용하여 KMS 키를 식별할 수 있습니다. 복호화에 사용되는 AWS KMS 키링에서는 키 ARN을 사용하여 각 KMS 키를 식별해야 합니다. 복호화에 암호화 키링을 재사용하려는 경우 모든 KMS 키에 키 ARN 식별자를 사용합니다.

```
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
};
```

3단계: 키링을 생성합니다.

키링을 생성하려면 키링 입력 객체를 사용하여 키링 메서드를 호출합니다. 이 예제에서는 KMS 키를 하나만 사용하는 `CreateAwsKmsKeyring()` 메서드를 사용합니다.

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

4단계: 암호화 컨텍스트를 정의합니다.

암호화 컨텍스트는에서 암호화 작업의 선택적이지만 강력히 권장되는 요소입니다 AWS Encryption SDK. 비밀이 아닌 키-값 페어를 하나 이상 정의할 수 있습니다.

> **Note**
>
> AWS Encryption SDK for .NET 버전 4.x에서는 필요한 암호화 컨텍스트 CMM을 사용하여 모든 암호화 요청에 암호화 컨텍스트를 요구할 수 있습니다.

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

5단계: 암호화에 대한 입력 객체를 생성합니다.

`Encrypt()` 메서드를 호출하기 전에 `EncryptInput` 클래스의 인스턴스를 생성합니다.

```
string plaintext = File.ReadAllText("C:\\Documents\\CryptoTest\\TestFile.txt");

// Define the encrypt input
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
};
```

6단계: 일반 텍스트를 암호화합니다.

의 `Encrypt()` 메 AWS Encryption SDK 서드를 사용하여 정의한 키링을 사용하여 일반 텍스트를 암호화합니다.

`Encrypt()` 메서드가 반환하는 `EncryptOutput`에는 암호화된 메시지(`Ciphertext`), 암호화 컨텍스트 및 알고리즘 제품군을 가져오는 메서드가 있습니다.

```
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

7단계: 암호화된 메시지를 가져옵니다.

AWS Encryption SDK for .NET의 `Decrypt()` 메서드는 `EncryptOutput` 인스턴스의 `Ciphertext` 멤버를 가져옵니다.

`EncryptOutput` 객체의 `Ciphertext` 멤버는 암호화 컨텍스트를 비롯해 암호화된 데이터, 암호화된 데이터 키 및 메타데이터를 포함하는 이동 가능 객체인 암호화된 메시지입니다. 암호화된 메시지를 장기간 안전하게 저장하거나 `Decrypt()` 메서드에 제출하여 일반 텍스트를 복구할 수 있습니다.

```
var encryptedMessage = encryptOutput.Ciphertext;
```

## AWS Encryption SDK for .NET에서 엄격 모드에서 복호화

모범 사례에서는 데이터를 복호화하는 데 사용할 키를 지정하는 것이 좋으며, 이러한 옵션을 엄격한 모드라고 합니다. 는 키링에 지정한 KMS 키만 AWS Encryption SDK 사용하여 사이퍼텍스트를 복호화합니다. 복호화 키링의 키에는 데이터를 암호화한 키가 하나 이상 포함되어야 합니다.

이 예제에서는 AWS Encryption SDK for .NET을 사용하여 엄격한 모드에서 복호화하는 기본 패턴을 보여줍니다.

1단계: AWS Encryption SDK 및 재료 공급자 라이브러리를 인스턴스화합니다.

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

2단계: 키링에 대한 입력 객체를 생성합니다.

키링 메서드의 파라미터를 지정하려면 입력 객체를 생성합니다. AWS Encryption SDK for .NET의 각 키링 메서드에는 해당 입력 객체가 있습니다. 이 예제에서는 `CreateAwsKmsKeyring()` 메서드를 사용하여 키링을 만들기 때문에 입력에 대한 `CreateAwsKmsKeyringInput` 클래스를 인스턴스화합니다.

복호화 키링에서는 키 ARN을 사용하여 KMS 키를 식별해야 합니다.

```
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
};
```

3단계: 키링을 생성합니다.

이 예제에서는 복호화 키링을 생성하기 위해 `CreateAwsKmsKeyring()` 메서드와 키링 입력 객체를 사용합니다.

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

4단계: 복호화에 대한 입력 객체를 생성합니다.

`Decrypt()` 메서드의 입력 객체를 만들려면 `DecryptInput` 클래스를 인스턴스화합니다.

`DecryptInput()` 생성자의 `Ciphertext` 파라미터는 `Encrypt()` 메서드가 반환한 `EncryptOutput` 객체의 `Ciphertext` 멤버를 가져옵니다. `Ciphertext` 속성은 암호화된 메시지를 나타내며, 여기에는 AWS Encryption SDK 가 메시지를 복호화하는 데 필요한 암호화된 데이터, 암호화된 데이터 키 및 메타데이터가 포함됩니다.

AWS Encryption SDK for .NET 버전 4.x에서는 선택적 `EncryptionContext` 파라미터를 사용하여 `Decrypt()` 메서드에서 암호화 컨텍스트를 지정할 수 있습니다.

`EncryptionContext` 파라미터를 사용하여 암호화에 사용된 암호화 컨텍스트가 사이퍼텍스트를 복호화하는 데 사용되는 암호화 컨텍스트에 포함되어 있는지 확인합니다. 는 기본 알고리즘 제품군과 같이 서명과 함께 알고리즘 제품군을 사용하는 경우 디지털 서명을 포함하여 암호화 컨텍스트에 페어를 AWS Encryption SDK 추가합니다.

```
var encryptedMessage = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = encryptedMessage,
    Keyring = keyring,
    EncryptionContext = encryptionContext // OPTIONAL
};
```

5단계: 사이퍼텍스트를 복호화합니다.

```
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

6단계: 암호화 컨텍스트 - 버전 3.x를 확인합니다.

AWS Encryption SDK for .NET 버전 3.x의 `Decrypt()` 메서드는 암호화 컨텍스트를 사용하지 않습니다. 암호화된 메시지의 메타데이터에서 암호화 컨텍스트 값을 가져옵니다. 하지만 일반 텍스트를 반환하거나 사용하기 전에, 사이퍼텍스트를 복호화하는 데 사용된 암호화 컨텍스트에 암호화 시 제공한 암호화 컨텍스트가 포함되어 있는지 확인하는 것이 모범 사례입니다.

암호화에 사용된 암호화 컨텍스트가 사이퍼텍스트를 복호화하는 데 사용된 암호화 컨텍스트에 포함되어 있는지 확인합니다. 는 기본 알고리즘 제품군과 같이 서명과 함께 알고리즘 제품군을 사용하는 경우 디지털 서명을 포함하여 암호화 컨텍스트에 페어를 AWS Encryption SDK 추가합니다.

```
// 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 .NET에서 검색 키링을 사용하여 복호화

복호화를 위한 KMS 키를 지정하는 대신, KMS 키를 지정하지 않는 키링인 AWS KMS 검색 키링을 제공할 수 있습니다. 호출자에게 키에 대한 AWS Encryption SDK 복호화 권한이 있는 경우 검색 키링을 사용하면가 데이터를 암호화한 KMS 키를 사용하여 데이터를 복호화할 수 있습니다. 모범 사례를 위해 지정된 파티션 AWS 계정 의 KMS 키로 사용할 수 있는 KMS 키를 제한하는 검색 필터를 추가합니다.

AWS Encryption SDK for .NET은 클라이언트가 필요한 AWS KMS 기본 검색 키링과 하나 이상의를 지정해야 하는 검색 다중 키링을 제공합니다 AWS 리전. 클라이언트와 리전은 암호화된 메시지를 복호화하는 데 사용할 수 있는 KMS 키를 제한합니다. 두 키링의 입력 객체는 권장 검색 필터를 사용합니다.

다음 예제에서는 AWS KMS 검색 키링 및 검색 필터를 사용하여 데이터를 복호화하는 패턴을 보여줍니다.

1단계: AWS Encryption SDK 및 재료 공급자 라이브러리를 인스턴스화합니다.

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

2단계: 키링에 대한 입력 객체를 생성합니다.

키링 메서드의 파라미터를 지정하려면 입력 객체를 생성합니다. AWS Encryption SDK for .NET의 각 키링 메서드에는 해당 입력 객체가 있습니다. 이 예제에서는 `CreateAwsKmsDiscoveryKeyring()` 메서드를 사용하여 키링을 만들기 때문에 입력에 대한 `CreateAwsKmsDiscoveryKeyringInput` 클래스를 인스턴스화합니다.

```
List<string> accounts = new List<string> { "111122223333" };

var discoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = accounts,
        Partition = "aws"
    }
};
```

3단계: 키링을 생성합니다.

이 예제에서는 복호화 키링을 생성하기 위해 `CreateAwsKmsDiscoveryKeyring()` 메서드와 키링 입력 객체를 사용합니다.

```
var discoveryKeyring =
 materialProviders.CreateAwsKmsDiscoveryKeyring(discoveryKeyringInput);
```

4단계: 복호화에 대한 입력 객체를 생성합니다.

`Decrypt()` 메서드의 입력 객체를 만들려면 `DecryptInput` 클래스를 인스턴스화합니다. `Ciphertext` 파라미터의 값은 `Encrypt()` 메서드가 반환하는 `EncryptOutput` 객체의 `Ciphertext` 멤버입니다.

AWS Encryption SDK for .NET 버전 4.x에서는 선택적 `EncryptionContext` 파라미터를 사용하여 `Decrypt()` 메서드에서 암호화 컨텍스트를 지정할 수 있습니다.

`EncryptionContext` 파라미터를 사용하여 암호화에 사용된 암호화 컨텍스트가 사이퍼텍스트를 복호화하는 데 사용되는 암호화 컨텍스트에 포함되어 있는지 확인합니다. 는 기본 알고리즘 제품군과 같이 서명과 함께 알고리즘 제품군을 사용하는 경우 디지털 서명을 포함하여 암호화 컨텍스트에 페어를 AWS Encryption SDK 추가합니다.

```
var ciphertext = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = discoveryKeyring,
    EncryptionContext = encryptionContext // OPTIONAL

};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

5단계: 암호화 컨텍스트 - 버전 3.x를 확인합니다.

AWS Encryption SDK for .NET 버전 3.x의 `Decrypt()` 메서드는에서 암호화 컨텍스트를 가져오지 않습니다`Decrypt()`. 암호화된 메시지의 메타데이터에서 암호화 컨텍스트 값을 가져옵니다. 하지만 일반 텍스트를 반환하거나 사용하기 전에, 사이퍼텍스트를 복호화하는 데 사용된 암호화 컨텍스트에 암호화 시 제공한 암호화 컨텍스트가 포함되어 있는지 확인하는 것이 모범 사례입니다.

암호화에 사용된 암호화 컨텍스트가 사이퍼텍스트를 복호화하는 데 사용된 암호화 컨텍스트에 포함되어 있는지 확인합니다. 는 기본 알고리즘 제품군과 같이 서명과 함께 알고리즘 제품군을 사용하는 경우 디지털 서명을 포함하여 암호화 컨텍스트에 페어를 AWS Encryption SDK 추가합니다.

```
// 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 Go용

이 주제에서는 AWS Encryption SDK for Go를 설치하고 사용하는 방법을 설명합니다. AWS Encryption SDK for Go를 사용한 프로그래밍에 대한 자세한 내용은 GitHub에서 aws-encryption-sdk 리포지토리의 go 디렉터리를 참조하세요.

AWS Encryption SDK for Go는 다음과 같은 점 AWS Encryption SDK 에서의 다른 프로그래밍 언어 구현과 다릅니다.

- 데이터 키 캐싱은 지원되지 않습니다. 그러나 AWS Encryption SDK for Go는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 지원합니다.
- 스트리밍 데이터가 지원되지 않음

AWS Encryption SDK for Go에는의 다른 언어 구현 버전 2.0.x 이상에 도입된 모든 보안 기능이 포함되어 있습니다 AWS Encryption SDK. 그러나 AWS Encryption SDK for Go를 사용하여의 다른 언어 구현인 2.0.x 이전 버전으로 암호화된 데이터를 복호화 AWS Encryption SDK하는 경우 커밋 정책을 조정해야 할 수 있습니다. 자세한 내용은 커밋 정책 설정 방법을 참조하세요.

AWS Encryption SDK for Go는 사양을 작성하는 공식 확인 언어인 Dafny AWS Encryption SDK 의 제품이며, 이를 구현할 코드와 이를 테스트하기 위한 증거입니다. 그 결과, 기능적 정확성을 보장하는 프레임워크에서 AWS Encryption SDK 의 기능을 구현하는 라이브러리가 탄생했습니다.

자세히 알아보기

- 대체 알고리즘 제품군 지정 AWS Encryption SDK, 암호화된 데이터 키 제한, AWS KMS 다중 리전 키 사용 등에서 옵션을 구성하는 방법을 보여주는 예제는 섹션을 참조하세요구성 AWS Encryption SDK.
- AWS Encryption SDK for Go를 구성하고 사용하는 방법을 보여주는 예제는 GitHub의 aws-encryption-sdk 리포지토리에 있는 Go 예제를 참조하세요.

주제



## 사전 조건

AWS Encryption SDK for Go를 설치하기 전에 다음 사전 조건이 있는지 확인합니다.

Go의 지원되는 버전

Go용 에는 AWS Encryption SDK Go 1.23 이상이 필요합니다.

Go 다운로드 및 설치에 대한 자세한 내용은 Go installation을 참조하세요.

## 설치

최신 버전의 AWS Encryption SDK for Go를 설치합니다. for Go 설치 및 빌드 AWS Encryption SDK 에 대한 자세한 내용은 GitHub에서 aws-encryption-sdk 리포지토리의 go 디렉터리에 있는 README.md 참조하십시오.

최신 버전 설치

- AWS Encryption SDK for Go 설치

```
go get github.com/aws/aws-encryption-sdk/releases/go/encryption-sdk@latest
```

- 암호화 자료 공급자 라이브러리(MPL) 설치

```
go get github.com/aws/aws-cryptographic-material-providers-library/releases/go/mpl
```

# AWS Encryption SDK for Java

이 주제에서는 AWS Encryption SDK for Java를 설치 및 사용하는 방법을 설명합니다. 를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-encryption-sdk-java 리포지토리를 AWS Encryption SDK for Java참조하세요. API 설명서를 보려면 AWS Encryption SDK for Java용 Javadoc을 참조하세요.

주제

- 사전 조건
- 설치
- AWS Encryption SDK for Java 예제

## 사전 조건

를 설치하기 전에 다음 사전 조건이 있는지 AWS Encryption SDK for Java확인합니다.

Java 개발 환경

Java 8 이상이 필요합니다. Oracle 웹 사이트에서 Java SE 다운로드로 이동한 다음 Java SE Development Kit(JDK)를 다운로드하여 설치합니다.

Oracle JDK를 사용하는 경우 Java Cryptography Extension(JCE) Unlimited Strength Jurisdiction Policy File도 다운로드하여 설치해야 합니다.

Bouncy Castle

에는 Bouncy Castle이 AWS Encryption SDK for Java 필요합니다.

- AWS Encryption SDK for Java 버전 1.6.1 이상에서는 Bouncy Castle을 사용하여 암호화 객체를 직렬화하고 역직렬화합니다. 이 요구 사항을 충족하기 위해 Bouncy Castle 또는 Bouncy Castle FIPS를 사용할 수 있습니다. Bouncy Castle FIPS 설치 및 구성에 대한 도움말은 BC FIPS 설명서, 특히 사용 설명서 및 보안 정책 PDF를 참조하세요.
- 이전 버전의 에서는 Java용 Bouncy Castle의 암호화 API를 AWS Encryption SDK for Java 사용합니다. 이 요구 사항은 비 FIPS Bouncy Castle만 만족합니다.

Bouncy Castle이 없는 경우 Java용 Bouncy Castle 다운로드로 이동하여 JDK에 해당하는 공급자 파일을 다운로드합니다. Apache Maven을 사용하여 표준 Bouncy Castle 공급자(bcprov-ext-jdk15on)용 아티팩트 또는 Bouncy Castle FIPS(bc-fips)용 아티팩트를 가져올 수 있습니다.

AWS SDK for Java

의 버전 3.x에는 AWS KMS 키링을 사용하지 AWS SDK for Java 2.x않더라도가 AWS Encryption SDK for Java 필요합니다.

버전 2.x 이하 AWS Encryption SDK for Java 에서는가 필요하지 않습니다 AWS SDK for Java. 그러나 AWS Key Management Service (AWS KMS)를 마스터 키 공급자로 사용하려면 AWS SDK for Java 가 필요합니다. AWS Encryption SDK for Java 버전 2.4.0부터는 1.x 및 2.x용 AWS SDK for Java. AWS Encryption SDK code 버전 AWS SDK for Java 1.x 및 2.x를 모두 AWS Encryption SDK for Java 지원하며 상호 운용할 수 있습니다. 예를 들어, AWS SDK for Java 가1.x를 지원하는 AWS Encryption SDK 코드로 데이터를 암호화하고가 지원하는 코드 AWS SDK for Java 2.x (또는 그 반대)를 사용하여 복호화할 수 있습니다. 2.4.0 AWS Encryption SDK for Java 이전 버전의는 AWS SDK for Java 1.x만 지원합니다. 버전 업데이트에 대한 자세한 내용은 섹션을 AWS Encryption SDK참조하세요마이그레이션 AWS Encryption SDK.

AWS Encryption SDK for Java 코드를 AWS SDK for Java 1.x에서 로 업데이트할 때 `AWSKMS` 인터페이스 in AWS SDK for Java 1.x에 대한 참조를 `KmsClient` 인터페이스 in에 대한 참조로 AWS SDK for Java 2.x바꿉니다 AWS SDK for Java 2.x. AWS Encryption SDK for Java 는

KmsAsyncClient 인터페이스를 지원하지 않습니다. 또한 kms 네임스페이스 대신 kmssdkv2 네임스페이스의 AWS KMS관련 객체를 사용하도록 코드를 업데이트하세요.

를 설치하려면 Apache Maven을 AWS SDK for Java사용합니다.

- 전체 AWS SDK for Java를 종속성으로 가져오려면 pom.xml 파일에 선언하세요.
- AWS SDK for Java 1.x의 AWS KMS 모듈에 대해서만 종속성을 생성하려면 특정 모듈 지정 지침을 따르고를 artifactId로 설정합니다aws-java-sdk-kms.
- AWS SDK for Java 2.x의 AWS KMS 모듈에 대해서만 종속성을 생성하려면 특정 모듈 지정 지침을 따르세요. groupId를 software.amazon.awssdk로, artifactId를 kms로 설정합니다.

자세한 내용은 AWS SDK for Java 2.x 개발자 안내서의 AWS SDK for Java 1.x와 2.x의 차이점을 참조하세요.

AWS Encryption SDK 개발자 안내서의 Java 예제에서는를 사용합니다 AWS SDK for Java 2.x.

## 설치

AWS Encryption SDK for Java의 최신 버전을 설치합니다.

> **Note**
>
> 2.0.0 AWS Encryption SDK for Java 이전의 모든 버전은 end-of-support 단계에 있습니다. 코드나 데이터를 변경하지 않고 버전 2.0.x 이상에서 AWS Encryption SDK for Java 의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.0.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 1.7.x 이하 버전에서 2.0.x 이상 버전으로 업데이트하려면 먼저 AWS Encryption SDK의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.

다음과 같은 AWS Encryption SDK for Java 방법으로를 설치할 수 있습니다.

직접

를 설치하려면 aws-encryption-sdk-java GitHub 리포지토리를 AWS Encryption SDK for Java복제하거나 다운로드합니다.

Apache Maven 사용

AWS Encryption SDK for Java 는 다음 종속성 정의와 함께 Apache Maven을 통해 사용할 수 있습니다.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>3.0.0</version>
</dependency>
```

SDK를 설치한 후에는 이 가이드의 예제 Java 코드와 GitHub의 Javadoc을 살펴보는 것부터 시작하세요.

## AWS Encryption SDK for Java 예제

다음 예제에서는를 사용하여 데이터를 암호화하고 복호화 AWS Encryption SDK for Java 하는 방법을 보여줍니다. 이 예제에서는 버전 3.x 이상을 사용하는 방법을 보여줍니다 AWS Encryption SDK for Java. 의 버전 3.x에는가 AWS Encryption SDK for Java 필요합니다 AWS SDK for Java 2.x. 의 버전 3.x는 마스터 키 공급자를 키링으로 AWS Encryption SDK for Java 대체합니다. ??? 이하 버전을 사용하는 예제의 경우 GitHub의 aws-encryption-sdk-java 리포지토리의 릴리스 목록에서 해당하는 릴리스를 찾을 수 있습니다.

주제

- 문자열 암호화 및 복호화
- 바이트 스트림 암호화 및 복호화
- 다중 키 링을 사용하여 바이트 스트림 암호화 및 복호화

### 문자열 암호화 및 복호화

다음 예제에서는의 버전 3.x AWS Encryption SDK for Java 를 사용하여 문자열을 암호화하고 복호화하는 방법을 보여줍니다. 문자열을 사용하기 전에 바이트 배열로 변환하세요.

이 예제에서는 AWS KMS 키링을 사용합니다. AWS KMS 키링으로 암호화할 때 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN을 사용하여 KMS 키를 식별할 수 있습니다. 복호화할 때 키 ARN을 사용하여 KMS 키를 식별해야 합니다.

`encryptData()` 메서드를 호출하면 사이퍼텍스트, 암호화된 데이터 키 및 암호화 컨텍스트를 포함하는 암호화된 메시지(`CryptoResult`)가 반환됩니다. `CryptoResult` 객체에서 `getResult`를 호출하면 `decryptData()` 메서드에 전달할 수 있는 암호화된 메시지의 base-64 인코딩 문자열 버전이 반환됩니다.

마찬가지로 `decryptData()`를 호출할 때 반환되는 `CryptoResult` 객체에는 일반 텍스트 메시지와 AWS KMS key ID가 포함됩니다. 애플리케이션이 일반 텍스트를 반환하기 전에 암호화된 메시지의 AWS KMS key ID와 암호화 컨텍스트가 예상한 것인지 확인합니다.

```
// 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.
 *
 * <p>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
    assert Arrays.equals(decryptResult.getResult(), EXAMPLE_DATA);
  }
}
```

## 바이트 스트림 암호화 및 복호화

다음 예제에서는를 사용하여 바이트 스트림을 암호화하고 복호화 AWS Encryption SDK 하는 방법을 보여줍니다.

이 예제에서는 원시 AES 키링을 사용합니다.

암호화할 때 이 예제에서는 `AwsCrypto.builder() .withEncryptionAlgorithm()` 메서드를 사용하여 디지털 서명이 없는 알고리즘 제품군을 지정합니다.
이 예제에서는 복호화할 때 사이퍼텍스트에 서명이 없는지 확인하기 위해 `createUnsignedMessageDecryptingStream()` 메서드를 사용합니다. 디지털 서명이 있는 사이퍼텍스트가 발생하면 `createUnsignedMessageDecryptingStream()` 메서드가 실패합니다.

디지털 서명이 포함된 기본 알고리즘 제품군으로 암호화하는 경우 다음 예제와 같이 `createDecryptingStream()` 메서드를 대신 사용하세요.

```
// 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;


/**
 * <p>
 * Encrypts and then decrypts a file under a random key.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 *
 * <p>
 * This program demonstrates using a standard Java {@link SecretKey} object as a {@link
 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("Exampl
 {
            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");
    }
}
```

## 다중 키 링을 사용하여 바이트 스트림 암호화 및 복호화

다음 예제에서는 다중 키 링과 AWS Encryption SDK 함께를 사용하는 방법을 보여줍니다. 다중 키링을 사용하여 데이터를 암호화하는 경우 해당 키링의 모든 래핑 키로 해당 데이터를 복호화할 수 있습니다. 이 예제에서는 AWS KMS 키링과 Raw RSA 키링을 하위 키링으로 사용합니다.

이 예제는 디지털 서명이 포함된 기본 알고리즘 제품군을 사용하여 암호화합니다. 스트리밍할 때는 무결성 검사 후 디지털 서명을 확인하기 전에 일반 텍스트를 AWS Encryption SDK 릴리스합니다. 서명이 확인될 때까지 일반 텍스트를 사용하지 않도록 이 예제에서는 일반 텍스트를 버퍼링하고 복호화 및 확인이 완료될 때만 디스크에 씁니다.

```
// 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.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>
 * 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>Name of file containing plaintext data to encrypt
 * </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.
 * <p>
 * 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.
 * <p>
 * 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,
        // 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);
```

```
        // 4. Create the multi-keyring.
        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);

        // 4. Create the multi-keyring.
        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");
        final FileOutputStream out = new FileOutputStream(fileName + ".decrypted");
        // 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.

        // 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

AWS Encryption SDK for JavaScript 는 JavaScript로 웹 브라우저 애플리케이션을 작성하거나 Node.js로 웹 서버 애플리케이션을 작성하는 개발자를 위한 클라이언트 측 암호화 라이브러리를 제공하도록 설계되었습니다.

의 모든 구현과 마찬가지로 AWS Encryption SDK는 고급 데이터 보호 기능을 AWS Encryption SDK for JavaScript 제공합니다. 이러한 기능에는 봉투 암호화, 추가 인증 데이터(AAD), 안전하고 인증된 대칭 키 알고리즘 제품군(예: 키 유도 및 서명을 사용하는 256비트 AES-GCM)이 포함됩니다.

의 모든 언어별 구현 AWS Encryption SDK 은 언어의 제약 조건에 따라 상호 운용 가능하도록 설계되었습니다. JavaScript의 언어 제약에 대한 자세한 내용은 the section called “호환성” 섹션을 참조하세요.

자세히 알아보기

- 를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-encryption-sdk-javascript 리포지토리를 AWS Encryption SDK for JavaScript참조하세요.
- 프로그래밍 예제는 the section called “예제” 문서와, aws-encryption-sdk-javascript 리포지토리의 example-browser 및 example-node 모듈을 참조하세요.
- 를 사용하여 웹 애플리케이션에서 데이터를 AWS Encryption SDK for JavaScript 암호화하는 실제 예제는 AWS 보안 블로그의 AWS Encryption SDK for JavaScript 및 Node.js를 사용하여 브라우저에서 암호화를 활성화하는 방법을 참조하세요.

주제

- 의 호환성 AWS Encryption SDK for JavaScript
- 설치 AWS Encryption SDK for JavaScript

- 의 모듈 AWS Encryption SDK for JavaScript
- AWS Encryption SDK for JavaScript 예제

# 의 호환성 AWS Encryption SDK for JavaScript

AWS Encryption SDK for JavaScript 는의 다른 언어 구현과 상호 운용 가능하도록 설계되었습니다 AWS Encryption SDK. 대부분의 경우를 사용하여 데이터를 암호화 AWS Encryption SDK for JavaScript 하고 AWS Encryption SDK 명령줄 인터페이스를 포함한 다른 언어 구현으로 데이터를 해독할 수 있습니다. 또한를 사용하여의 다른 언어 구현에서 생성된 암호화된 메시지를 복호화 AWS Encryption SDK for JavaScript 할 수 있습니다 AWS Encryption SDK.

그러나를 사용할 때는 JavaScript 언어 구현 및 웹 브라우저에서 몇 가지 호환성 문제를 알고 AWS Encryption SDK for JavaScript있어야 합니다.

또한 다른 언어 구현을 사용할 때는 호환 가능한 마스터 키 공급자, 마스터 키 및 키링을 구성해야 합니다. 자세한 내용은 키링 호환성을 참조하세요.

## AWS Encryption SDK for JavaScript 호환성

의 JavaScript 구현은 다음과 같은 점에서 다른 언어 구현과 AWS Encryption SDK 다릅니다.

- 의 암호화 작업은 프레임 처리되지 않은 사이퍼텍스트를 반환하지 AWS Encryption SDK for JavaScript 않습니다. 그러나 AWS Encryption SDK for JavaScript 는의 다른 언어 구현에서 반환된 프레임 처리된 사이퍼텍스트와 프레임 처리되지 않은 사이퍼텍스트를 해독합니다 AWS Encryption SDK.
- Node.js 버전 12.9.0 이상 Node.js는 다음 RSA 키 래핑 옵션을 지원합니다.
  - OAEP와 SHA1, SHA256, SHA384 또는 SHA512
  - OAEP와 SHA1 및 MGF1과 SHA1
  - PKCS1v15
- 버전 12.9.0 이전의 Node.js는 다음 RSA 키 래핑 옵션만 지원합니다.
  - OAEP와 SHA1 및 MGF1과 SHA1
  - PKCS1v15

## 브라우저 호환성

일부 웹 브라우저는 AWS Encryption SDK for JavaScript 에 필요한 기본 암호화 작업을 지원하지 않습니다. 브라우저에서 구현하는 WebCrypto API에 대한 폴백을 구성하여 일부 누락된 작업을 보완할 수 있습니다.

### 웹 브라우저 제한 사항

다음 제한 사항은 모든 웹 브라우저에 공통적으로 적용됩니다.

- WebCrypto API는 PKCS1v15 키 래핑을 지원하지 않습니다.
- 브라우저는 192비트 키를 지원하지 않습니다.

### 필요한 암호화 작업

를 사용하려면 웹 브라우저에서 다음 작업이 AWS Encryption SDK for JavaScript 필요합니다. 브라우저가 이러한 작업을 지원하지 않는 경우, AWS Encryption SDK for JavaScript와 호환되지 않습니다.

- 브라우저는 암호화 방식으로 임의의 값을 생성하는 메서드인 `crypto.getRandomValues()`를 포함해야 합니다. `crypto.getRandomValues()`를 지원하는 웹 브라우저 버전에 대한 자세한 내용은 crypto.getRandomValues()를 사용할 수 있나요?를 참조하세요.

### 필요한 폴백

에는 웹 브라우저에서 다음과 같은 라이브러리 및 작업이 AWS Encryption SDK for JavaScript 필요합니다. 이러한 요구 사항을 충족하지 않는 웹 브라우저를 지원하는 경우 폴백을 구성해야 합니다. 그렇지 않으면 브라우저 AWS Encryption SDK for JavaScript 에서를 사용하려는 시도가 실패합니다.

- 웹 애플리케이션에서 기본 암호화 작업을 수행하는 WebCrypto API는 일부 브라우저에서 사용할 수 없습니다. 웹 암호화를 지원하는 웹 브라우저 버전에 대한 자세한 내용은 웹 암호화를 사용할 수 있나요?를 참조하세요.
- 최신 버전의 Safari 웹 브라우저는에 AWS Encryption SDK 필요한 0바이트의 AES-GCM 암호화를 지원하지 않습니다. 브라우저가 WebCrypto API를 구현하지만 AES-GCM을 사용하여 0바이트를 암호화할 수 없는 경우는 0바이트 암호화에만 폴백 라이브러리를 AWS Encryption SDK for JavaScript 사용합니다. 다른 모든 작업에는 WebCrypto API를 사용합니다.

두 가지 중 한 제한 사항에 폴백을 구성하려면 코드에 다음 문을 추가합니다. configureFallback 함수에서 누락된 기능을 지원하는 라이브러리를 지정합니다. 다음 예제는 Microsoft Research JavaScript 암

호화 라이브러리(msrcrypto)를 사용하지만 호환 가능한 라이브러리로 바꿀 수 있습니다. 전체 예제는 fallback.ts를 참조하세요.

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

## 설치 AWS Encryption SDK for JavaScript

는 상호 종속 모듈 모음으로 AWS Encryption SDK for JavaScript 구성됩니다. 여러 모듈은 함께 작동하도록 설계된 모듈 모음일 뿐입니다. 일부 모듈은 독립적으로 작동하도록 설계됩니다. 모든 구현에는 몇 개의 모듈이 필요합니다. 다른 몇 개 모듈은 특수한 경우에만 필요합니다. JavaScript AWS Encryption SDK 용의 모듈에 대한 자세한 내용은 GitHub의 aws-encryption-sdk-javascript 리포지토리에 있는 각 모듈의 의 모듈 AWS Encryption SDK for JavaScript 및 README.md 파일을 참조하세요.

> **Note**
>
> 2.0.0 AWS Encryption SDK for JavaScript 이전의 모든 버전은 end-of-support 단계에 있습니다.
> 코드나 데이터를 변경하지 않고 버전 2.0.x 이상에서 AWS Encryption SDK for JavaScript 의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.0.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 1.7.x 이하 버전에서 2.0.x 이상 버전으로 업데이트하려면 먼저 AWS Encryption SDK for JavaScript의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.

모듈을 설치하려면 npm 패키지 관리자를 사용하세요.

예를 들어 Node.js AWS Encryption SDK for JavaScript 에서 로 프로그래밍해야 하는 모든 모듈이 포함된 client-node 모듈을 설치하려면 다음 명령을 사용합니다.

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

AWS Encryption SDK for JavaScript 브라우저에서 로 프로그래밍해야 하는 모든 모듈이 포함된 client-browser 모듈을 설치하려면 다음 명령을 사용합니다.

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

사용 방법에 대한 실제 예제는 GitHub의 aws-encryption-sdk-javascript 리포지토리에 있는 example-node 및 example-browser 모듈의 예제를 AWS Encryption SDK for JavaScript참조하세요.

# 의 모듈 AWS Encryption SDK for JavaScript

의 모듈을 AWS Encryption SDK for JavaScript 사용하면 프로젝트에 필요한 코드를 쉽게 설치할 수 있습니다.

## JavaScript Node.js 모듈

client-node

Node.js AWS Encryption SDK for JavaScript 에서를 사용하여 프로그래밍해야 하는 모든 모듈을 포함합니다.

caching-materials-manager-node

Node.js의에서 데이터 키 캐싱 기능을 지원하는 함수 AWS Encryption SDK for JavaScript 를 내보냅니다.

decrypt-node

데이터 및 데이터 스트림을 나타내는 암호화된 메시지를 복호화하고 확인하는 함수를 내보냅니다. `client-node` 모듈에 포함됩니다.

encrypt-node

다양한 유형의 데이터를 암호화하고 서명하는 함수를 내보냅니다. `client-node` 모듈에 포함됩니다.

example-node

Node.js AWS Encryption SDK for JavaScript 에서를 사용한 프로그래밍의 실제 예제를 내보냅니다. 다양한 유형의 키링 및 다양한 유형의 데이터 예제를 포함합니다.

hkdf-node

Node.js의가 특정 알고리즘 제품군 AWS Encryption SDK for JavaScript 에서 사용하는 HMAC 기반 키 파생 함수(HKDF)를 내보냅니다. 브라우저 AWS Encryption SDK for JavaScript 의는 WebCrypto API의 기본 HKDF 함수를 사용합니다.

integration-node

Node.js AWS Encryption SDK for JavaScript 의가의 다른 언어 구현과 호환되는지 확인하는 테스트를 정의합니다 AWS Encryption SDK.

kms-keyring-node

Node.js에서 AWS KMS 키링을 지원하는 함수를 내보냅니다.

raw-aes-keyring-node

Node.js에서 Raw AES 키링을 지원하는 함수를 내보냅니다.

raw-rsa-keyring-node

Node.js에서 Raw RSA 키링을 지원하는 함수를 내보냅니다.

### JavaScript 브라우저용 모듈

client-browser

AWS Encryption SDK for JavaScript 브라우저에서를 사용하여 프로그래밍해야 하는 모든 모듈을 포함합니다.

caching-materials-manager-browser

브라우저에서 JavaScript용으로 데이터 키 캐싱 기능을 지원하는 함수를 내보냅니다.

decrypt-browser

데이터 및 데이터 스트림을 나타내는 암호화된 메시지를 복호화하고 확인하는 함수를 내보냅니다.

encrypt-browser

다양한 유형의 데이터를 암호화하고 서명하는 함수를 내보냅니다.

example-browser

브라우저 AWS Encryption SDK for JavaScript 에서를 사용한 프로그래밍의 실제 예제입니다. 다양한 유형의 키링 및 다양한 유형의 데이터 예제를 포함합니다.

integration-browser

브라우저의 스크립트가 AWS Encryption SDK for Java의 다른 언어 구현과 호환되는지 확인하는 테스트를 정의합니다 AWS Encryption SDK.

kms-keyring-browser

브라우저에서 AWS KMS 키링을 지원하는 함수를 내보냅니다.

raw-aes-keyring-browser

브라우저에서 Raw AES 키링을 지원하는 함수를 내보냅니다.

raw-rsa-keyring-browser

브라우저에서 Raw RSA 키링을 지원하는 함수를 내보냅니다.

### 모든 구현을 위한 모듈

cache-material

데이터 키 캐싱 기능을 지원합니다. 각 데이터 키로 캐시된 암호화 자료를 어셈블하기 위한 코드를 제공합니다.

kms-keyring

KMS 키링을 지원하는 함수를 내보냅니다.

material-management

암호화 자료 관리자(CMM)를 구현합니다.

raw-keyring

Raw AES 및 RSA 키링에 필요한 함수를 내보냅니다.

serialize

SDK가 출력을 직렬화하는 데 사용하는 함수를 내보냅니다.

web-crypto-backend

브라우저의에서 WebCrypto API를 사용하는 함수 AWS Encryption SDK for JavaScript 를 내보냅니다.

## AWS Encryption SDK for JavaScript 예제

다음 예제에서는 AWS Encryption SDK for JavaScript 를 사용하여 데이터를 암호화 및 복호화하는 방법을 보여줍니다.

GitHub의 aws-encryption-sdk-javascript AWS Encryption SDK for JavaScript 리포지토리의 example-node 및 example-browser 모듈에서를 사용하는 더 많은 예제를 찾을 수 있습니다. https://github.com/aws/aws-encryption-sdk-javascript/tree/master/modules/example-node aws-encryption-sdk-javascript 이러한 예제 모듈은 `client-browser` 또는 `client-node` 모듈을 설치할 때 설치되지 않습니다.

전체 코드 샘플을 참조: 노드: kms_simple.ts, 브라우저: kms_simple.ts

주제

- AWS KMS 키링을 사용하여 데이터 암호화
- AWS KMS 키링을 사용하여 데이터 복호화

## AWS KMS 키링을 사용하여 데이터 암호화

다음 예제에서는를 사용하여 짧은 문자열 또는 바이트 배열 AWS Encryption SDK for JavaScript 을 암호화하고 해독하는 방법을 보여줍니다.

이 예제에서는 AWS KMS를 사용하여 데이터 키를 생성하고 암호화하는 키링 유형인 키링 AWS KMS key 이 특징입니다. 생성에 대한 도움말은 AWS Key Management Service 개발자 안내서의 키 생성을 AWS KMS key참조하세요. https://docs.aws.amazon.com/kms/latest/developerguide/create-keys.html AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요. AWS KMS 키링 AWS KMS keys 에서 식별

1단계: 커밋 정책을 설정합니다.

버전 1.7.x부터 AWS Encryption SDK 클라이언트를 인스턴스화하는 새 `buildClient` 함수를 호출할 때 커밋 정책을 설정할 AWS Encryption SDK for JavaScript수 있습니다. `buildClient` 함수는 커밋 정책을 나타내는 열거형 값을 사용합니다. 암호화 및 복호화 시 커밋 정책을 적용하는 업데이트된 `encrypt` 및 `decrypt` 함수를 반환합니다.

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한`buildClient`할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

JavaScript Browser

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

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

JavaScript Node.js

```
import {
  KmsKeyringNode,
  buildClient,
```

```
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

2단계: 키링을 구성합니다.

암호화를 위한 AWS KMS 키링을 생성합니다.

AWS KMS 키링으로 암호화할 때는 생성기 키, 즉 일반 텍스트 데이터 키를 생성하고 암호화하는 데 AWS KMS key 사용되는를 지정해야 합니다. 동일한 일반 텍스트 데이터 키를 암호화하는 0개 이상의 추가 키를 지정할 수도 있습니다. 키링은 일반 텍스트 데이터 키와 생성기 키를 포함하여 키링의 각 AWS KMS key 에 대한 해당 데이터 키의 암호화된 사본 하나를 반환합니다. 데이터를 복호화하려면 암호화된 데이터 키 중 하나를 복호화해야 합니다.

에서 암호화 키링에 AWS KMS keys 대해를 지정하려면 지원되는 모든 AWS KMS 키 식별자를 사용할 AWS Encryption SDK for JavaScript수 있습니다. 이 예에서는 별칭 ARN으로 식별되는 생성기 키와 키 ARN으로 식별되는 추가 키 하나를 사용합니다.

> **Note**
>
> 복호화를 위해 AWS KMS 키링을 재사용하려는 경우 키 ARNs 사용하여 키링 AWS KMS keys 에서를 식별해야 합니다.

이 코드를 실행하기 전에 예제 AWS KMS key 식별자를 유효한 식별자로 바꿉니다. 키링에서 AWS KMS keys를 사용하는 데 필요한 권한이 있어야 합니다.

JavaScript Browser

브라우저에 자격 증명을 제공하여 시작하세요. 이 AWS Encryption SDK for JavaScript 예에서는 자격 증명 상수를 실제 자격 증명으로 대체하는 webpack.DefinePlugin을 사용합니다. 그러나 모든 방법을 사용하여 자격 증명을 제공할 수 있습니다. 그런 다음 자격 증명을 사용하여 AWS KMS 클라이언트를 생성합니다.

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

const clientProvider = getClient(KMS, {
```

```
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken
  }
})
```

그런 다음 생성기 키 및 추가 키에 AWS KMS keys 를 지정합니다. 그런 다음 클라이언트와를 사용하여 AWS KMS AWS KMS 키링을 생성합니다 AWS KMS keys.

```
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

```
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 })
```

3단계: 암호화 컨텍스트를 설정합니다.

암호화 컨텍스트는 비밀이 아닌 임의의 추가 인증 데이터입니다. 암호화에 암호화 컨텍스트를 제공하면 AWS Encryption SDK 는 암호화 컨텍스트를 사이퍼텍스트에 암호화 방식으로 바인딩하여 데이터를 복호화하는 데 동일한 암호화 컨텍스트가 필요합니다. 암호화 컨텍스트를 사용하는 것은 선택 사항이지만 권장되는 모범 사례입니다.

암호화 컨텍스트 페어를 포함하는 단순 객체를 만듭니다. 각 페어의 키와 값은 문자열이어야 합니다.

JavaScript Browser

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

JavaScript Node.js

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

4단계: 데이터를 암호화합니다.

일반 텍스트 데이터를 암호화하려면 `encrypt` 함수를 호출합니다. AWS KMS 키링, 일반 텍스트 데이터 및 암호화 컨텍스트를 전달합니다.

이 `encrypt` 함수는 암호화된 데이터, 암호화된 데이터 키 및 암호화 컨텍스트 및 서명을 포함한 중요한 메타데이터를 포함하는 암호화된 메시지(`result`)를 반환합니다.

지원되는 프로그래밍 언어 AWS Encryption SDK 에 대해를 사용하여 이 암호화된 메시지를 복호화할 수 있습니다.

JavaScript Browser

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

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

JavaScript Node.js

```
const plaintext = 'asdf'

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

## AWS KMS 키링을 사용하여 데이터 복호화

를 사용하여 암호화된 메시지를 해독하고 원본 데이터를 복구 AWS Encryption SDK for JavaScript 할 수 있습니다.

이 예에서는 the section called “AWS KMS 키링을 사용하여 데이터 암호화” 예제에서 암호화된 데이터를 복호화합니다.

1단계: 커밋 정책을 설정합니다.

버전 1.7.x부터 AWS Encryption SDK 클라이언트를 인스턴스화하는 새 `buildClient` 함수를 호출할 때 커밋 정책을 설정할 AWS Encryption SDK for JavaScript수 있습니다. `buildClient` 함수는 커밋 정책을 나타내는 열거형 값을 사용합니다. 암호화 및 복호화 시 커밋 정책을 적용하는 업데이트된 `encrypt` 및 `decrypt` 함수를 반환합니다.

다음 예제에서는 `buildClient` 함수를 사용하여 기본 커밋 정책인를 지정합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 를 사용하여 암호화된 메시지의 암호화된 데이터 키 수를 제한`buildClient`할 수도 있습니다. 자세한 내용은 the section called “암호화된 데이터 키 제한” 단원을 참조하십시오.

JavaScript Browser

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

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

JavaScript Node.js

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

2단계: 키링을 구성합니다.

데이터를 복호화하려면 `encrypt` 함수가 반환한 암호화된 메시지(`result`)를 전달하세요. 암호화된 메시지는 암호화된 데이터, 암호화된 데이터 키 및 암호화 컨텍스트 및 서명을 포함한 중요한 메타데이터를 포함합니다.

또한 복호화할 때 AWS KMS 키링을 지정해야 합니다. 데이터를 암호화하는 데 사용된 것과 동일한 키링이나 다른 키링을 사용할 수 있습니다. 성공하려면 복호화 키링 AWS KMS key 에 있는 하나 이상의가 암호화된 메시지의 암호화된 데이터 키 중 하나를 복호화할 수 있어야 합니다. 데이터 키가 생성되지 않으므로 복호화 키링에 생성기 키를 지정할 필요가 없습니다. 이렇게 하면 생성기 키와 추가 키가 같은 방식으로 처리됩니다.

에서 복호화 키링에 AWS KMS key 대해를 지정하려면 키 ARN을 사용해야 AWS Encryption SDK for JavaScript합니다. 그렇지 않으면 AWS KMS key 가 인식되지 않습니다. AWS KMS 키링 AWS KMS keys 에서를 식별하는 데 도움이 필요하면 섹션을 참조하세요. AWS KMS 키링 AWS KMS keys 에서 식별

> **Note**
>
> 암호화 및 복호화에 동일한 키링을 사용하는 경우 키 ARNs 사용하여 키링 AWS KMS keys 에서를 식별합니다.

이 예제에서는 암호화 키링 AWS KMS keys 에 중 하나만 포함하는 키링을 생성합니다. 이 코드를 실행하기 전에 예제 키 ARN을 유효한 키로 바꿉니다. AWS KMS key에 대한 `kms:Decrypt` 권한이 있어야 합니다.

JavaScript Browser

브라우저에 자격 증명을 제공하여 시작하세요. 이 AWS Encryption SDK for JavaScript 예에서는 자격 증명 상수를 실제 자격 증명으로 대체하는 webpack.DefinePlugin을 사용합니다. 그러나 모든 방법을 사용하여 자격 증명을 제공할 수 있습니다. 그런 다음 자격 증명을 사용하여 AWS KMS 클라이언트를 생성합니다.

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

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

```
    sessionToken
  }
})
```

그런 다음 AWS KMS 클라이언트를 사용하여 AWS KMS 키링을 생성합니다. 이 예제에서는 AWS KMS keys 암호화 키링의 중 하나만 사용합니다.

```
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 })
```

3단계: 데이터 암호화를 해제합니다.

그런 다음 `decrypt` 함수를 호출합니다. 방금 생성한 복호화 키링(`keyring`)과, `encrypt` 함수가 반환한 암호화된 메시지(`result`)를 전달하세요. 는 키링을 AWS Encryption SDK 사용하여 암호화된 데이터 키 중 하나를 복호화합니다. 그런 다음 일반 텍스트 데이터 키를 사용하여 데이터를 복호화합니다.

호출이 성공하면 `plaintext` 필드에 일반 텍스트(복호화된) 데이터가 포함됩니다. 이 `messageHeader` 필드에는 데이터 복호화에 사용된 암호화 컨텍스트를 포함하여 복호화 프로세스에 대한 메타데이터가 포함됩니다.

JavaScript Browser

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

JavaScript Node.js

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

4단계: 암호화 컨텍스트를 확인합니다.

데이터를 복호화하는 데 사용된 암호화 컨텍스트는 `decrypt` 함수가 반환하는 메시지 헤더(`messageHeader`)에 포함됩니다. 애플리케이션에서 일반 텍스트 데이터를 반환하기 전에 암호화

할 때 제공한 암호화 컨텍스트가 복호화할 때 사용된 암호화 컨텍스트에 포함되어 있는지 확인합니다. 불일치는 데이터가 변조되었거나 올바른 암호화 텍스트를 복호화하지 않았음을 나타낼 수 있습니다.

암호화 컨텍스트를 확인할 때 정확히 일치할 필요는 없습니다. 서명과 함께 암호화 알고리즘을 사용하는 경우 암호화 자료 관리자(CMM)은 메시지를 암호화하기 전에 암호화 컨텍스트에 퍼블릭 서명 키를 추가합니다. 그러나 제출한 모든 암호화 컨텍스트 페어는 반환된 암호화 컨텍스트에 포함되어야 합니다.

먼저 메시지 헤더에서 암호화 컨텍스트를 가져옵니다. 그런 다음 원래 암호화 컨텍스트(`context`)의 각 키-값 페어가 반환된 암호화 컨텍스트(`encryptionContext`)의 키-값 페어와 일치하는지 확인합니다.

JavaScript Browser

```
const { encryptionContext } = messageHeader

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

JavaScript Node.js

```
const { encryptionContext } = messageHeader

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

암호화 컨텍스트 검사가 성공하면 일반 텍스트 데이터를 반환할 수 있습니다.

# AWS Encryption SDK for Python

이 주제에서는 AWS Encryption SDK for Python를 설치 및 사용하는 방법을 설명합니다. 를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-encryption-sdk-python 리포지토리를 AWS Encryption SDK for Python참조하세요. API 설명서는 문서 읽기를 참조하세요.

주제



## 사전 조건

를 설치하기 전에 다음 사전 조건이 있는지 AWS Encryption SDK for Python확인합니다.

지원되는 Python 버전

AWS Encryption SDK for Python 버전 3.2.0 이상에서는 Python 3.8 이상이 필요합니다.

> **Note**
>
> AWS 암호화 자료 공급자 라이브러리(MPL)는 버전 4.x에 AWS Encryption SDK for Python 도입된에 대한 선택적 종속성입니다. MPL을 설치하려는 경우 Python 3.11 이상을 사용해야 합니다.

이전 버전의는 Python 2.7 및 Python 3.4 이상을 AWS Encryption SDK 지원하지만 최신 버전의를 사용하는 것이 좋습니다 AWS Encryption SDK.

Python을 다운로드하려면 Python 다운로드를 참조하세요.

Python용 pip 설치 도구

`pip`는 Python 3.6 이상 버전에 포함되어 있지만 업그레이드가 필요할 수도 있습니다. `pip` 업그레이드 또는 설치에 관한 자세한 정보는 `pip` 설명서의 설치를 참조하세요.

## 설치

AWS Encryption SDK for Python의 최신 버전을 설치합니다.

> **Note**
>
> 3.0.0 AWS Encryption SDK for Python 이전의 모든 버전은 end-of-support 단계에 있습니다. 코드나 데이터를 변경하지 않고 버전 2.0.x 이상에서 AWS Encryption SDK 의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.0.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 1.7.x 이하 버전에서 2.0.x 이상 버전으로 업데이트하려면 먼저 AWS Encryption SDK의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.

다음 예제와 AWS Encryption SDK for Python같이 `pip`를 사용하여를 설치합니다.

최신 버전 설치

```
pip install "aws-encryption-sdk[MPL]"
```

접`[MPL]`미사는 AWS 암호화 자료 공급자 라이브러리(MPL)를 설치합니다. MPL에는 데이터를 암호화하고 해독하기 위한 구문이 포함되어 있습니다. MPL은 버전 4.x에 AWS Encryption SDK for Python 도입된에 대한 선택적 종속성입니다. MPL을 설치하는 것이 좋습니다. 그러나 MPL을 사용하지 않으려는 경우 접`[MPL]`미사를 생략할 수 있습니다.

pip를 사용하여 패키지를 설치 및 업그레이드하는 방법에 대한 자세한 내용은 패키지 설치를 참조하세요.

에는 모든 플랫폼에서 암호화 라이브러리(pyca/cryptography)가 AWS Encryption SDK for Python 필요합니다. `pip`의 모든 버전은 `cryptography` 라이브러리를 Windows에 자동으로 설치하고 빌드합니다. `pip` 8.1 이상 버전은 Linux에 `cryptography`를 자동으로 설치하고 빌드합니다. 이하 버전의 `pip`를 사용 중이며 `cryptography` 라이브러리 빌드에 필요한 도구가 Linux 환경에 없는 경우에는 이러한 도구를 설치해야 합니다. 자세한 내용은 Linux에서 암호화 빌드를 참조하세요.

버전 1.10.0 및 2.5.0은 2.5.0과 3.3.2 사이의 암호화 종속성을 AWS Encryption SDK for Python 고정합니다. 다른 버전의는 최신 버전의 암호화를 AWS Encryption SDK for Python 설치합니다. 3.3.2 이상 버전의 암호화가 필요한 경우 AWS Encryption SDK for Python의 최신 메이저 버전을 사용하는 것이 좋습니다.

의 최신 개발 버전은 GitHub의 aws-encryption-sdk-python 리포지토리를 AWS Encryption SDK for Python참조하십시오.

를 설치한 후이 가이드의 Python 예제 코드를 살펴보 AWS Encryption SDK for Python면서 시작합니다.

## AWS Encryption SDK for Python 예제 코드

다음 예제에서는를 사용하여 데이터를 암호화하고 복호화 AWS Encryption SDK for Python 하는 방법을 보여줍니다.

이 섹션의 예제에서는 선택적 암호화 자료 공급자 라이브러리 종속성()과 AWS Encryption SDK for Python 함께 버전 4.x를 사용하는 방법을 보여줍니다`aws-cryptographic-material-providers`. 이전 버전을 사용하거나 재료 공급자 라이브러리(MPL)가 없는 설치를 사용하는 예제를 보려면 GitHub의 aws-encryption-sdk-python 리포지토리 릴리스 목록에서 릴리스를 찾습니다.

MPL과 AWS Encryption SDK for Python 함께 버전 4.x를 사용하는 경우 키링을 사용하여 봉투 암호화를 수행합니다. 는 이전 버전에서 사용한 마스터 키 공급자와 호환되는 키링을 AWS Encryption SDK 제공합니다. 자세한 내용은 the section called “키링 호환성” 단원을 참조하십시오. 마스터 키 공급자에서 키링으로 마이그레이션하는 방법에 대한 예제는 GitHub의 `aws-encryption-sdk-python`리포지토리에서 마이그레이션 예제를 참조하세요.

주제

- 문자열 암호화 및 복호화
- 바이트 스트림 암호화 및 복호화

### 문자열 암호화 및 복호화

다음 예제에서는를 사용하여 문자열을 암호화하고 복호화 AWS Encryption SDK 하는 방법을 보여줍니다. 이 예제에서는 대칭 암호화 KMS 키와 함께 키AWS KMS 링을 사용합니다.

이 예제에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. 자세한 내용은 the section called “커밋 정책 설정” 단원을 참조하십시오.

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example sets up the KMS Keyring

The AWS KMS keyring uses symmetric encryption KMS keys to generate, encrypt and
decrypt data keys. This example creates a KMS Keyring and then encrypts a custom input
 EXAMPLE_DATA
```

```
with an encryption context. This example also includes some sanity checks for
 demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
These sanity checks are for demonstration in the example only. You do not need these in
 your code.

AWS KMS keyrings can be used independently or in a multi-keyring with other keyrings
of the same or a different type.

"""

import boto3
from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import CreateAwsKmsKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy

EXAMPLE_DATA: bytes = b"Hello World"


def encrypt_and_decrypt_with_keyring(
    kms_key_id: str
):
    """Demonstrate an encrypt/decrypt cycle using an AWS KMS keyring.

    Usage: encrypt_and_decrypt_with_keyring(kms_key_id)
    :param kms_key_id: KMS Key identifier for the KMS key you want to use for
 encryption and
    decryption of your data keys.
    :type kms_key_id: string

    """
    # 1. Instantiate the encryption SDK client.
    # 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 were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. Create a boto3 client for KMS.
    kms_client = boto3.client('kms', region_name="us-west-2")

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "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",
    }

    # 4. Create your keyring
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
    )

    keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
        kms_key_id=kms_key_id,
        kms_client=kms_client
    )

    kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
        input=keyring_input
    )

    # 5. Encrypt the data with the encryptionContext.
    ciphertext, _ = client.encrypt(
        source=EXAMPLE_DATA,
        keyring=kms_keyring,
        encryption_context=encryption_context
    )

    # 6. Demonstrate that the ciphertext and plaintext are different.
```

```
    # (This is an example for demonstration; you do not need to do this in your own
 code.)
    assert ciphertext != EXAMPLE_DATA, \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 7. Decrypt your encrypted data using the same keyring you used on encrypt.
    plaintext_bytes, _ = client.decrypt(
        source=ciphertext,
        keyring=kms_keyring,
        # Provide the encryption context that was supplied to the encrypt method
        encryption_context=encryption_context,
    )

    # 8. Demonstrate that the decrypted plaintext is identical to the original
 plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
 code.)
    assert plaintext_bytes == EXAMPLE_DATA, \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
 decryption"
```

## 바이트 스트림 암호화 및 복호화

다음 예제에서는를 사용하여 바이트 스트림을 암호화하고 복호화 AWS Encryption SDK 하는 방법을 보여줍니다. 이 예제에서는 원시 AES 키링을 사용합니다.

이 예제에서는 기본 커밋 정책인를 사용하여 AWS Encryption SDK 클라이언트를 인스턴스화합니다REQUIRE_ENCRYPT_REQUIRE_DECRYPT. 자세한 내용은 the section called “커밋 정책 설정” 단원을 참조하십시오.

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example demonstrates file streaming for encryption and decryption.

File streaming is useful when the plaintext or ciphertext file/data is too large to
 load into
memory. Therefore, the AWS Encryption SDK allows users to stream the data, instead of
 loading it
all at once in memory. In this example, we demonstrate file streaming for encryption
 and decryption
using a Raw AES keyring. However, you can use any keyring with streaming.
```

```
This example creates a Raw AES Keyring and then encrypts an input stream from the file
`plaintext_filename` with an encryption context to an output (encrypted) file
 `ciphertext_filename`.
It then decrypts the ciphertext from `ciphertext_filename` to a new file
 `decrypted_filename`.
This example also includes some sanity checks for demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
These sanity checks are for demonstration in the example only. You do not need these in
 your code.

See raw_aes_keyring_example.py in the same directory for another raw AES keyring
 example
in the AWS Encryption SDK for Python.
"""
import filecmp
import secrets

from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import AesWrappingAlg,
 CreateRawAesKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def encrypt_and_decrypt_with_keyring(
    plaintext_filename: str,
    ciphertext_filename: str,
    decrypted_filename: str
):
    """Demonstrate a streaming encrypt/decrypt cycle.

    Usage: encrypt_and_decrypt_with_keyring(plaintext_filename
                                            ciphertext_filename
                                            decrypted_filename)
    :param plaintext_filename: filename of the plaintext data
    :type plaintext_filename: string
    :param ciphertext_filename: filename of the ciphertext data
    :type ciphertext_filename: string
```

```
    :param decrypted_filename: filename of the decrypted data
    :type decrypted_filename: string
    """
    # 1. Instantiate the encryption SDK client.
    # 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 were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. The key namespace and key name are defined by you.
    # and are used by the Raw AES keyring to determine
    # whether it should attempt to decrypt an encrypted data key.
    key_name_space = "Some managed raw keys"
    key_name = "My 256-bit AES wrapping key"

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "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",
    }

    # 4. Generate a 256-bit AES key to use with your keyring.
    # In practice, you should get this key from a secure key management system such as
an HSM.

    # Here, the input to secrets.token_bytes() = 32 bytes = 256 bits
    static_key = secrets.token_bytes(32)

    # 5. Create a Raw AES keyring
    # We choose to use a raw AES keyring, but any keyring can be used with streaming.
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
```

```
    )

    keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
        key_namespace=key_name_space,
        key_name=key_name,
        wrapping_key=static_key,
        wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
    )

    raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
        input=keyring_input
    )

    # 6. Encrypt the data stream with the encryptionContext
    with open(plaintext_filename, 'rb') as pt_file, open(ciphertext_filename, 'wb') as
ct_file:
        with client.stream(
            mode='e',
            source=pt_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as encryptor:
            for chunk in encryptor:
                ct_file.write(chunk)

    # 7. Demonstrate that the ciphertext and plaintext are different.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert not filecmp.cmp(plaintext_filename, ciphertext_filename), \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 8. Decrypt your encrypted data stream using the same keyring you used on
encrypt.
    with open(ciphertext_filename, 'rb') as ct_file, open(decrypted_filename, 'wb') as
pt_file:
        with client.stream(
            mode='d',
            source=ct_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as decryptor:
            for chunk in decryptor:
                pt_file.write(chunk)
```

```
    # 10. Demonstrate that the decrypted plaintext is identical to the original
plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert filecmp.cmp(plaintext_filename, decrypted_filename), \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
decryption"
```

# AWS Encryption SDK Rust용

이 주제에서는 AWS Encryption SDK for Rust를 설치하고 사용하는 방법을 설명합니다. AWS Encryption SDK for Rust를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-encryption-sdk 리포지토리의 Rust 디렉터리를 참조하세요.

AWS Encryption SDK for Rust는 다음과 AWS Encryption SDK 같은 방식으로의 다른 프로그래밍 언어 구현과 다릅니다.

- 데이터 키 캐싱은 지원되지 않습니다. 그러나 AWS Encryption SDK for Rust는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 지원합니다.
- 스트리밍 데이터가 지원되지 않음

AWS Encryption SDK for Rust에는의 다른 언어 구현 버전 2.0.x 이상에 도입된 모든 보안 기능이 포함되어 있습니다 AWS Encryption SDK. 그러나 AWS Encryption SDK for Rust를 사용하여의 다른 언어 구현인 2.0.x 이전 버전으로 암호화된 데이터를 복호화 AWS Encryption SDK하는 경우 커밋 정책을 조정해야 할 수 있습니다. 자세한 내용은 커밋 정책 설정 방법을 참조하세요.

AWS Encryption SDK for Rust는 사양을 작성하는 공식 확인 언어인 Dafny AWS Encryption SDK 의 제품이며, 이를 구현할 코드와 이를 테스트하기 위한 증거입니다. 그 결과, 기능적 정확성을 보장하는 프레임워크에서 AWS Encryption SDK 의 기능을 구현하는 라이브러리가 탄생했습니다.

자세히 알아보기

- 대체 알고리즘 제품군 지정 AWS Encryption SDK, 암호화된 데이터 키 제한, AWS KMS 다중 리전 키 사용 등에서 옵션을 구성하는 방법을 보여주는 예제는 섹션을 참조하세요구성 AWS Encryption SDK.
- AWS Encryption SDK for Rust를 구성하고 사용하는 방법을 보여주는 예제는 GitHub의 aws-encryption-sdk 리포지토리에 있는 Rust 예제를 참조하세요.

주제



## 사전 조건

AWS Encryption SDK for Rust를 설치하기 전에 다음 사전 조건이 있는지 확인합니다.

Rust 및 Cargo 설치

Rustup을 사용하여 현재 안정적인 Rust 릴리스를 설치합니다.

rustup 다운로드 및 설치에 대한 자세한 내용은 카고 북의 설치 절차를 참조하세요.

## 설치

AWS Encryption SDK for Rust는 Crates.io `aws-esdk` 크레이트로 사용할 수 있습니다. AWS Encryption SDK for Rust 설치 및 빌드에 대한 자세한 내용은 GitHub의 aws-encryption-sdk 리포지토리에서 README.md 참조하십시오.

다음과 같은 방법으로 AWS Encryption SDK for Rust를 설치할 수 있습니다.

직접

AWS Encryption SDK for Rust를 설치하려면 aws-encryption-sdk GitHub 리포지토리를 복제하거나 다운로드합니다.

Crates.io 사용

프로젝트 디렉터리에서 다음 Cargo 명령을 실행합니다.

```
cargo add aws-esdk
```

또는 Cargo.toml에 다음 줄을 추가합니다.

```
aws-esdk = "<version>"
```

# AWS Encryption SDK for Rust 예제 코드

다음 예제에서는 AWS Encryption SDK for Rust를 사용하여 프로그래밍할 때 사용하는 기본 코딩 패턴을 보여줍니다. 특히 AWS Encryption SDK 및 재료 공급자 라이브러리를 인스턴스화합니다. 그런 다음 각 메서드를 호출하기 전에 메서드에 대한 입력을 정의하는 객체를 인스턴스화합니다.

대체 알고리즘 제품군 지정 및 암호화된 데이터 키 제한 AWS Encryption SDK과 같이에서 옵션을 구성하는 방법을 보여주는 예제는 GitHub의 aws-encryption-sdk 리포지토리에 있는 Rust 예제를 참조하세요.

## AWS Encryption SDK for Rust에서 데이터 암호화 및 복호화

이 예제에서는 데이터 암호화 및 복호화의 기본 패턴을 보여줍니다. 하나의 AWS KMS 래핑 키로 보호되는 데이터 키로 작은 파일을 암호화합니다.

1단계: 인스턴스화 AWS Encryption SDK.

의 메서드를 사용하여 데이터를 암호화하고 복호화 AWS Encryption SDK 합니다.

```
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;
```

2단계: AWS KMS 클라이언트를 생성합니다.

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);
```

선택 사항: 암호화 컨텍스트를 생성합니다.

```
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);
```

3단계: 재료 공급자 라이브러리를 인스턴스화합니다.

구성 요소 공급자 라이브러리의 메서드를 사용하여, 데이터를 보호하는 키를 지정하는 키링을 만들 수 있습니다.

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
```

4단계: AWS KMS 키링을 생성합니다.

키링을 생성하려면 키링 입력 객체를 사용하여 키링 메서드를 호출합니다. 이 예제에서는 `create_aws_kms_keyring()` 메서드를 사용하고 KMS 키 하나를 지정합니다.

```
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;
```

5단계: 일반 텍스트를 암호화합니다.

```
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

let ciphertext = encryption_response
                    .ciphertext
                    .expect("Unable to unwrap ciphertext from encryption response");
```

6단계: 암호화에 사용한 것과 동일한 키링을 사용하여 암호화된 데이터를 복호화합니다.

```
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
```

```
    .encryption_context(encryption_context)
    .send()
    .await?;

let decrypted_plaintext = decryption_response
                            .plaintext
                            .expect("Unable to unwrap plaintext from decryption
 response");
```

# AWS Encryption SDK 명령줄 인터페이스

AWS Encryption SDK 명령줄 인터페이스(AWS 암호화 CLI)를 사용하면를 사용하여 명령줄과 스크립트에서 대화형으로 데이터를 암호화하고 해독 AWS Encryption SDK 할 수 있습니다. 암호학이나 프로그래밍 전문 지식이 없어도 됩니다.

> **Note**
>
> 4.0.0 이전의 AWS Encryption CLI 버전은 end-of-support 단계에 있습니다.
> 코드나 데이터를 변경하지 않고 버전 2.1.x 이상에서 AWS Encryption CLI의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.1.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 버전 1.7.x 이하에서 업데이트하려면 먼저 AWS Encryption CLI의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.
> 새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

의 모든 구현과 마찬가지로 AWS Encryption SDK AWS 암호화 CLI는 고급 데이터 보호 기능을 제공합니다. 이러한 기능에는 봉투 암호화, 추가 인증 데이터(AAD) 및 보안, 인증, 대칭 키 알고리즘 제품군(예: 키 추출, 키 커밋 및 서명을 사용하는 256비트 AES-GCM)이 포함됩니다.

AWS Encryption CLI는를 기반으로 하며 Linux, macOS 및 Windows에서 지원AWS Encryption SDK for Python됩니다. Linux 또는 macOS의 기본 쉘, Windows의 명령 프롬프트 창(cmd.exe), 모든 시스템의 PowerShell 콘솔에서 명령과 스크립트를 실행하여 데이터를 암호화하고 복호화할 수 있습니다.

AWS Encryption CLI를 AWS Encryption SDK포함하여의 모든 언어별 구현은 상호 운용 가능합니다. 예를 들어를 사용하여 데이터를 암호화AWS Encryption SDK for Java하고 AWS Encryption CLI를 사용하여 복호화할 수 있습니다.

이 주제에서는 AWS Encryption CLI를 소개하고, 이를 설치 및 사용하는 방법을 설명하고, 시작하는 데 도움이 되는 몇 가지 예제를 제공합니다. 빠른 시작은 AWS 보안 블로그의 암호화 CLI를 사용하여 데이터를 AWS 암호화 및 해독하는 방법을 참조하세요. 자세한 내용은 문서 읽기를 참조하고 GitHub의 aws-encryption-sdk-cli 리포지토리에서 AWS Encryption CLI 개발에 참여하세요. aws-encryption-sdk-cli

성능

AWS Encryption CLI는를 기반으로 합니다 AWS Encryption SDK for Python. CLI를 실행할 때마다 Python 런타임의 새 인스턴스가 시작됩니다. 가능한 경우 성능을 개선하려면 일련의 독립 명령 대신 단일 명령을 사용합니다. 예를 들어, 각 파일에 대해 별도의 명령을 실행하는 대신 디렉터리에서 파일을 재귀적으로 처리하는 명령 하나를 실행합니다.

주제

- AWS Encryption SDK 명령줄 인터페이스 설치
- AWS Encryption CLI 사용 방법
- AWS 암호화 CLI의 예
- AWS Encryption SDK CLI 구문 및 파라미터 참조
- AWS 암호화 CLI 버전

## AWS Encryption SDK 명령줄 인터페이스 설치

이 주제에서는 AWS 암호화 CLI를 설치하는 방법을 설명합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리와 문서 읽기를 참조하세요.

주제

- 사전 필수 소프트웨어 설치
- AWS Encryption CLI 설치 및 업데이트

## 사전 필수 소프트웨어 설치

AWS 암호화 CLI는를 기반으로 합니다 AWS Encryption SDK for Python. AWS Encryption CLI를 설치하려면 Python 패키지 관리 도구pip인 Python 및가 필요합니다. Python 및 pip는 지원되는 모든 플랫폼에서 사용할 수 있습니다.

AWS Encryption CLI를 설치하기 전에 다음 사전 조건을 설치합니다.

Python

Python 3.8 이상은 AWS Encryption CLI 버전 4.2.0 이상에서 필요합니다.

이전 버전의 AWS Encryption CLI는 Python 2.7 및 3.4 이상을 지원하지만 최신 버전의 AWS Encryption CLI를 사용하는 것이 좋습니다.

Python은 대부분의 Linux 및 macOS 설치에 포함되어 있지만 Python 3.6 이상으로 업그레이드해야 합니다. 최신 버전의 Python전을 사용하는 것이 좋습니다. Windows에서는 Python을 설치해야 하며, 이는 기본적으로 설치되어 있지 않습니다. Python을 다운로드하고 설치하려면 Python 다운로드를 참조하세요.

Python 설치 여부를 알아보려면 명령줄에서 다음을 입력합니다.

```
python
```

Python 버전을 확인하려면 -V(대문자 V) 파라미터를 사용합니다.

```
python -V
```

Windows에서는 Python을 설치한 후 Path 환경 변수의 값에 Python.exe 파일의 경로를 추가합니다.

기본적으로 Python은 모든 사용자 디렉터리 또는 AppData\Local\Programs\Python 하위 디렉터리의 사용자 프로필 디렉터리($home 또는 %userprofile%)에 설치됩니다. 시스템에서 Python.exe 파일의 위치를 찾으려면 다음 레지스트리 키 중 하나를 확인합니다. PowerShell을 사용하여 레지스트리를 검색할 수 있습니다.

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

pip

`pip`는 Python 패키지 관리자입니다. AWS Encryption CLI 및 해당 종속성을 설치하려면 `pip` 8.1 이상이 필요합니다. `pip` 설치 또는 업그레이드에 도움이 필요하면 `pip` 설명서의 설치를 참조하세요.

Linux 설치에서 8.1 `pip` 이전의 버전은 AWS Encryption CLI에 필요한 암호화 라이브러리를 빌드할 수 없습니다. `pip` 버전을 업데이트하지 않기로 선택한 경우 빌드 도구를 별도로 설치할 수 있습니다. 자세한 내용은 Linux에서 암호화 빌드를 참조하세요.

AWS Command Line Interface

AWS Command Line Interface (AWS CLI)는 AWS 암호화 CLI와 함께 (AWS KMS) AWS KMS keys 에서 AWS Key Management Service 를 사용하는 경우에만 필요합니다. 다른 마스터 키 공급자를 사용하는 경우 AWS CLI 가 필요하지 않습니다.

AWS Encryption CLI와 AWS KMS keys 함께를 사용하려면를 설치하고 구성해야 합니다 AWS CLI. 구성은 인증에 사용하는 자격 증명을 AWS Encryption CLI에서 AWS KMS 사용할 수 있도록 합니다.

## AWS Encryption CLI 설치 및 업데이트

최신 버전의 AWS Encryption CLI를 설치합니다. `pip`를 사용하여 AWS 암호화 CLI를 설치하면 , Python 암호화 라이브러리 및를 포함하여 CLIAWS Encryption SDK for Python에 필요한 라이브러리가 자동으로 설치됩니다AWS SDK for Python (Boto3).

> **Note**
>
> 4.0.0 이전의 AWS Encryption CLI 버전은 end-of-support 단계에 있습니다.
> 코드나 데이터를 변경하지 않고 버전 2.1.x 이상에서 AWS Encryption CLI의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.1.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 버전 1.7.x 이하에서 업데이트하려면 먼저 AWS Encryption CLI의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.
> 새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

최신 버전의 AWS Encryption CLI를 설치하려면

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

AWS Encryption CLI의 최신 버전으로 업그레이드하려면

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

AWS Encryption CLI 및의 버전 번호를 찾으려면 AWS Encryption SDK

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

출력에는 두 라이브러리의 버전 번호가 나열됩니다.

```
aws-encryption-sdk-cli/2.1.0 aws-encryption-sdk/2.0.0
```

AWS Encryption CLI의 최신 버전으로 업그레이드하려면

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

AWS Encryption CLI를 설치하면 아직 설치되지 않은 AWS SDK for Python (Boto3)경우 최신 버전의 도 설치됩니다. Boto3가 설치된 경우 설치 프로그램은 Boto3 버전을 확인하고 필요한 경우 업데이트합니다.

설치된 Boto3 버전 찾기

```
pip show boto3
```

Boto3의 최신 버전으로 업데이트

```
pip install --upgrade boto3
```

현재 개발 중인 AWS Encryption CLI 버전을 설치하려면 GitHub의 aws-encryption-sdk-cli 리포지토리를 참조하세요.

pip를 사용하여 Python 패키지를 설치 및 업그레이드하는 방법에 대한 자세한 내용은 pip 설명서를 참조하세요.

# AWS Encryption CLI 사용 방법

이 주제에서는 AWS 암호화 CLI에서 파라미터를 사용하는 방법을 설명합니다. 예제는 AWS 암호화 CLI의 예 섹션을 참조하세요. 전체 설명서는 문서 읽기를 참조하세요. 이 예제에 표시된 구문은 AWS Encryption CLI 버전 2.1.x 이상용입니다.

> **Note**
>
> 4.0.0 이전의 AWS Encryption CLI 버전은 end-of-support 단계에 있습니다.
> 코드나 데이터를 변경하지 않고 버전 2.1.x 이상에서 AWS Encryption CLI의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.1.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 버전 1.7.x 이하에서 업데이트하려면 먼저 AWS Encryption CLI의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.
> 새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

암호화된 데이터 키를 제한하는 보안 기능을 사용하는 방법을 보여주는 예제는 암호화된 데이터 키 제한 섹션을 참조하세요.

AWS KMS 다중 리전 키를 사용하는 방법을 보여주는 예는 섹션을 참조하세요다중 리전 사용 AWS KMS keys.

주제

- 데이터 암호화 및 복호화 방법
- 래핑 키를 지정하는 방법
- 입력을 제공하는 방법
- 출력 위치를 지정하는 방법
- 암호화 컨텍스트를 사용하는 방법
- 커밋 정책을 지정하는 방법
- 구성 파일에 파라미터를 저장하는 방법

## 데이터 암호화 및 복호화 방법

AWS Encryption CLI는의 기능을 사용하여 데이터를 쉽게 암호화하고 해독할 수 AWS Encryption SDK 있습니다.

> **Note**
>
> 이 `--master-keys` 파라미터는 AWS Encryption CLI의 버전 1.8.x에서 더 이상 사용되지 않으며 버전 2.1x에서 제거되었습니다. 대신 `--wrapping-keys` 파라미터를 사용합니다. 버전 2.1.x부터 암호화 및 복호화 시 `--wrapping-keys` 파라미터가 필요합니다. 자세한 내용은 AWS Encryption SDK CLI 구문 및 파라미터 참조을 참조하세요.

- AWS 암호화 CLI에서 데이터를 암호화할 때 일반 텍스트 데이터와 AWS KMS key in AWS Key Management Service ()과 같은 래핑 키(또는 마스터 키)를 지정합니다AWS KMS. 사용자 지정 마스터 키 공급자를 사용하는 경우 공급자를 지정해야 합니다. 또한 암호화 작업에 대한 암호화된 메시지 및 메타데이터의 출력 위치를 지정할 수 있습니다. 암호화 컨텍스트는 선택 사항이지만 권장됩니다.

  버전 1.8.x에서, `--wrapping-keys` 파라미터를 사용하는 경우 `--commitment-policy` 파라미터가 필요하며, 그러지 않으면 유효하지 않습니다. 버전 2.1.x부터 `--commitment-policy` 파라미터는 선택 사항이지만 권장됩니다.

  ```
  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
  ```

  AWS Encryption CLI는 고유한 데이터 키로 데이터를 암호화합니다. 그러면 지정한 래핑 키에서 데이터 키가 암호화됩니다. 작업에 대한 암호화된 메시지 및 메타데이터를 반환합니다. 암호화된 메시지에는 암호화된 데이터(사이퍼텍스트) 및 암호화된 데이터 키 사본이 포함되어 있습니다. 데이터 키의 저장, 관리 또는 분실에 대해 걱정할 필요가 없습니다.

- 데이터를 복호화할 때는 암호화된 메시지, 선택적 암호화 컨텍스트, 일반 텍스트 출력 및 메타데이터의 위치를 전달합니다. 또한 AWS 암호화 CLI가 메시지를 해독하는 데 사용할 수 있는 래핑 키를 지정하거나 AWS 암호화 CLI에 메시지를 암호화한 래핑 키를 사용할 수 있다고 알립니다.

버전 1.8.x부터 복호화 시 `--wrapping-keys` 파라미터는 선택 사항이지만 권장됩니다. 버전 2.1.x부터 암호화 및 복호화 시 `--wrapping-keys` 파라미터가 필요합니다.

복호화할 때 `--wrapping-keys` 파라미터의 key 속성을 사용하여 데이터를 복호화하는 래핑 키를 지정할 수 있습니다. 복호화 시 AWS KMS 래핑 키를 지정하는 것은 선택 사항이지만 사용하지 않을 키를 사용하지 못하게 하는 모범 사례입니다. 사용자 지정 마스터 키 공급자를 사용하는 경우 공급자 및 래핑 키를 지정해야 합니다.

키 속성을 사용하지 않는 경우 `--wrapping-keys` 파라미터의 검색 속성을 로 설정해야 합니다. `true`그러면 AWS 암호화 CLI가 메시지를 암호화한 래핑 키를 사용하여 복호화할 수 있습니다.

암호화된 데이터 키가 너무 많은 잘못된 형식의 메시지를 복호화하지 않도록 `--max-encrypted-data-keys` 파라미터를 사용하는 것이 모범 사례입니다. 암호화된 데이터 키의 예상 수(암호화에 사용되는 각 래핑 키당 하나) 또는 합리적인 최대값(예: 5개)을 지정합니다. 자세한 내용은 암호화된 데이터 키 제한을 참조하세요.

`--buffer` 파라미터는 디지털 서명이 있는지 확인하는 등 모든 입력이 처리된 후에만 일반 텍스트를 반환합니다.

`--decrypt-unsigned` 파라미터는 사이퍼텍스트를 복호화하고, 복호화 전에 메시지가 서명되지 않도록 합니다. `--algorithm` 파라미터를 사용하고 디지털 서명이 없는 알고리즘 제품군을 선택하여 데이터를 암호화한 경우 이 파라미터를 사용합니다. 사이퍼텍스트가 서명된 경우 복호화가 실패합니다.

`--decrypt` 또는 `--decrypt-unsigned`를 복호화에 사용할 수 있지만 둘 다 사용할 수는 없습니다.

```
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
```

AWS Encryption CLI는 래핑 키를 사용하여 암호화된 메시지의 데이터 키를 복호화합니다. 그런 다음 데이터 키를 사용하여 해당 데이터를 복호화합니다. 작업에 대한 일반 텍스트 데이터 및 메타데이터를 반환합니다.

## 래핑 키를 지정하는 방법

AWS 암호화 CLI에서 데이터를 암호화할 때는 하나 이상의 래핑 키(또는 마스터 키)를 지정해야 합니다. AWS KMS keys in AWS Key Management Service (AWS KMS), 사용자 지정 마스터 키 공급자의 래핑 키 또는 둘 다를 사용할 수 있습니다. 사용자 지정 마스터 키 공급자는 호환되는 모든 Python 마스터 키 공급자일 수 있습니다.

버전 1.8.x 이상에서 래핑 키를 지정하려면 `--wrapping-keys` 파라미터(`-w`)를 사용합니다. 이 파라미터의 값은 `attribute=value` 형식의 속성 모음입니다. 사용하는 속성은 마스터 키 공급자 및 명령에 따라 달라집니다.

- AWS KMS. 암호화 명령에서 key 속성이 있는 `--wrapping-keys` 파라미터를 지정해야 합니다. 버전 2.1.x부터 복호화 명령에 `--wrapping-keys` 파라미터도 필요합니다. 복호화할 때는 `--wrapping-keys` 파라미터에 값이 `true`인 key 속성 또는 discovery 속성이 있어야 합니다(둘 다는 아님). 다른 속성은 선택 사항입니다.
- 사용자 지정 마스터 키 공급자. 모든 명령에 `--wrapping-keys` 파라미터를 지정해야 합니다. 파라미터 값에는 key 및 provider 속성이 있어야 합니다.

동일한 명령에 여러 `--wrapping-keys` 파라미터 및 여러 key 속성을 포함할 수 있습니다.

주요 파라미터 속성 래핑

`--wrapping-keys` 파라미터 값은 다음 속성 및 해당 값으로 구성됩니다. 모든 암호화 명령에는 `--wrapping-keys`(또는 `--master-keys`) 파라미터가 필요합니다. 버전 2.1.x부터 복호화 시 `--wrapping-keys` 파라미터도 필요합니다.

속성 이름 또는 값에 공백이나 특수 문자가 포함된 경우 이름과 값을 모두 인용 부호로 묶습니다. 예를 들어 `--wrapping-keys key=12345 "provider=my cool provider"`입니다.

Key: 래핑 키 지정

key 속성을 사용하여 래핑 키를 식별합니다. 암호화할 때 값은 마스터 키 공급자가 인식하는 모든 키 식별자일 수 있습니다.

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

암호화 명령에는 하나 이상의 key 속성 및 값을 포함해야 합니다. 여러 래핑 키로 데이터 키를 암호화하려면 여러 key 속성을 사용합니다.

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

를 사용하는 암호화 명령에서 키 값은 키 ID AWS KMS keys, 키 ARN, 별칭 이름 또는 별칭 ARN일 수 있습니다. 예를 들어, 이 암호화 명령은 key 속성 값에 별칭 ARN을 사용합니다. 의 키 식별자에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 키 식별자를 AWS KMS key 참조하세요.

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

사용자 지정 마스터 키 공급자를 사용하는 복호화 명령에서 key 및 provider 속성이 필요합니다.

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

가 사용하는 복호화 명령에서 키 속성을 사용하여 복호화에 AWS KMS keys 사용할를 지정하거나 값이 인 검색 속성을 AWS KMS지정하여 AWS Encryption CLItrue가 메시지를 암호화하는 데 사용된 AWS KMS key 를 사용할 수 있습니다. 를 지정하는 경우 메시지를 암호화하는 데 사용되는 래핑 키 중 하나여야 AWS KMS key합니다.

래핑 키를 지정하는 것이 AWS Encryption SDK 모범 사례입니다. 이를 통해 사용하려는를 사용할 수 AWS KMS key 있습니다.

복호화 명령에서 key 속성 값은 키 ARN이어야 합니다.

```
\\ AWS KMS key
aws-encryption-cli --decrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```

검색: 복호화 AWS KMS key 시 사용

복호화 시 사용할 AWS KMS keys 를 제한할 필요가 없는 경우 값이 인 검색 속성을 사용할 수 있습니다`true`. 값이 이면 AWS Encryption CLI`true`가 메시지를 암호화 AWS KMS key 한를 사용하여 복호화할 수 있습니다. discovery 속성을 지정하지 않는 경우 discovery는 `false`(기본값)입니다. 검색 속성은 복호화 명령에서만 유효하며 메시지가 암호화된 경우에만 유효합니다 AWS KMS keys.

값이 `true`인 discovery 속성은 key 속성을 사용하여 AWS KMS keys를 지정하는 대신 사용할 수 있습니다. 로 암호화된 메시지를 복호화할 때 AWS KMS keys각 `--wrapping-keys` 파라미터에는 키 속성 또는 값이 `true`이지만 둘 다는 아닌 검색 속성이 있어야 합니다.

검색이 true인 경우 discovery-partition 및 discovery-account 속성을 사용하여 AWS 계정 지정한의 속성으로 AWS KMS keys 사용되는를 제한하는 것이 가장 좋습니다. 다음 예제에서 검색 속성을 사용하면 AWS Encryption CLI가 지정된 AWS KMS key 의를 사용할 수 있습니다 AWS 계정.

```
aws-encryption-cli --decrypt --wrapping-keys \
    discovery=true \
    discovery-partition=aws \
    discovery-account=111122223333 \
    discovery-account=444455556666
```

Provider: 마스터 키 공급자 지정

provider 속성은 마스터 키 공급자를 식별합니다. 기본값은 `aws-kms`이며, 이는 AWS KMS를 나타냅니다. 다른 마스터 키 공급자를 사용하는 경우 provider 속성이 필요합니다.

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

사용자 지정(비AWS KMS) 마스터 키 공급자 사용에 대한 자세한 내용은 AWS Encryption CLI 리포지토리의 README 파일에서 고급 구성 항목을 참조하세요.

리전: 지정 AWS 리전

리전 속성을 사용하여 AWS 리전 의를 지정합니다 AWS KMS key. 이 속성은 암호화 명령에서, 그리고 마스터 키 공급자가 AWS KMS인 경우에만 유효합니다.

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

AWS 암호화 CLI 명령은 ARN과 같은 리전 AWS 리전 이 포함된 경우 키 속성 값에 지정된를 사용합니다. 키 값이를 지정하면 AWS 리전리전 속성이 무시됩니다.

region 속성은 다른 리전 사양보다 우선합니다. 리전 속성을 사용하지 않는 경우 AWS 암호화 CLI 명령은 AWS CLI 명명된 프로필에 AWS 리전 지정된 또는 기본 프로필을 사용합니다.

Profile: 명명된 프로필 지정

profile 속성을 사용하여 AWS CLI 명명된 프로필을 지정합니다. 명명된 프로필에는 보안 인증 및 AWS 리전이 포함될 수 있습니다. 이 속성은 마스터 키 공급자가 AWS KMS인 경우에만 유효합니다.

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

profile 속성을 사용하여 암호화 및 복호화 명령에서 대체 보안 인증을 지정할 수 있습니다. 암호화 명령에서 AWS Encryption CLI는 키 값에 리전이 포함되지 않고 리전 속성이 없는 경우에만 명명된 프로파일 AWS 리전 에서를 사용합니다. 복호화 명령에서는 이름 프로필 AWS 리전 의이 무시됩니다.

여러 래핑 키를 지정하는 방법

각 명령에 여러 래핑 키(또는 마스터 키)를 지정할 수 있습니다.

래핑 키를 두 개 이상 지정하면 첫 번째 래핑 키가 데이터 암호화에 사용되는 데이터 키를 생성 및 암호화합니다. 다른 래핑 키는 동일한 데이터 키를 암호화합니다. 결과적으로 생성되는 암호화된 메시지에는 암호화된 데이터("사이퍼텍스트")와 암호화된 데이터 키 모음(각 래핑 키로 하나씩 암호화됨)이 포함됩니다. 모든 래핑은 암호화된 데이터 키 하나를 복호화한 다음 데이터를 복호화할 수 있습니다.

여러 래핑 키는 다음과 같이 두 가지 방법으로 지정할 수 있습니다.

- `--wrapping-keys` 파라미터 값에 여러 key 속성을 포함합니다.

```
$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
```

- 동일한 명령에 여러 `--wrapping-keys` 파라미터를 포함합니다. 지정한 속성 값이 명령의 일부 래핑 키에 적용되지 않는 경우 다음 구문을 사용합니다.

```
--wrapping-keys region=us-east-2 key=alias/test_key \
--wrapping-keys region=us-west-1 key=alias/test_key
```

값이 인 검색 속성을 `true` 사용하면 AWS 암호화 CLI가 메시지를 암호화 AWS KMS key 한 모든 를 사용할 수 있습니다. 동일한 명령에서 여러 `--wrapping-keys` 파라미터를 사용하는 경우 `--wrapping-keys` 파라미터의 `discovery=true`를 사용하면 다른 `--wrapping-keys` 파라미터에 있는 key 속성의 제한을 효과적으로 재정의할 수 있습니다.

예를 들어 다음 명령에서 첫 번째 `--wrapping-keys` 파라미터의 키 속성은 AWS Encryption CLI를 지정된 로 제한합니다 AWS KMS key. 그러나 두 번째 `--wrapping-keys` 파라미터의 검색 속성을 사용하면 AWS Encryption CLI가 지정된 계정 AWS KMS key 의를 사용하여 메시지를 해독할 수 있습니다.

```
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
```

## 입력을 제공하는 방법

Encryption CLI의 AWS 암호화 작업은 일반 텍스트 데이터를 입력으로 받아 암호화된 메시지를 반환합니다. 복호화 작업은 암호화된 메시지를 입력으로 받아 일반 텍스트 데이터를 반환합니다.

입력 위치를 AWS Encryption CLI에 알려주는 `--input` 파라미터(`-i`)는 모든 AWS Encryption CLI 명령에 필요합니다.

다음 방법 중 하나를 사용하여 입력을 제공할 수 있습니다.

- 파일을 사용합니다.

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

- 파일 이름 패턴을 사용합니다.

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

- 디렉터리 또는 디렉터리 이름 패턴을 사용합니다. 입력이 디렉터리인 경우 `--recursive` 파라미터(`-r`, `-R`)가 필요합니다.

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

- 입력을 명령(stdin)에 전달합니다. `-` 파라미터에 `--input` 값을 사용합니다. (`--input` 파라미터는 항상 필요합니다.)

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

## 출력 위치를 지정하는 방법

`--output` 파라미터는 AWS 암호화 또는 복호화 작업의 결과를 작성할 위치를 암호화 CLI에 알려줍니다. 이는 모든 AWS Encryption CLI 명령에 필요합니다. AWS Encryption CLI는 작업의 모든 입력 파일에 대해 새 출력 파일을 생성합니다.

출력 파일이 이미 있는 경우 기본적으로 AWS Encryption CLI는 경고를 인쇄한 다음 파일을 덮어씁니다. 덮어쓰기를 방지하려면, 덮어쓰기 전에 확인 메시지를 표시하는 `--interactive` 파라미터를 사용하거나, 출력으로 인해 덮어쓰기가 발생할 경우 입력을 건너뛰는 `--no-overwrite`를 사용합니다. 덮어쓰기 경고를 표시하지 않으려면 `--quiet`를 사용합니다. AWS Encryption CLI에서 오류 및 경고를 캡처하려면 `2>&1`리디렉션 연산자를 사용하여 출력 스트림에 기록합니다.

> **Note**
>
> 출력 파일을 덮어쓰는 명령은 출력 파일을 삭제하여 시작합니다. 명령이 실패하는 경우 출력 파일이 이미 삭제되었을 수 있습니다.

여러 가지 방법으로 출력 위치를 지정할 수 있습니다.

- 파일 이름을 지정합니다. 파일 경로를 지정하는 경우 명령이 실행되기 전에 경로에 있는 모든 디렉터리가 존재해야 합니다.

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

- 디렉터리를 지정합니다. 명령이 실행되기 전에 출력 디렉터리가 있어야 합니다.

  입력에 하위 디렉터리가 포함된 경우 명령은 지정된 디렉터리 아래의 하위 디렉터리를 다시 생성합니다.

  ```
  --output Test
  ```

  출력 위치가 디렉터리(파일 이름 없음)인 경우 AWS Encryption CLI는 입력 파일 이름과 접미사를 기반으로 출력 파일 이름을 생성합니다. 암호화 작업은 `.encrypted`를 입력 파일 이름에 추가하고 복호화 작업은 `.decrypted`를 추가합니다. 접미사를 변경하려면 `--suffix` 파라미터를 사용합니다.

  예를 들어, `file.txt`를 암호화하면 암호화 명령이 `file.txt.encrypted`를 생성합니다. `file.txt.encrypted`를 복호화하면 복호화 명령이 `file.txt.encrypted.decrypted`를 생성합니다.

- 명령줄(stdout)에 씁니다. `--output` 파라미터의 `-` 값을 입력합니다. `--output -`을 사용하여 출력을 다른 명령이나 프로그램으로 전달할 수 있습니다.

```
--output -
```

## 암호화 컨텍스트를 사용하는 방법

AWS 암호화 CLI를 사용하면 암호화 및 복호화 명령에서 암호화 컨텍스트를 제공할 수 있습니다. 필수는 아니지만 권장되는 암호화 모범 사례입니다.

암호화 컨텍스트는 비밀이 아닌 임의의 추가 인증 데이터 유형입니다. AWS Encryption CLI에서 암호화 컨텍스트는 `name=value` 페어 모음으로 구성됩니다. 파일에 대한 정보, 로그에서 암호화 작업을 찾는 데 도움이 되는 데이터 또는 허가나 정책에 필요한 데이터를 포함하여 페어로 구성된 모든 콘텐츠를 사용할 수 있습니다.

암호화 명령

암호화 명령에서 지정하는 암호화 컨텍스트와, CMM이 추가하는 추가 페어는 암호화된 데이터에 암호화 방식으로 바인딩됩니다. 또한 명령이 반환하는 암호화된 메시지에도 (일반 텍스트로) 포함됩니다. 를 사용하는 경우 AWS KMS key암호화 컨텍스트는와 같은 감사 레코드 및 로그에 일반 텍스트로 표시될 수도 있습니다 AWS CloudTrail.

다음 예제는 `name=value` 페어 3개로 구성된 암호화 컨텍스트를 보여줍니다.

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

복호화 명령

복호화 명령에서 암호화 컨텍스트는 암호화된 메시지를 올바르게 복호화하고 있는지 확인하는 데 도움이 됩니다.

암호화 시 암호화 컨텍스트를 사용한 경우에도 복호화 명령에 암호화 컨텍스트를 제공할 필요는 없습니다. 그러나 이렇게 하면 AWS 암호화 CLI는 복호화 명령의 암호화 컨텍스트에 있는 모든 요소가 암호화된 메시지의 암호화 컨텍스트에 있는 요소와 일치하는지 확인합니다. 요소가 일치하지 않으면 복호화 명령이 실패합니다.

예를 들어, 다음 명령은 암호화 컨텍스트에 `dept=IT`가 포함된 경우에만 암호화된 메시지를 복호화합니다.

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

암호화 컨텍스트는 보안 전략의 중요한 부분입니다. 그러나 암호화 컨텍스트를 선택할 때는 해당 값이 비밀이 아님을 기억해야 합니다. 암호화 컨텍스트에 기밀 데이터를 포함하지 마세요.

암호화 컨텍스트 지정

- 암호화 명령에서 `--encryption-context` 파라미터를 하나 이상의 `name=value` 페어와 함께 사용합니다. 공백을 사용하여 각 페어를 구분합니다.

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

- 복호화 명령의 `--encryption-context` 파라미터 값에는 `name=value` 페어, `name` 요소(값 없음) 또는 이 둘의 조합이 포함될 수 있습니다.

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

`name=value` 페어의 `name` 또는 `value`에 공백이나 특수 문자가 포함된 경우 전체 페어를 인용 부호로 묶습니다.

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

예를 들어, 이 암호화 명령에는 두 페어의 `purpose=test` 및 `dept=23`이 포함된 암호화 컨텍스트가 포함됩니다.

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

이러한 복호화 명령은 성공합니다. 각 명령의 암호화 컨텍스트는 원래 암호화 컨텍스트의 하위 집합입니다.

```
\\ 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 ...
```

하지만 이러한 복호화 명령은 실패합니다. 암호화된 메시지의 암호화 컨텍스트에는 지정된 요소가 포함되어 있지 않습니다.

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

### 커밋 정책을 지정하는 방법

명령에 대한 커밋 정책을 설정하려면 `--commitment-policy` 파라미터를 사용합니다. 이 파라미터는 버전 1.8.x에 도입되었습니다. 이는 암호화 및 복호화 명령에 유효합니다. 설정한 커밋 정책은 해당 정책이 나타나는 명령에만 유효합니다. 명령에 대한 커밋 정책을 설정하지 않으면 AWS Encryption CLI가 기본값을 사용합니다.

예를 들어 다음 파라미터 값은 커밋 정책을 `require-encrypt-allow-decrypt`로 설정합니다. 이 경우 커밋 정책은 항상 키 커밋으로 암호화하지만 키 커밋 사용 여부와 관계없이 암호화된 사이퍼텍스트는 복호화됩니다.

```
--commitment-policy require-encrypt-allow-decrypt
```

### 구성 파일에 파라미터를 저장하는 방법

자주 사용되는 AWS Encryption CLI 파라미터와 값을 구성 파일에 저장하여 시간을 절약하고 입력 오류를 방지할 수 있습니다.

구성 파일은 AWS Encryption CLI 명령의 파라미터와 값을 포함하는 텍스트 파일입니다. AWS Encryption CLI 명령에서 구성 파일을 참조하면 참조가 구성 파일의 파라미터 및 값으로 대체됩니다. 명령줄에 파일 내용을 입력한 경우에도 동일한 효과가 나타납니다. 구성 파일은 어떤 이름이든 가질 수 있으며 현재 사용자가 액세스할 수 있는 모든 디렉터리에 위치할 수 있습니다.

다음 예제 구성 파일인 `key.conf`는 서로 다른 리전에 AWS KMS keys 두 개를 지정합니다.

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

명령에서 구성 파일을 사용하려면 파일 이름 앞에 at 기호(@)를 붙입니다. PowerShell 콘솔에서는 백틱 문자를 사용하여 at 기호(`@)를 이스케이프 처리합니다.

이 예제 명령은 암호화 명령에서 `key.conf` 파일을 사용합니다.

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
```

### 구성 파일 규칙

구성 파일 사용 규칙은 다음과 같습니다.

- 각 구성 파일에 여러 파라미터를 포함하고 원하는 순서대로 나열할 수 있습니다. 각 파라미터를 해당 값(있는 경우)과 함께 별도의 줄에 나열합니다.
- #을 사용하여 줄 전체 또는 일부에 주석을 추가합니다.
- 다른 구성 파일에 대한 참조를 포함할 수 있습니다. PowerShell에서도 백틱을 사용하여 @ 기호를 이스케이프 처리하지 마세요.
- 구성 파일에서 따옴표를 사용하는 경우 인용된 텍스트는 여러 줄에 걸쳐 있을 수 없습니다.

예를 들어, 예제 `encrypt.conf` 파일의 내용은 다음과 같습니다.

```
# 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
```

명령에 여러 구성 파일을 포함할 수도 있습니다. 이 예제 명령은 `encrypt.conf` 및 `master-keys.conf` 구성 파일을 모두 사용합니다.

Bash

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

PowerShell

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

다음: AWS Encryption CLI 예제 사용해 보기

## AWS 암호화 CLI의 예

다음 예제를 사용하여 원하는 플랫폼에서 AWS 암호화 CLI를 사용해 보세요. 마스터 키 및 기타 파라미터에 대한 도움말은 AWS Encryption CLI 사용 방법 섹션을 참조하세요. 빠른 참조는 AWS Encryption SDK CLI 구문 및 파라미터 참조 섹션을 참조하세요.

> **Note**
>
> 다음 예제에서는 AWS Encryption CLI 버전 2.1.x의 구문을 사용합니다.
> 새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

암호화된 데이터 키를 제한하는 보안 기능을 사용하는 방법을 보여주는 예제는 암호화된 데이터 키 제한 섹션을 참조하세요.

AWS KMS 다중 리전 키를 사용하는 방법을 보여주는 예는 섹션을 참조하세요다중 리전 사용 AWS KMS keys.

주제

- 파일 암호화
- 파일 복호화
- 디렉터리의 모든 파일 암호화
- 디렉터리의 모든 파일 복호화
- 명령줄에서 암호화 및 복호화
- 여러 마스터 키 사용
- 스크립트의 암호화 및 복호화

- 데이터 키 캐싱 사용

## 파일 암호화

이 예제에서는 AWS Encryption CLI를 사용하여 "Hello World" 문자열이 포함된 `hello.txt` 파일의 내용을 암호화합니다.

파일에서 암호화 명령을 실행하면 AWS Encryption CLI는 파일의 콘텐츠를 가져오고, 고유한 데이터 키를 생성하고, 데이터 키 아래의 파일 콘텐츠를 암호화한 다음 암호화된 메시지를 새 파일에 씁니다.

첫 번째 명령은의 키 ARN을 `$keyArn` 변수 AWS KMS key 에 저장합니다. 를 사용하여 암호화할 때 키 ID AWS KMS key, 키 ARN, 별칭 이름 또는 별칭 ARN을 사용하여 식별할 수 있습니다. 의 키 식별자에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 키 식별자를 AWS KMS key참조하세요.

두 번째 명령은 파일 내용을 암호화합니다. 이 명령은 `--encrypt` 파라미터를 사용하여 작업을 지정하고 `--input` 파라미터를 사용하여 암호화할 파일을 표시합니다. `--wrapping-keys` 파라미터와 필요한 키 속성은 키 ARN으로 AWS KMS key 표시되는를 사용하도록 명령에 지시합니다.

이 명령은 `--metadata-output` 파라미터를 사용하여 암호화 작업에 대한 메타데이터를 위한 텍스트 파일을 지정합니다. 명령이 `--encryption-context` 파라미터를 사용하여 암호화 컨텍스트를 지정하는 것이 모범 사례입니다.

또한 이 명령은 `--commitment-policy` 파라미터를 사용하여 커밋 정책을 명시적으로 설정합니다. 버전 1.8.x에서는 `--wrapping-keys` 파라미터를 사용할 때 이 파라미터가 필요합니다. 버전 2.1.x부터 `--commitment-policy` 파라미터는 선택 사항이지만 권장됩니다.

`--output` 파라미터의 값인 점(.)은 출력 파일을 현재 디렉터리에 쓰도록 명령에 지시합니다.

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 --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 `
                           --output .
```

암호화 명령이 성공하면 어떤 출력도 반환하지 않습니다. 명령의 성공 여부를 확인하려면 $? 변수의 부울 값을 확인합니다. 명령이 성공하면 $?의 값은 0(Bash) 또는 True(PowerShell)입니다. 명령이 실패하면 $?의 값은 0이 아니거나(Bash) False(PowerShell)입니다.

Bash

```
$ echo $?
0
```

PowerShell

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

디렉터리 목록 명령을 사용하여 암호화 명령이 hello.txt.encrypted라는 새 파일을 생성했는지 확인할 수도 있습니다. 암호화 명령이 출력의 파일 이름을 지정하지 않았기 때문에 AWS Encryption CLI는 입력 파일과 이름이 같고 접.encrypted미사가 있는 파일에 출력을 작성했습니다. 다른 접미사를 사용하거나 접미사를 숨기려면 --suffix 파라미터를 사용합니다.

hello.txt.encrypted 파일에는 hello.txt 파일의 사이퍼텍스트, 데이터 키의 암호화된 사본, 추가 메타데이터(암호화 컨텍스트 포함)가 포함된 암호화된 메시지가 들어 있습니다.

Bash

```
$  ls
```

```
hello.txt  hello.txt.encrypted
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

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

## 파일 복호화

이 예제에서는 AWS Encryption CLI를 사용하여 이전 예제에서 암호화된 `Hello.txt.encrypted` 파일의 내용을 해독합니다.

복호화 명령은 `--decrypt` 파라미터를 사용하여 작업을 표시하고 `--input` 파라미터를 사용하여 복호화할 파일을 식별합니다. `--output` 파라미터의 값은 현재 디렉터리를 나타내는 점입니다.

key 속성이 있는 `--wrapping-keys` 파라미터는 암호화된 메시지를 복호화하는 데 사용되는 래핑 키를 지정합니다. 를 사용한 복호화 명령에서 키 속성 AWS KMS keys의 값은 키 ARN이어야 합니다. `--wrapping-keys` 파라미터는 복호화 명령에 반드시 필요합니다. AWS KMS keys를 사용하는 경우 key 속성을 사용하여 복호화를 위해 AWS KMS keys 를 지정하거나 discovery 속성을 `true` 값으로 지정할 수 있습니다(둘 다 지정할 수는 없음). 사용자 지정 마스터 키 공급자를 사용하는 경우 key 속성과 provider 속성이 필요합니다.

버전 2.1.x부터 `--commitment-policy` 파라미터는 선택 사항이지만 권장됩니다. 이를 명시적으로 사용하면 기본값인 `require-encrypt-require-decrypt`를 지정하더라도 의도를 명확히 알 수 있습니다.

암호화 명령에 암호화 컨텍스트가 제공된 경우에도 `--encryption-context` 파라미터는 복호화 명령에서 선택 사항입니다. 이 경우 복호화 명령은 암호화 명령에 제공된 것과 동일한 암호화 컨텍스트를 사용합니다. 암호화 해제 전에 AWS 암호화 CLI는 암호화된 메시지의 암호화 컨텍스트에 `purpose=test` 페어가 포함되어 있는지 확인합니다. 그러지 않으면 복호화 명령이 실패합니다.

`--metadata-output` 파라미터는 복호화 작업에 대한 메타데이터를 위한 파일을 지정합니다. `--output` 파라미터의 값인 점(.)은 출력 파일을 현재 디렉터리에 씁니다.

암호화된 데이터 키가 너무 많은 잘못된 형식의 메시지를 복호화하지 않도록 `--max-encrypted-data-keys` 파라미터를 사용하는 것이 모범 사례입니다. 암호화된 데이터 키의 예상 수(암호화에 사용되는 각 래핑 키당 하나) 또는 합리적인 최대값(예: 5개)을 지정합니다. 자세한 내용은 암호화된 데이터 키 제한을 참조하세요.

`--buffer`는 디지털 서명이 있는지 확인하는 등 모든 입력이 처리된 후에만 일반 텍스트를 반환합니다.

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 .
```

복호화 명령이 성공하면 어떤 출력도 반환하지 않습니다. 명령의 성공 여부를 확인하려면 `$?` 변수의 값을 가져옵니다. 디렉터리 목록 명령을 사용하여 명령이 `.decrypted`라는 접미사가 붙은 새 파일

을 생성했는지 확인할 수도 있습니다. 일반 텍스트 콘텐츠를 보려면 파일 콘텐츠를 가져오는 명령(예: `cat` 또는 Get-Content)을 사용합니다.

Bash

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

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

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

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


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

## 디렉터리의 모든 파일 암호화

이 예제에서는 AWS Encryption CLI를 사용하여 디렉터리에 있는 모든 파일의 내용을 암호화합니다.

명령이 여러 파일에 영향을 미치는 경우 AWS Encryption CLI는 각 파일을 개별적으로 처리합니다. 파일 내용을 가져오고, 마스터 키에서 파일의 고유한 데이터 키를 가져오고, 데이터 키로 파일 내용을 암호화하고, 결과를 출력 디렉터리의 새 파일에 씁니다. 따라서 출력 파일을 독립적으로 복호화할 수 있습니다.

이 `TestDir` 디렉터리 목록에는 암호화하려는 일반 텍스트 파일이 표시됩니다.

Bash

```
$  ls testdir
```

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

PowerShell

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

    Directory: C:\TestDir

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

첫 번째 명령은의 Amazon 리소스 이름(ARN)을 $keyArn 변수 AWS KMS key 에 저장합니다.

두 번째 명령은 TestDir 디렉터리에 있는 파일의 콘텐츠를 암호화하고 암호화된 콘텐츠의 파일을 TestEnc 디렉터리에 씁니다. TestEnc 디렉터리가 존재하지 않으면 명령이 실패합니다. 입력 위치가 디렉터리이므로 --recursive 파라미터가 필요합니다.

--wrapping-keys 파라미터와 필수 key 속성은 사용할 래핑 키를 지정합니다. 암호화 명령에는 암호화 컨텍스트인 dept=IT가 포함됩니다. 여러 파일을 암호화하는 명령에 암호화 컨텍스트를 지정하면 모든 파일에 동일한 암호화 컨텍스트가 사용됩니다.

명령에는 암호화 작업에 대한 메타데이터를 작성할 위치를 AWS Encryption CLI에 알려주는 --metadata-output 파라미터도 있습니다. AWS Encryption CLI는 암호화된 각 파일에 대해 하나의 메타데이터 레코드를 씁니다.

--commitment-policy parameter는 버전 2.1.x부터 선택 사항이지만 권장됩니다. 명령이나 스크립트가 사이퍼텍스트를 복호화할 수 없어 실패하는 경우 명시적 커밋 정책 설정을 사용하면 문제를 빠르게 감지하는 데 도움이 될 수 있습니다.

명령이 완료되면 AWS Encryption CLI는 암호화된 파일을 TestEnc 디렉터리에 기록하지만 출력을 반환하지 않습니다.

마지막 명령은 TestEnc 디렉터리의 파일을 나열합니다. 일반 텍스트 콘텐츠의 각 입력 파일마다 암호화된 콘텐츠의 출력 파일이 하나씩 있습니다. 명령이 대체 접미사를 지정하지 않았으므로 암호화 명령이 각 입력 파일의 이름에 .encrypted를 추가했습니다.

### 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\
                     --wrapping-keys key=$keyArn \
                     --encryption-context dept=IT \
                     --commitment-policy require-encrypt-require-decrypt \
                     --metadata-output ~/metadata \
                     --output testenc

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

### PowerShell

```
# To run this example, replace the fictitious 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 `
                           --input .\TestDir --recursive `
                           --wrapping-keys key=$keyArn `
                           --encryption-context dept=IT `
                           --commitment-policy require-encrypt-require-decrypt `
                           --metadata-output .\Metadata\Metadata.txt `
                           --output .\TestEnc

PS C:\> dir .\TestEnc

    Directory: C:\TestEnc

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

## 디렉터리의 모든 파일 복호화

이 예제는 디렉터리에 있는 모든 파일을 복호화합니다. 이전 예제에서 암호화된 TestEnc 디렉터리의 파일부터 시작합니다.

Bash

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

PowerShell

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

    Directory: C:\TestEnc

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

이 복호화 명령은 TestEnc 디렉터리의 모든 파일을 복호화하고 일반 텍스트 파일을 TestDec 디렉터리에 씁니다. 키 속성과 키 ARN 값이 있는 `--wrapping-keys` 파라미터는 AWS 파일을 복호화 AWS KMS keys 하는 데 사용할 암호화 CLI에 지시합니다. 명령은 `--interactive` 파라미터를 사용하여 동일한 이름의 파일을 덮어쓰기 전에 AWS Encryption CLI에 프롬프트를 표시하도록 지시합니다.

또한 이 명령은 파일이 암호화될 때 제공된 암호화 컨텍스트를 사용합니다. 여러 파일을 복호화할 때 AWS 암호화 CLI는 모든 파일의 암호화 컨텍스트를 확인합니다. 파일에 대한 암호화 컨텍스트 확인이 실패하면 AWS Encryption CLI는 파일을 거부하고 경고를 작성하고 메타데이터에 실패를 기록한 다음 나머지 파일을 계속 확인합니다. AWS Encryption CLI가 다른 이유로 파일을 복호화하지 못하면 전체 복호화 명령이 즉시 실패합니다.

이 예제에서는 모든 입력 파일의 암호화된 메시지에 `dept=IT` 암호화 컨텍스트 요소가 포함되어 있습니다. 하지만 암호화 컨텍스트가 다른 메시지를 복호화하는 경우에도 암호화 컨텍스트의 일부는 확인 가능할 수도 있습니다. 예를 들어 일부 메시지의 암호화 컨텍스트가 `dept=finance`이고 다른 메시지의 암호화된 컨텍스트가 `dept=IT`인 경우 값을 지정하지 않고도 암호화 컨텍스트에 항상 `dept` 이름이 포함되어 있는지 확인할 수 있습니다. 좀 더 구체적으로 확인하려면 별도의 명령으로 파일을 복호화할 수 있습니다.

복호화 명령은 어떤 출력도 반환하지 않지만 디렉터리 목록 명령을 사용하여 복호화 명령이 `.decrypted` 접미사가 붙은 새 파일을 만들었는지 확인할 수 있습니다. 일반 텍스트 콘텐츠를 보려면 파일 콘텐츠를 가져오는 명령을 사용합니다.

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

```
# 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
```

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

## 명령줄에서 암호화 및 복호화

이 예제는 입력을 명령(stdin)으로 파이프하고 출력을 명령줄(stdout)에 쓰는 방법을 보여줍니다. 명령에서 stdin 및 stdout을 표현하는 방법과, 기본 제공 Base64 인코딩 도구를 사용하여 쉘이 ASCII가 아닌 문자를 잘못 해석하지 않도록 하는 방법을 설명합니다.

이 예제에서는 일반 텍스트 문자열을 암호화 명령으로 파이프하고 암호화된 메시지를 변수에 저장합니다. 그런 다음 변수에 있는 암호화된 메시지를 복호화 명령으로 파이프하고, 해당 명령은 출력을 파이프라인(stdout)에 씁니다.

이 예제는 다음과 같은 세 가지 명령으로 구성되어 있습니다.

- 첫 번째 명령은의 키 ARN을 `$keyArn` 변수 AWS KMS key 에 저장합니다.

  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'
  ```

- 두 번째 명령은 `Hello World` 문자열을 암호화 명령으로 파이프하고 그 결과를 `$encrypted` 변수에 저장합니다.

  `--input` 및 `--output` 파라미터는 모든 AWS Encryption CLI 명령에 반드시 필요합니다. 입력이 명령(stdin)으로 파이프되고 있음을 나타내려면 `--input` 파라미터의 값에 하이픈(-)을 사용합니다. 출력을 명령줄(stdout)로 보내려면 `--output` 파라미터의 값에 하이픈을 사용합니다.

`--encode` 파라미터는 출력을 반환하기 전에 출력을 Base64 인코딩합니다. 이렇게 하면 쉘이 암호화된 메시지의 ASCII가 아닌 문자를 잘못 해석하는 것을 방지할 수 있습니다.

이 명령은 개념 증명일 뿐이므로 암호화 컨텍스트는 생략하고 메타데이터(`-S`)는 표시하지 않습니다.

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
```

- 세 번째 명령은 `$encrypted` 변수의 암호화된 메시지를 복호화 명령으로 파이프합니다.

  이 복호화 명령은 `--input -`을 사용하여 입력이 파이프라인(stdin)에서 들어오고 있음을 나타내고 `--output -`을 사용하여 출력을 파이프라인(stdout)으로 보냅니다. (입력 파라미터는 실제 입력 바이트가 아니라 입력 위치를 사용하므로 `$encrypted` 변수를 `--input` 파라미터의 값으로 사용할 수 없습니다.)

  이 예제에서는 `--wrapping-keys` 파라미터의 검색 속성을 사용하여 AWS Encryption CLI가를 사용하여 데이터를 복호화 AWS KMS key 하도록 허용합니다. 여기서는 커밋 정책을 지정하지 않으므로 버전 2.1.x 이상에 대한 기본값인 `require-encrypt-require-decrypt`를 사용합니다.

  출력이 암호화된 후 인코딩되었으므로 복호화 명령은 `--decode` 파라미터를 사용하여 Base64로 인코딩된 입력을 복호화하기 전에 디코딩합니다. 또한 `--decode` 파라미터를 사용하여 Base64로 인코딩된 입력을 암호화하기 전에 디코딩할 수 있습니다.

  앞서 말했듯이, 이 명령은 암호화 컨텍스트를 생략하고 메타데이터(`-S`)를 표시하지 않습니다.

Bash

```
$  echo $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=true
 --input - --output - --decode --buffer -S
Hello World
```

PowerShell

```
PS C:\> $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=$true
 --input - --output - --decode --buffer -S
Hello World
```

중간 변수 없이 단일 명령으로 암호화 및 복호화 작업을 수행할 수도 있습니다.

이전 예제에서처럼 `--input` 및 `--output` 파라미터에는 `-` 값이 있으며 명령은 `--encode` 파라미터를 사용하여 출력을 인코딩하고 `--decode` 파라미터를 사용하여 입력을 디코딩합니다.

Bash

```
$  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
```

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
```

## 여러 마스터 키 사용

이 예제에서는 AWS Encryption CLI에서 데이터를 암호화하고 해독할 때 여러 마스터 키를 사용하는 방법을 보여줍니다.

여러 마스터 키를 사용하여 데이터를 암호화하면 해당 마스터 키 중 하나를 사용하여 데이터를 복호화할 수 있습니다. 이 전략을 사용하면 마스터 키 중 하나를 사용할 수 없는 상황에서도 데이터를 복호화할 수 있습니다. 암호화된 데이터를 여러에 저장하는 경우 AWS 리전이 전략을 사용하면 동일한 리전의 마스터 키를 사용하여 데이터를 해독할 수 있습니다.

여러 마스터 키로 암호화하는 경우 첫 번째 마스터 키가 특별한 역할을 합니다. 이 키는 데이터를 암호화하는 데 사용되는 데이터 키를 생성합니다. 나머지 마스터 키는 일반 텍스트 데이터 키를 암호화합니다. 그 결과, 암호화된 메시지에는 암호화된 데이터와 암호화된 데이터 키 모음이 각 마스터 키마다 하나씩 포함됩니다. 데이터 키를 만든 것은 첫 번째 마스터 키이지만, 다른 모든 마스터 키로도 데이터 키를 복호화하여 데이터를 복호화할 수 있습니다.

세 개의 마스터 키를 사용한 암호화

이 예제 명령은 세 개의 래핑 키를 사용하여 세 개의 AWS 리전각각에 하나씩 `Finance.log` 파일을 암호화합니다.

이 명령은 암호화된 메시지를 `Archive` 디렉터리에 씁니다. 이 명령은 값이 없는 `--suffix` 파라미터를 사용하여 접미사를 표시하지 않으므로 입력 및 출력 파일 이름이 동일합니다.

이 명령은 세 가지 key 속성을 가진 `--wrapping-keys` 파라미터를 사용합니다. 같은 명령에 여러 개의 `--wrapping-keys` 파라미터를 사용할 수도 있습니다.

로그 파일을 암호화하기 위해 AWS Encryption CLI는 목록의 첫 번째 래핑 키인 `$key1`에 데이터를 암호화하는 데 사용하는 데이터 키를 생성하도록 요청합니다. 그런 다음 나머지 래핑 키를 각각 사용하여 동일한 데이터 키의 일반 텍스트 사본을 암호화합니다. 출력 파일의 암호화된 메시지에는 암호화된 데이터 키 세 개가 모두 포함됩니다.

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

```
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
```

이 명령은 `Finance.log` 파일의 암호화된 사본을 복호화하여 `Finance` 디렉터리의 `Finance.log.clear` 파일에 씁니다. 세 개로 암호화된 데이터를 해독하려면 동일한 세 AWS KMS keys 개 또는 하위 집합을 지정할 AWS KMS keys수 있습니다. 이 예제에서는 AWS KMS keys중 하나만 지정합니다.

데이터 복호화에 AWS KMS keys 사용할 AWS 암호화 CLI에 알리려면 `--wrapping-keys` 파라미터의 키 속성을 사용합니다. 를 사용하여 복호화할 때 키 속성 AWS KMS keys의 값은 키 ARN이어야 합니다.

AWS KMS keys 지정한에서 Decrypt API를 호출할 수 있는 권한이 있어야 합니다. 자세한 정보는 AWS KMS에 대한 인증 및 액세스 제어를 참조하세요.

모범 사례로서, 이 예제는 암호화된 데이터 키가 너무 많은 잘못된 형식의 메시지를 복호화하지 않도록 `--max-encrypted-data-keys` 파라미터를 사용합니다. 이 예제에서는 복호화에 래핑 키 하나만 사용하지만, 암호화된 메시지에는 암호화할 때 사용되는 세 개의 래핑 키에 대해 각각 하나씩 총 세 개의 암호화된 데이터 키가 있습니다. 암호화된 데이터 키의 예상 개수 또는 적절한 최대값(예: 5)을 지정합니다. 최대값을 3보다 작게 지정하면 명령이 실패합니다. 자세한 내용은 암호화된 데이터 키 제한을 참조하세요.

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
```

## 스크립트의 암호화 및 복호화

이 예제는 스크립트에서 AWS 암호화 CLI를 사용하는 방법을 보여줍니다. 데이터 암호화 및 복호화만 하는 스크립트를 작성하거나 데이터 관리 프로세스의 일부로 암호화 또는 복호화하는 스크립트를 작성할 수 있습니다.

이 예제에서 스크립트는 로그 파일 모음을 가져와 압축하고 암호화한 다음 암호화된 파일을 Amazon S3 버킷에 복사합니다. 이 스크립트는 각 파일을 개별적으로 처리하므로 사용자는 파일을 독립적으로 복호화하고 확장할 수 있습니다.

파일을 압축하고 암호화할 때는 반드시 암호화하기 전에 압축해야 합니다. 올바르게 암호화된 데이터는 압축할 수 없습니다.

> **Warning**
>
> 악의적인 공격자가 제어할 수도 있는 비밀 키 및 데이터가 모두 포함된 데이터를 압축할 때는 주의해야 합니다. 압축된 데이터의 최종 크기로 인해 의도치 않게 데이터 내용에 대한 민감한 정보가 드러날 수 있습니다.

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,

    [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 {
    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)
                    {
                        Write-S3Object -BucketName $S3Bucket -File
 "$EncryptDirectory\$fileName.zip.encrypted" -Key "$S3BucketFolder/
$fileName.zip.encrypted"

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

# 데이터 키 캐싱 사용

이 예제에서는 많은 수의 파일을 암호화하는 명령에 데이터 키 캐싱을 사용합니다.

기본적으로 AWS Encryption CLI(및 기타 버전의 AWS Encryption SDK)는 암호화하는 각 파일에 대해 고유한 데이터 키를 생성합니다. 각 작업에 고유한 데이터 키를 사용하는 것이 암호화 모범 사례이긴 하지만, 일부 상황에서는 데이터 키를 제한적으로 재사용할 수 있습니다. 데이터 키 캐싱을 고려 중인 경우 보안 엔지니어에게 문의하여 애플리케이션의 보안 요구 사항을 파악하고 적합한 보안 임계값을 결정하세요.

이 예제에서는 데이터 키 캐싱으로 마스터 키 공급자에 대한 요청 빈도를 줄여서 암호화 작업의 속도를 높입니다.

이 예제의 명령은 총 약 800개의 작은 로그 파일이 포함된 여러 개의 하위 디렉터리가 있는 큰 디렉터리를 암호화합니다. 첫 번째 명령은 AWS KMS key 의 ARN을 keyARN 변수에 저장합니다. 두 번째 명령은 입력 디렉터리의 모든 파일을 (재귀적으로) 암호화하여 아카이브 디렉터리에 씁니다. 이 명령은 `--suffix` 파라미터를 사용하여 `.archive` 접미사를 지정합니다.

`--caching` 파라미터는 데이터 키 캐싱 사용을 설정합니다. 직렬 파일 처리는 한 번에 두 개 이상의 데이터 키를 사용하지 않기 때문에 캐시의 데이터 키 수를 제한하는 capacity 속성은 1로 설정됩니다. 캐시된 데이터 키를 사용할 수 있는 기간을 결정하는 max_age 속성은 10초로 설정됩니다.

선택 사항인 max_messages_crypted 속성은 메시지 10개로 설정되므로 하나의 데이터 키는 10개 이상의 파일을 암호화하는 데 사용되지 않습니다. 각 데이터 키로 암호화되는 파일 수를 제한하면 혹시라도 데이터 키가 손상되는 예기치 못한 상황이 발생하더라도 영향을 받는 파일 수를 줄일 수 있습니다.

운영 체제에서 생성하는 로그 파일에서 이 명령을 실행하려면 관리자 권한(Linux의 경우 sudo, Windows의 경우 관리자 권한으로 실행)이 필요할 수 있습니다.

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

```
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
```

이 예제에서는 데이터 키 캐싱의 효과를 테스트하기 위해 PowerShell의 Measure-Command cmdlet을 사용합니다. 이 예제를 데이터 키 캐싱 없이 실행하면 완료하는 데 약 25초가 걸립니다. 이 프로세스는 디렉터리의 각 파일에 대해 새 데이터 키를 생성합니다.

```
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 }


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

데이터 키 캐싱을 사용하면 각 데이터 키를 최대 10개 파일로 제한하는 경우에도 프로세스가 더 빨라집니다. 이제 이 명령은 완료하는 데 12초 미만이 소요되며 마스터 키 공급자에 대한 호출 수를 원래 값의 1/10로 줄입니다.

```
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
TotalMinutes      : 0.196887733333333
TotalSeconds      : 11.813264
TotalMilliseconds : 11813.264
```

max_messages_encrypted 제한을 제거하면 모든 파일이 동일한 데이터 키로 암호화됩니다. 이렇게 변경하면 프로세스가 훨씬 빨라지지는 않고 데이터 키를 재사용할 위험이 증가합니다. 그러나 마스터 키 공급자에 대한 호출 횟수는 1로 줄어듭니다.

```
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}


Days              : 0
```

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

# AWS Encryption SDK CLI 구문 및 파라미터 참조

이 주제는 AWS Encryption SDK Command Line Interface(CLI)를 사용하는 데 도움이 될 구문 다이어그램 및 간단한 파라미터 설명을 제공합니다. 래핑 키 및 기타 파라미터 관련 도움말은 AWS Encryption CLI 사용 방법 섹션을 참조하세요. 예제는 AWS 암호화 CLI의 예 섹션을 참조하세요. 전체 설명서는 문서 읽기를 참조하세요.

주제

- AWS 암호화 CLI 구문
- AWS 암호화 CLI 명령줄 파라미터
- 고급 파라미터

## AWS 암호화 CLI 구문

이러한 AWS Encryption CLI 구문 다이어그램은 AWS Encryption CLI로 수행하는 각 작업의 구문을 보여줍니다. 이는 AWS Encryption CLI 버전 2.1.x 이상에서 권장되는 구문을 나타냅니다.

새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

> **Note**
>
> 파라미터 설명에 명시되어 있지 않는 한, 각 파라미터 또는 속성은 각 명령에서 한 번만 사용할 수 있습니다.
> 파라미터가 지원하지 않는 속성을 사용하는 경우 AWS Encryption CLI는 경고나 오류 없이 지원되지 않는 속성을 무시합니다.

### 지원 받기

파라미터 설명과 함께 전체 AWS Encryption CLI 구문을 가져오려면 `--help` 또는를 사용합니다`-h`.

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

### 버전 가져오기

AWS Encryption CLI 설치의 버전 번호를 가져오려면를 사용합니다`--version`. 질문을 하거나, 문제를 보고하거나, AWS 암호화 CLI 사용에 대한 팁을 공유할 때는 버전을 포함해야 합니다.

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

### Encrypt data

다음 구문 다이어그램은 encrypt 명령에 사용되는 파라미터를 보여줍니다.

```
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]
```

### 데이터 복호화

다음 구문 다이어그램은 decrypt 명령에 사용되는 파라미터를 보여줍니다.

버전 1.8.x부터 복호화 시 `--wrapping-keys` 파라미터는 선택 사항이지만 권장됩니다. 버전 2.1.x부터 암호화 및 복호화 시 `--wrapping-keys` 파라미터가 필요합니다. AWS KMS keys의 경우 key 속성을 사용하여 래핑 키를 지정(모범 사례)하거나 discovery 속성을 `true`로 설정할 수 있습니다. 그러면 AWS Encryption CLI에서 사용할 수 있는 래핑 키가 제한되지 않습니다.

```
aws-encryption-cli --decrypt (or [--decrypt-unsigned])
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite]  [--suffix
 [<suffix>]] [--encode]
                   --wrapping-keys  [--wrapping-keys] ...
                       [key=<keyID>] [key=<keyID>] ...
                       [discovery={true|false}] [discovery-partition=<aws-partition-
name> discovery-account=<aws-account-ID> [discovery-account=<aws-account-ID>] ...]
                       [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>
 ...]]
                   [--buffer]
                   [--max-encrypted-data-keys <integer>]
                   [--caching <attributes>]
                   [--max-length <length>]
                   [-v | -vv | -vvv | -vvvv]
                   [--quiet]
```

### 구성 파일 사용

파라미터와 해당 값이 포함된 구성 파일을 참조할 수 있습니다. 이는 명령에 파라미터와 값을 입력하는 것과 같습니다. 예제는 구성 파일에 파라미터를 저장하는 방법 섹션을 참조하세요.

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

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

# AWS 암호화 CLI 명령줄 파라미터

이 목록은 AWS Encryption CLI 명령 파라미터에 대한 기본 설명을 제공합니다. 전체 설명은 aws-encryption-sdk-cli 설명서를 참조하세요.

--encrypt (-e)

입력 데이터를 암호화합니다. 모든 명령에는 `--encrypt`, `--decrypt` 또는 `--decrypt-unsigned` 파라미터가 있어야 합니다.

--decrypt (-d)

입력 데이터를 복호화합니다. 모든 명령에는 `--encrypt`, `--decrypt` 또는 `--decrypt-unsigned` 파라미터가 있어야 합니다.

--decrypt-unsigned[버전 1.9.x 및 2.2.x에 도입됨]

`--decrypt-unsigned` 파라미터는 사이퍼텍스트를 복호화하고, 복호화 전에 메시지가 서명되지 않도록 합니다. `--algorithm` 파라미터를 사용하고 디지털 서명이 없는 알고리즘 제품군을 선택하여 데이터를 암호화한 경우 이 파라미터를 사용합니다. 사이퍼텍스트가 서명된 경우 복호화가 실패합니다.

`--decrypt` 또는 `--decrypt-unsigned`를 복호화에 사용할 수 있지만 둘 다 사용할 수는 없습니다.

--wrapping-keys (-w)[버전 1.8.x에 도입됨]

암호화 및 복호화 작업에 사용되는 래핑 키(또는 마스터 키)를 지정합니다. 각 명령에서 여러 `--wrapping-keys` 파라미터를 사용할 수 있습니다.

버전 2.1.x부터는 `--wrapping-keys` 파라미터가 암호화 및 복호화 명령에 필요합니다. 버전 1.8.x에서 암호화 명령에는 `--wrapping-keys` 또는 `--master-keys` 파라미터가 필요합니다. 버전 1.8.x에서 복호화 명령의 경우 `--wrapping-keys` 파라미터는 선택 사항이지만 권장됩니다.

사용자 지정 마스터 키 공급자를 사용하는 경우 암호화 및 복호화 명령에는 key 및 provider 속성이 필요합니다. 를 사용할 때 AWS KMS keys암호화 명령에는 키 속성이 필요합니다. 복호화 명령에는 값이 `true`인 key 속성 또는 discovery 속성이 필요합니다(둘 다는 아님). 복호화 시 key 속성을 사용하는 것이 AWS Encryption SDK 모범 사례입니다. 이는 Amazon S3 버킷 또는 Amazon SQS 대기열에 있는 메시지와 같이 익숙하지 않은 메시지를 일괄 복호화하는 경우 특히 중요합니다.

AWS KMS 다중 리전 키를 래핑 키로 사용하는 방법을 보여주는 예는 섹션을 참조하세요다중 리전 사용 AWS KMS keys.

속성: `--wrapping-keys` 파라미터의 값은 다음 속성으로 구성됩니다. 형식은 `attribute_name=value`입니다.

key

작업에 사용된 래핑 키를 식별합니다. 형식은 key=ID 페어입니다. 각 `--wrapping-keys` 파라미터 값에 여러 key 속성을 지정할 수 있습니다.

- 암호화 명령: 모든 암호화 명령에는 key 속성이 반드시 필요합니다. 암호화 명령 AWS KMS key 에서를 사용하는 경우 키 속성의 값은 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN일 수 있습니다. AWS KMS 키 식별자에 대한 설명은 AWS Key Management Service 개발자 안내서의 키 식별자를 참조하세요.
- 복호화 명령: AWS KMS keys를 사용하여 복호화하는 경우 `--wrapping-keys` 파라미터에는 키 ARN 값이 있는 key 속성 또는 값이 `true`인 discovery 속성이 필요합니다(둘 다는 아님). key 속성을 사용하는 것이 AWS Encryption SDK 모범 사례입니다. 사용자 지정 마스터 키 공급자를 사용하여 복호화할 때는 key 속성이 반드시 필요합니다.

> **Note**
>
> 복호화 명령에서 AWS KMS 래핑 키를 지정하려면 키 속성의 값이 키 ARN이어야 합니다. 키 ID, 별칭 이름 또는 별칭 ARN을 사용하는 경우 AWS Encryption CLI는 래핑 키를 인식하지 못합니다.

각 `--wrapping-keys` 파라미터 값에 여러 key 속성을 지정할 수 있습니다. 그러나 `--wrapping-keys` 파라미터의 모든 provider, region 및 profile 속성은 해당 파라미터 값의 모든 래핑 키에 적용됩니다. 서로 다른 속성 값으로 래핑 키를 지정하려면 명령에서 여러 `--wrapping-keys` 파라미터를 사용합니다.

discovery

AWS Encryption CLI가 임의의를 사용하여 메시지를 복호화 AWS KMS key 하도록 허용합니다. discovery 값은 `true` 또는 `false`일 수 있습니다. 기본값은 `false`입니다. discovery 속성은 복호화 명령에서, 그리고 마스터 키 공급자가 AWS KMS인 경우에만 유효합니다.

를 사용하여 복호화 AWS KMS keys할 때 `--wrapping-keys` 파라미터에는 키 속성 또는 값이 인 검색 속성이 필요합니다`true`(둘 다 아님). key 속성을 사용하는 경우 값이 `false`인 discovery 속성을 사용하여 검색을 명시적으로 거부할 수 있습니다.

- `False` (기본값) - 검색 속성이 지정되지 않았거나 값이 인 경우 `false` AWS Encryption CLI는 `--wrapping-keys` 파라미터의 키 속성으로 AWS KMS keys 지정된 만 사용하여 메시지

를 복호화합니다. discovery가 `false`일 때 key 속성을 지정하지 않으면 복호화 명령이 실패합니다. 이 값은 AWS 암호화 CLI 모범 사례를 지원합니다.

- `True` - 검색 속성의 값이 인 경우 `true` AWS Encryption CLI는 암호화된 메시지의 메타데이터 AWS KMS keys 에서를 가져오고 이를 사용하여 메시지를 AWS KMS keys 복호화합니다. 값이 인 검색 속성은 복호화 시 래핑 키를 지정할 수 없는 버전 1.8.x 이전의 AWS Encryption CLI 버전처럼 `true` 동작합니다. 그러나를 사용하려는 의도 AWS KMS key 는 명시적입니다. discovery가 `true`일 때 key 속성을 지정하면 복호화 명령이 실패합니다.

  이 `true` 값을 사용하면 AWS Encryption CLI가 서로 다른 AWS 계정 및 리전 AWS KMS keys 에서를 사용하거나 사용자에게 사용 권한이 없는 AWS KMS keys 를 사용하려고 할 수 있습니다.

검색이 인 경우 discovery-partition 및 discovery-account 속성을 사용하여 AWS 계정 지정한의 속성으로 AWS KMS keys 사용되는를 제한하는 `true`것이 가장 좋습니다.

discovery-account

복호화에 AWS KMS keys 사용되는를 지정된의 로 제한합니다 AWS 계정. 이 속성에 사용할 수 있는 유일한 값은 AWS 계정 ID입니다.

이 속성은 선택 사항이며 검색 속성 AWS KMS keys 이 로 설정`true`되고 검색 파티션 속성이 지정된 복호화 명령에서만 유효합니다.

각 discovery-account 속성은 하나의 AWS 계정 ID만 사용하지만 동일한 `--wrapping-keys` 파라미터에서 여러 discovery-account 속성을 지정할 수 있습니다. 지정된 `--wrapping-keys` 파라미터에 지정된 모든 계정은 지정된 AWS 파티션에 있어야 합니다.

discovery-partition

discovery-account 속성의 계정에 대한 AWS 파티션을 지정합니다. 값은 , `aws` `aws-cn`또는와 같은 AWS 파티션이어야 합니다`aws-gov-cloud`. 자세한 내용은 AWS 일반 참조의 Amazon 리소스 이름을 참조하세요.

이 속성은 discovery-account 속성을 사용할 때 필요합니다. 각 `--wrapping keys` 파라미터에는 discovery-partition 속성을 하나만 지정할 수 있습니다. 여러 파티션 AWS 계정 에서를 지정하려면 추가 `--wrapping-keys` 파라미터를 사용합니다.

provider

마스터 키 공급자를 식별합니다. 형식은 provider=ID 페어입니다. 기본값인 aws-kms는를 나타냅니다 AWS KMS. 이 속성은 마스터 키 공급자가가 아닌 경우에만 필요합니다 AWS KMS.

리전

AWS 리전 의를 식별합니다 AWS KMS key. 이 속성은 에만 유효합니다 AWS KMS keys. 이 속성은 key 식별자에 리전이 지정되지 않을 때만 사용되며, 그렇지 않은 경우에는 무시됩니다. 사용 시 AWS CLI 명명된 프로파일의 기본 리전을 재정의합니다.

profile

AWS CLI 이름이 지정된 프로파일을 식별합니다. 이 속성은 에만 유효합니다 AWS KMS keys. 이 프로필의 리전은 키 식별자에 리전이 지정되지 않고 이 명령에 region 속성이 없을 때만 사용됩니다.

--input (-i)

암호화 또는 복호화할 데이터의 위치를 지정합니다. 이 파라미터는 필수 사항입니다. 값은 파일 또는 디렉터리 경로 또는 파일 이름 패턴일 수 있습니다. 명령(stdin)에 대한 입력을 전달하는 경우 `-`를 사용합니다.

입력이 없는 경우 명령이 오류나 경고 없이 성공적으로 완료됩니다.

--recursive (-r, -R)

입력 디렉터리 및 해당 하위 디렉터리의 파일에 대해 작업을 수행합니다. 이 파라미터는 값이 `--input`인 디렉터리의 경우 필수입니다.

--decode

Base64로 인코딩된 입력을 디코딩합니다.

암호화된 후 인코딩된 메시지를 복호화하려면 메시지를 복호화하기 전에 먼저 메시지를 디코딩해야 합니다. 이 파라미터가 이 작업을 수행합니다.

예를 들어, 암호화 명령에 `--encode` 파라미터를 사용한 경우 해당 복호화 명령의 `--decode` 파라미터를 사용합니다. 또한 암호화하기 전에 이 파라미터를 사용하여 Base64로 인코딩된 입력을 디코딩할 수 있습니다.

--output (-o)

출력 대상을 지정합니다. 이 파라미터는 필수 사항입니다. 값은 파일 이름, 기존 디렉터리, 또는 출력을 명령줄(stdout)에 쓰는 `-`일 수 있습니다.

지정된 출력 디렉터리가 존재하지 않으면 명령은 실패합니다. 입력에 하위 디렉터리가 포함된 경우 AWS Encryption CLI는 지정한 출력 디렉터리 아래에 하위 디렉터리를 재현합니다.

기본적으로 AWS Encryption CLI는 동일한 이름의 파일을 덮어씁니다. 이 동작을 변경하려면 `--interactive` 또는 `--no-overwrite` 파라미터를 사용합니다. 덮어쓰기 경고를 표시하지 않으려면 `--quiet` 파라미터를 사용합니다.

> **Note**
>
> 출력 파일을 덮어쓰는 명령이 실패하면 출력 파일이 삭제됩니다.

--interactive

파일을 덮어쓰기 전에 메시지를 표시합니다.

--no-overwrite

파일을 덮어쓰지 않습니다. 대신 출력 파일이 있는 경우 AWS 암호화 CLI는 해당 입력을 건너뜁니다.

--suffix

AWS Encryption CLI가 생성하는 파일의 사용자 지정 파일 이름 접미사를 지정합니다. 접미사가 없음을 나타내려면 값이 없는 파라미터(`--suffix`)를 사용합니다.

기본적으로 `--output` 파라미터가 파일 이름을 지정하지 않는 경우 출력 파일 이름은 입력 파일 이름에 접미사를 더한 것과 같은 이름을 가집니다. 암호화 명령의 접미사는 `.encrypted`입니다. 복호화 명령의 접미사는 `.decrypted`입니다.

--encode

Base64(바이너리를 텍스트로) 인코딩을 출력에 적용합니다. 인코딩은 쉘 호스트 프로그램이 출력 텍스트의 비ASCII 문자를 잘못 해석하는 것을 방지합니다.

특히 PowerShell 콘솔에서 출력을 다른 명령으로 전달하거나 변수에 저장하는 경우에도 stdout(`--output -`)에 암호화된 출력을 쓸 때 이 파라미터를 사용합니다.

--metadata-output

암호화 작업에 대한 메타데이터의 위치를 지정합니다. 경로와 파일 이름을 입력합니다. 디렉터리가 존재하지 않으면 명령은 실패합니다. 명령줄(stdout)에 메타데이터를 쓰려면 `-`를 사용합니다.

명령 출력(`--output`) 및 메타데이터 출력(`--metadata-output`)을 동일한 명령으로 stdout에 쓸 수 없습니다. 또한 `--input` 또는 `--output` 값이 디렉터리(파일 이름 제외)인 경우 메타데이터 출력을 동일한 디렉터리나 해당 디렉터리의 하위 디렉터리에 쓸 수 없습니다.

기존 파일을 지정하는 경우 기본적으로 AWS Encryption CLI는 파일의 모든 콘텐츠에 새 메타데이터 레코드를 추가합니다. 이 기능을 사용하면 모든 암호화 작업에 대한 메타데이터가 포함된 단일 파일을 생성할 수 있습니다. 기존 파일의 내용을 덮어쓰려면 `--overwrite-metadata` 파라미터를 사용합니다.

AWS Encryption CLI는 명령이 수행하는 각 암호화 또는 복호화 작업에 대해 JSON 형식의 메타데이터 레코드를 반환합니다. 각 메타데이터 레코드에는 입력 및 출력 파일의 전체 경로, 암호화 컨텍스트, 알고리즘 제품군, 그리고 작업을 검토하고 보안 표준을 충족하는지 확인하는 데 사용할 수 있는 기타 중요한 정보가 포함됩니다.

--overwrite-metadata

메타데이터 출력 파일의 내용을 덮어씁니다. 기본적으로 `--metadata-output` 파라미터는 파일의 기존 내용에 메타데이터를 추가합니다.

--suppress-metadata (-S)

암호화 또는 복호화 작업에 대한 메타데이터를 숨깁니다.

--commitment-policy

암호화 및 복호화 명령에 대한 커밋 정책을 지정합니다. 커밋 정책은 메시지가 키 커밋 보안 기능을 사용하여 암호화되고 복호화되는지 여부를 결정합니다.

`--commitment-policy` 파라미터는 버전 1.8.x에 도입되었습니다. 이는 암호화 및 복호화 명령에 유효합니다.

버전 1.8.x에서 AWS Encryption CLI는 모든 암호화 및 복호화 작업에 `forbid-encrypt-allow-decrypt` 커밋 정책을 사용합니다. 암호화 또는 복호화 명령에서 `--wrapping-keys` 파라미터를 사용하는 경우 `forbid-encrypt-allow-decrypt` 값이 있는 `--commitment-policy` 파라미터가 반드시 필요합니다. `--wrapping-keys` 파라미터를 사용하지 않으면 `--commitment-policy` 파라미터는 유효하지 않습니다. 커밋 정책을 설정하면 커밋 정책이 버전 2.1.x로 업그레이드할 때 `require-encrypt-require-decrypt`로 자동 변경되는 것을 명시적으로 방지할 수 있습니다.

버전 2.1.x부터 모든 커밋 정책 값이 지원됩니다. `--commitment-policy` 파라미터는 선택 사항이며 기본값은 `require-encrypt-require-decrypt`입니다.

이 파라미터의 값은 다음과 같습니다.

- `forbid-encrypt-allow-decrypt` - 키 커밋으로 암호화할 수 없습니다. 암호화된 사이퍼텍스트를 키 커밋 사용 여부와 관계없이 복호화할 수 있습니다.

버전 1.8.x에서 이는 유일하게 유효한 값입니다. AWS Encryption CLI는 모든 암호화 및 복호화 작업에 `forbid-encrypt-allow-decrypt` 커밋 정책을 사용합니다.

- `require-encrypt-allow-decrypt` - 키 커밋을 통해서만 암호화합니다. 키 커밋 사용 여부와 관계없이 복호화합니다. 이 값은 버전 2.1.x에 도입되었습니다.
- `require-encrypt-require-decrypt`(기본값) - 키 커밋을 통해서만 암호화 및 복호화합니다. 이 값은 버전 2.1.x에 도입되었습니다. 이는 버전 2.1.x 이상에서 기본값입니다. 이 값을 사용하면 AWS Encryption CLI는 이전 버전의 로 암호화된 사이퍼텍스트를 복호화하지 않습니다 AWS Encryption SDK.

커밋 정책 설정에 대한 자세한 내용은 마이그레이션 AWS Encryption SDK 섹션을 참조하세요.

--encryption-context (-c)

작업의 암호화 컨텍스트를 지정합니다. 이 파라미터는 필수는 아니지만 권장됩니다.

- `--encrypt` 명령에 하나 이상의 `name=value` 페어를 입력합니다. 공백을 사용하여 페어를 구분합니다.
- `--decrypt` 명령에서 `name=value` 페어, 값이 없는 `name` 요소 또는 둘 다를 입력합니다.

`name=value` 페어의 `name` 또는 `value`에 공백이나 특수 문자가 포함된 경우 전체 페어를 인용 부호로 묶습니다. 예를 들어 `--encryption-context "department=software development"`입니다.

--buffer (-b)[버전 1.9.x 및 2.2.x에 도입됨]

디지털 서명이 있는지 확인하는 등 모든 입력이 처리된 후에만 일반 텍스트를 반환합니다.

--max-encrypted-data-keys[버전 1.9.x 및 2.2.x에 도입됨]

암호화된 메시지의 암호화된 데이터 키의 최대 수를 지정합니다. 이 파라미터는 선택 사항입니다.

유효한 값은 1~65,535입니다. 이 파라미터를 생략하면 AWS Encryption CLI는 최대값을 적용하지 않습니다. 암호화된 메시지는 최대 65,535(2^16 - 1)개의 암호화된 데이터 키를 보유할 수 있습니다.

암호화 명령에 이 파라미터를 사용하여 잘못된 형식의 메시지를 방지할 수 있습니다. 복호화 명령에 이를 사용하여 악성 메시지를 탐지하고 복호화할 수 없는 암호화된 데이터 키가 많이 포함된 메시지의 복호화를 방지할 수 있습니다. 자세한 정보 및 예제는 암호화된 데이터 키 제한 섹션을 참조하세요.

--help (-h)

명령줄에서 사용법과 구문을 인쇄합니다.

--version

AWS Encryption CLI의 버전을 가져옵니다.

-v | -vv | -vvv | -vvvv

자세한 정보, 경고 및 디버깅 메시지를 표시합니다. 출력의 세부 정보는 파라미터의 v 개수에 따라 증가합니다. 가장 세부적인 설정(-vvvv)은 AWS Encryption CLI 및 사용하는 모든 구성 요소에서 디버깅 수준 데이터를 반환합니다.

--quiet (-q)

출력 파일을 덮어쓸 때 나타나는 메시지와 같은 경고 메시지를 표시하지 않습니다.

--master-keys (-m)[더 이상 사용되지 않음]

> **Note**
>
> --master-keys 파라미터는 버전 1.8.x에서 더 이상 사용되지 않으며 버전 2.1.x에서 제거되었습니다. 대신 --wrapping-keys 파라미터를 사용합니다.

암호화 및 복호화 작업에 사용되는 마스터 키를 지정합니다. 각 명령에서 여러 마스터 키 파라미터를 사용할 수 있습니다.

`--master-keys` 파라미터는 암호화 명령에 반드시 필요합니다. 이 파라미터는 사용자 지정(비 AWS KMS) 마스터 키 공급자를 사용 중일 때만 복호화 명령에 반드시 필요합니다.

속성: `--master-keys` 파라미터의 값은 다음 속성으로 구성됩니다. 형식은 `attribute_name=value`입니다.

key

작업에 사용된 래핑 키를 식별합니다. 형식은 key=ID 페어입니다. key 속성은 모든 암호화 명령에 반드시 필요합니다.

암호화 명령 AWS KMS key 에서를 사용하는 경우 키 속성의 값은 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN일 수 있습니다. AWS KMS 키 식별자에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 키 식별자를 참조하세요.

key 속성은 마스터 키 공급자가 AWS KMS가 아닐 때 복호화 명령에서 필수 사항입니다. key 속성은 AWS KMS key에서 암호화된 데이터를 복호화하는 명령에는 허용되지 않습니다.

각 `--master-keys` 파라미터 값에 여러 key 속성을 지정할 수 있습니다. 그러나 모든 provider, region 및 profile 속성은 해당 파라미터 값의 모든 마스터 키에 적용됩니다. 서로 다른 속성 값으로 마스터 키를 지정하려면 명령에서 여러 `--master-keys` 파라미터를 사용합니다.

provider

마스터 키 공급자를 식별합니다. 형식은 provider=ID 페어입니다. 기본값인 aws-kms는를 나타냅니다 AWS KMS. 이 속성은 마스터 키 공급자가가 아닌 경우에만 필요합니다 AWS KMS.

리전

AWS 리전 의를 식별합니다 AWS KMS key. 이 속성은 에만 유효합니다 AWS KMS keys. 이 속성은 key 식별자에 리전이 지정되지 않을 때만 사용되며, 그렇지 않은 경우에는 무시됩니다. 사용 시 AWS CLI 명명된 프로파일의 기본 리전을 재정의합니다.

profile

AWS CLI 이름이 지정된 프로파일을 식별합니다. 이 속성은 에만 유효합니다 AWS KMS keys. 이 프로필의 리전은 키 식별자에 리전이 지정되지 않고 이 명령에 region 속성이 없을 때만 사용됩니다.

## 고급 파라미터

--algorithm

대체 알고리즘 제품군을 지정합니다. 이 파라미터는 선택 사항이며 암호화 명령에서만 유효합니다.

이 파라미터를 생략하면 AWS Encryption CLI는 버전 1.8.x에 AWS Encryption SDK 도입된에 대한 기본 알고리즘 제품군 중 하나를 사용합니다. 두 기본 알고리즘 모두 HKDF, ECDSA 서명 및 256비트 암호화 키와 함께 AES-GCM을 사용합니다. 하나는 키 커밋을 사용하고 다른 하나는 사용하지 않습니다. 기본 알고리즘 제품군 선택은 명령에 대한 커밋 정책에 따라 결정됩니다.

대부분의 암호화 작업에는 기본 알고리즘 제품군이 권장됩니다. 유효한 값 목록은 문서 읽기에서 `algorithm` 파라미터 값을 참조하세요.

--frame-length

지정된 프레임 길이로 출력을 생성합니다. 이 파라미터는 선택 사항이며 암호화 명령에서만 유효합니다.

값을 바이트 단위로 입력합니다. 유효한 값은 0 및 1~2^31 - 1입니다. 값이 0이면 프레임 처리되지 않은 데이터를 나타냅니다. 기본값은 4096(바이트)입니다.

> **Note**
>
> 가능하면 프레임 처리된 데이터를 사용하세요. 는 레거시 용도로만 프레임 처리되지 않은 데이터를 AWS Encryption SDK 지원합니다. 의 일부 언어 구현은 여전히 프레임 처리되지 않은 사이퍼텍스트를 생성할 AWS Encryption SDK 수 있습니다. 지원되는 모든 언어 구현은 프레임 처리된 사이퍼텍스트와 프레임 처리되지 않은 사이퍼텍스트를 복호화할 수 있습니다.

max-length

암호화된 메시지에서 읽을 바이트 단위의 최대 프레임 크기(또는 프레임 처리되지 않은 메시지의 경우 최대 콘텐츠 길이)를 지정합니다. 이 파라미터는 선택 사항이며 복호화 명령에서만 유효합니다. 매우 큰 악성 사이퍼텍스트를 복호화하지 못하도록 보호하기 위해 설계되었습니다.

값을 바이트 단위로 입력합니다. 이 파라미터를 생략하면는 복호화 시 프레임 크기를 제한하지 AWS Encryption SDK 않습니다.

--caching

각 입력 파일에 대해 새 데이터 키를 생성하는 대신 데이터 키를 재사용하는 데이터 키 캐싱 기능을 활성화합니다. 이 파라미터는 고급 시나리오를 지원합니다. 이 기능을 사용하기 전에 데이터 키 캐싱 설명서를 읽어보세요.

`--caching` 파라미터에는 다음 속성이 있습니다.

용량(필수)

캐시의 최대 항목 수를 결정합니다.

최소값은 1입니다. 최대값은 없습니다.

max_age(필수)

캐시 항목이 캐시에 추가된 시점부터 시작하여 캐시 항목의 사용 시간(초)을 결정합니다.

0보다 큰 값을 입력합니다. 최대값은 없습니다.

max_messages_encrypted(선택 사항)

캐시된 항목이 암호화할 수 있는 최대 메시지 수를 결정합니다.

유효한 값은 1~2^32입니다. 기본값은 메시지 2^32개입니다.

max_bytes_encrypted(선택 사항)

캐시된 항목이 암호화할 수 있는 최대 바이트 수를 결정합니다.

유효한 값은 0 및 1~2^63 - 1입니다. 기본값은 메시지 2^63 - 1개입니다. 값이 0이면 빈 메시지 문자열을 암호화하는 경우에만 데이터 키 캐싱을 사용할 수 있습니다.

## AWS 암호화 CLI 버전

최신 버전의 AWS Encryption CLI를 사용하는 것이 좋습니다.

> **Note**
>
> 4.0.0 이전의 AWS Encryption CLI 버전은 end-of-support 단계에 있습니다.
> 코드나 데이터를 변경하지 않고 버전 2.1.x 이상에서 AWS Encryption CLI의 최신 버전으로 안전하게 업데이트할 수 있습니다. 그러나 버전 2.1.x에 도입된 새로운 보안 기능은 이하 버전과 호환되지 않습니다. 버전 1.7.x 이하에서 업데이트하려면 먼저 AWS Encryption CLI의 최신 1.x 버전으로 업데이트해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.
> 새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

의 중요 버전에 대한 자세한 내용은 섹션을 AWS Encryption SDK참조하세요의 버전 AWS Encryption SDK.

어떤 버전을 사용해야 하나요?

AWS Encryption CLI를 처음 사용하는 경우 최신 버전을 사용합니다.

AWS Encryption SDK 이전 버전 1.7.x로 암호화된 데이터를 해독하려면 먼저 최신 버전의 AWS Encryption CLI로 마이그레이션합니다. 버전 2.1.x 이상으로 업데이트하기 전에 모든 권장 사항을 변경합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.

자세히 알아보기

- 변경 사항에 대한 자세한 내용과, 새 버전으로 마이그레이션하기 위한 지침은 마이그레이션 AWS Encryption SDK 섹션을 참조하세요.

- 새로운 AWS Encryption CLI 파라미터 및 속성에 대한 설명은 섹션을 참조하세요AWS Encryption SDK CLI 구문 및 파라미터 참조.

다음 목록은 버전 1.8.x 및 2.1.x에서 AWS Encryption CLI의 변경 사항을 설명합니다.

## AWS 암호화 CLI 버전 1.8.x 변경 사항

- `--master-keys` 파라미터를 더 이상 사용하지 않습니다. 대신 `--wrapping-keys` 파라미터를 사용합니다.
- `--wrapping-keys(-w)` 파라미터를 추가합니다. `--master-keys` 파라미터의 모든 속성을 지원합니다. 또한 AWS KMS keys를 사용하여 복호화할 때만 유효한 다음과 같은 선택적 속성을 추가합니다.
  - discovery
  - discovery-partition
  - discovery-account

  사용자 지정 마스터 키 공급자의 경우 `--encrypt` 및 `--decrypt` 명령에는 `--wrapping-keys` 또는 `--master-keys` 파라미터가 필요합니다(둘 다는 아님). 또한를 사용하는 `--encrypt` 명령에는 `--wrapping-keys` 파라미터 또는 `--master-keys` 파라미터(둘 다 아님)가 AWS KMS keys 필요합니다.

  를 사용하는 `--decrypt` 명령에서 AWS KMS keys`--wrapping-keys` 파라미터는 선택 사항이지만 버전 2.1.x에서 필요하므로 권장됩니다. 이를 사용하는 경우 key 속성 또는 discovery 속성 중 하나를 `true` 값으로 지정해야 합니다(둘 다는 아님).
- `--commitment-policy` 파라미터를 추가합니다. 유일한 유효 값은 `forbid-encrypt-allow-decrypt`입니다. `forbid-encrypt-allow-decrypt` 커밋 정책은 모든 암호화 및 복호화 명령에 사용됩니다.

  버전 1.8.x에서 `--wrapping-keys` 파라미터를 사용할 때는 `forbid-encrypt-allow-decrypt` 값이 있는 `--commitment-policy` 파라미터가 필요합니다. 이 값을 명시적으로 설정하면 버전 2.1.x로 업그레이드할 때 커밋 정책이 자동으로 `require-encrypt-require-decrypt`로 변경되는 것을 방지할 수 있습니다.

## AWS 암호화 CLI에 대한 버전 2.1.x 변경 사항

- `--master-keys` 파라미터를 제거합니다. 대신 `--wrapping-keys` 파라미터를 사용합니다.

- `--wrapping-keys` 파라미터는 모든 암호화 및 복호화 명령에 반드시 필요합니다. key 속성 또는 discovery 속성 중 하나를 `true` 값으로 지정해야 합니다(둘 다는 아님).
- `--commitment-policy` 파라미터는 다음과 같은 값을 지원합니다. 자세한 내용은 커밋 정책 설정을 참조하세요.
  - `forbid-encrypt-allow-decrypt`
  - `require-encrypt-allow-decrypt`
  - `require-encrypt-require decrypt`(기본값)
- `--commitment-policy` 파라미터는 버전 2.1x에서 선택 사항입니다. 기본값은 `require-encrypt-require-decrypt`입니다.

## AWS Encryption CLI 버전 1.9.x 및 2.2.x 변경 사항

- `--decrypt-unsigned` 파라미터를 추가합니다. 자세한 내용은 버전 2.2.x을 참조하세요.
- `--buffer` 파라미터를 추가합니다. 자세한 내용은 버전 2.2.x을 참조하세요.
- `--max-encrypted-data-keys` 파라미터를 추가합니다. 자세한 내용은 암호화된 데이터 키 제한을 참조하세요.

## AWS 암호화 CLI 버전 3.0.x 변경 사항

- AWS KMS 다중 리전 키에 대한 지원을 추가합니다. 자세한 내용은 다중 리전 사용 AWS KMS keys 섹션을 참조하십시오.

# 데이터 키 캐싱

데이터 키 캐싱은 데이터 키 및 관련 암호화 자료를 캐시에 저장합니다. 데이터를 암호화하거나 해독할 때는 캐시에서 일치하는 데이터 키를 AWS Encryption SDK 찾습니다. 일치하는 데이터 키를 찾으면 새 키를 생성하는 대신 캐시된 데이터 키를 사용합니다. 데이터 키 캐싱은 성능을 개선하고 비용을 절감하며 애플리케이션 확장에 따른 서비스 한도 내에서 유지하는 데 도움이 됩니다.

다음과 같은 경우 애플리케이션에서 데이터 키 캐싱의 이점을 누릴 수 있습니다.

- 데이터 키를 재사용할 수 있습니다.
- 수많은 데이터 키를 생성합니다.
- 암호화 작업은 용납할 수 없을 정도로 느리거나, 비용이 많이 들거나, 제한적이거나, 리소스 집약적입니다.

캐싱은 AWS Key Management Service ()와 같은 암호화 서비스의 사용을 줄일 수 있습니다AWS KMS. AWS KMS 초당 요청 수 한도에 도달한 경우 캐싱이 유용할 수 있습니다. 애플리케이션은 캐시된 키를 사용하여를 호출하는 대신 일부 데이터 키 요청을 처리할 수 있습니다 AWS KMS. (AWS Support Center에서 사례를 생성하여 계정 한도를 높일 수도 있습니다.)

는 데이터 키 캐시를 생성하고 관리하는 데 AWS Encryption SDK 도움이 됩니다. 로컬 캐시와, 캐시와 상호 작용하고 사용자가 설정한 보안 임계값을 적용하는 캐싱 암호화 자료 관리자(캐싱 CMM)를 제공합니다. 이러한 구성 요소를 함께 사용하면 시스템 보안을 유지하면서 데이터 키를 효율적으로 재사용할 수 있습니다.

데이터 키 캐싱은 주의해야 AWS Encryption SDK 하는의 선택적 기능입니다. 기본적으로는 모든 암호화 작업에 대해 새 데이터 키를 AWS Encryption SDK 생성합니다. 이 기법은 암호화 모범 사례를 지원하므로 데이터 키를 과도하게 재사용하지 않도록 합니다. 일반적으로 데이터 키 캐싱은 성능 목표를 달성하는 데 필요한 경우에만 사용합니다. 그런 다음 데이터 키 캐싱 보안 임계값을 사용하여 비용 및 성능 목표를 달성하는 데 필요한 최소한의 캐싱을 사용하는지 확인합니다.

버전 3.x는 키링 인터페이스가 아닌 레거시 마스터 키 공급자 인터페이스를 사용하는 캐싱 CMM AWS Encryption SDK for Java 만 지원합니다. 그러나 .NET AWS Encryption SDK 용의 버전 4.x,의 버전 3.x AWS Encryption SDK for Java, Rust AWS Encryption SDK 용 의 버전 4.x AWS Encryption SDK for Python, Go AWS Encryption SDK 용의 버전 1.x는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 지원합니다. AWS KMS 계층적 키링으로 암호화된 콘텐츠는 AWS KMS 계층적 키링으로만 복호화할 수 있습니다.

이러한 보안 트레이드오프에 대한 자세한 내용은 AWS 보안 블로그의 AWS Encryption SDK: 데이터 키 캐싱이 애플리케이션에 적합한지 결정하는 방법을 참조하세요.

주제

- 데이터 키 캐싱 사용 방법
- 캐시 보안 임계값 설정
- 데이터 키 캐싱 세부 정보
- 데이터 키 캐싱 예제

# 데이터 키 캐싱 사용 방법

이 주제에서는 애플리케이션에서 데이터 키 캐싱을 사용하는 방법을 보여줍니다. 프로세스를 단계별로 안내합니다. 그런 다음, 작업에서 데이터 키 캐싱을 사용하여 문자열을 암호화하는 간단한 예제로 단계들을 결합합니다.

이 섹션의 예제에서는 2.0.x 이상 버전의 AWS Encryption SDK를 사용하는 방법을 보여줍니다. 이하 버전을 사용하는 예제의 경우 프로그래밍 언어에 대한 GitHub의 리포지토리의 릴리스 목록에서 해당하는 릴리스를 찾을 수 있습니다.

에서 데이터 키 캐싱을 사용하는 전체 예제와 테스트된 예제는 다음을 AWS Encryption SDK참조하세요.

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java
- JavaScript 브라우저: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

AWS Encryption SDK for .NET은 데이터 키 캐싱을 지원하지 않습니다.

주제

- 데이터 키 캐싱 사용: 단계별
- 데이터 키 캐싱 예제: 문자열 암호화

# 데이터 키 캐싱 사용: 단계별

이 단계별 지침은 데이터 키 캐싱을 구현하는 데 필요한 구성 요소를 생성하는 방법을 보여줍니다.

- 데이터 키 캐시를 생성합니다. 이 예제에서는가 AWS Encryption SDK 제공하는 로컬 캐시를 사용합니다. 캐시를 10개의 데이터 키로 제한합니다.

  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

  다음 예제에서는의 버전 2.x를 사용합니다 AWS Encryption SDK for Java. 버전 3.x는 데이터 키 캐싱 CMM을 더 AWS Encryption SDK for Java 이상 사용하지 않습니다. 버전 3.x에서는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 사용할 수도 있습니다.

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

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

  JavaScript Browser

  ```
  const capacity = 10

  const cache = getLocalCryptographicMaterialsCache(capacity)
  ```

  JavaScript Node.js

  ```
  const capacity = 10

  const cache = getLocalCryptographicMaterialsCache(capacity)
  ```

Python

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

cache = aws_encryption_sdk.LocalCryptoMaterialsCache(MAX_CACHE_SIZE)
```

- 마스터 키 공급자(Java 및 Python) 또는 키링(C 및 JavaScript)을 생성합니다. 이 예제에서는 AWS Key Management Service (AWS KMS) 마스터 키 공급자 또는 호환되는 AWS KMS 키링을 사용합니다.

  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

  다음 예제에서는의 버전 2.x를 사용합니다 AWS Encryption SDK for Java. 버전 3.x는 데이터 키 캐싱 CMM을 더 AWS Encryption SDK for Java 이상 사용하지 않습니다. 버전 3.x에서는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 사용할 수도 있습니다.

  ```
  // 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

  브라우저에 보안 인증을 안전하게 입력해야 합니다. 이 예는 런타임 시 보안 인증을 확인하는 webpack(kms.webpack.config)에서 보안 인증을 정의합니다. AWS KMS 클라이언트 및 자격 증명에서 AWS KMS 클라이언트 공급자 인스턴스를 생성합니다. 그런 다음 키링을 생성할 때 클라이언트 공급자를 AWS KMS key (`generatorKeyId`).

```
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)
 */ of an AWS KMS key
const keyring = new KmsKeyringBrowser({
    clientProvider,
    generatorKeyId,
    keyIds,
  })
```

JavaScript Node.js

```
/* Create an AWS KMS keyring
 *   The input is the Amazon Resource Name (ARN)
*/   of an AWS KMS key
const keyring = new KmsKeyringNode({ generatorKeyId })
```

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])
```

- 캐싱 암호화 자료 관리자(캐싱 CMM)를 생성합니다.

캐싱 CMM을 캐시 및 마스터 키 공급자 또는 키링과 연결합니다. 그런 다음 캐싱 CMM에서 캐시 보안 임계값을 설정합니다.

C

에서는 기본 CMM과 같은 기본 CMM 또는 키링에서 캐싱 CMM을 생성할 AWS Encryption SDK for C수 있습니다. 이 예는 키링에서 캐싱 CMM을 생성합니다.

캐싱 CMM을 생성한 후 키링 및 캐시에 대한 참조를 릴리스할 수 있습니다. 자세한 내용은 the section called “참조 카운트”을 참조하세요.

```
// 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

다음 예제에서는의 버전 2.x를 사용합니다 AWS Encryption SDK for Java. 버전 3.x AWS Encryption SDK for Java 는 데이터 키 캐싱을 지원하지 않지만 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 지원합니다.

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

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

### 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
)
```

더 이상 수행할 작업이 없습니다. 그런 다음에서 캐시를 AWS Encryption SDK 관리하도록 하거나 자체 캐시 관리 로직을 추가합니다.

호출에서 데이터 키 캐싱을 사용하여 데이터를 암호화하거나 복호화하려는 경우 마스터 키 공급자 또는 다른 CMM 대신에 사용자의 캐싱 CMM을 지정합니다.

> **Note**
>
> 데이터 스트림이나, 크기를 알 수 없는 데이터를 암호화하는 경우 요청에서 데이터 크기를 지정해야 합니다. AWS Encryption SDK 는 알 수 없는 크기의 데이터를 암호화할 때 데이터 키 캐싱을 사용하지 않습니다.

C

에서 캐싱 CMM으로 세션을 AWS Encryption SDK for C생성한 다음 세션을 처리합니다.

기본적으로 메시지 크기를 알 수 없고 제한이 없는 경우는 데이터 키를 캐싱하지 AWS Encryption SDK 않습니다. 정확한 데이터 크기를 모를 때 캐싱을 허용하려면 `aws_cryptosdk_session_set_message_bound` 메서드를 사용하여 메시지의 최대 크기를 설

정합니다. 범위를 예상 메시지 크기보다 크게 설정합니다. 실제 메시지 크기가 범위를 초과하면 암호화 작업이 실패합니다.

```
/* 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

다음 예제에서는 버전 2.x를 사용합니다 AWS Encryption SDK for Java. 버전 3.x는 데이터 키 캐싱 CMM을 더 AWS Encryption SDK for Java 이상 사용하지 않습니다. 버전 3.x에서는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 사용할 수도 있습니다.

```
// 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();
```

JavaScript Browser

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

JavaScript Node.js

AWS Encryption SDK for JavaScript for Node.js에서 캐싱 CMM을 사용하는 경우 `encrypt` 메서드에는 일반 텍스트의 길이가 필요합니다. 제공하지 않으면 데이터 키가 캐시되지 않습니다. 길이는 제공해도 입력한 일반 텍스트 데이터가 해당 길이를 초과하면 암호화 작업이 실패합니다. 데이터를

스트리밍할 때와 같이 일반 텍스트의 정확한 길이를 모르는 경우 예상되는 가장 큰 값을 제공합니다.

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

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
)
```

## 데이터 키 캐싱 예제: 문자열 암호화

이 간단한 코드 예제는 문자열을 암호화할 때 데이터 키 캐싱을 사용합니다. 단계별 프로시저의 코드를 실행 가능한 테스트 코드로 결합합니다.

이 예제에서는 로컬 캐시 및 AWS KMS key에 대한 마스터 키 공급자 또는 키링을 생성합니다. 그런 다음 로컬 캐시 및 마스터 키 공급자 또는 키링을 사용하여 적절한 보안 임계값이 있는 캐싱 CMM을 생성합니다. Java 및 Python에서 암호화 요청은 캐싱 CMM, 암호화할 일반 텍스트 데이터 및 암호화 컨텍스트를 지정합니다. C에서는 세션에 캐싱 CMM이 지정되며, 세션은 암호화 요청에 제공됩니다.

이 예제를 실행하려면 AWS KMS key의 Amazon 리소스 이름(ARN)을 제공해야 합니다. 데이터 키를 생성하려면 AWS KMS key를 사용할 수 있는 권한이 있어야 합니다.

데이터 키 캐시 생성 및 사용에 대한 자세한 실제 예제는 데이터 키 캐싱 예제 코드 섹션을 참조하세요.

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

다음 예제에서는의 버전 2.x를 사용합니다 AWS Encryption SDK for Java. 버전 3.x는 데이터 키 캐싱 CMM을 더 AWS Encryption SDK for Java 이상 사용하지 않습니다. 버전 3.x에서는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 사용할 수도 있습니다.

```
// 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
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>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/
developerguide/find-cmk-id-arn.html
 * <li>Max entry age: Maximum time (in seconds) that a cached entry can be used
 * <li>Cache capacity: Maximum number of entries in the cache
 * </ol>
 */
public class SimpleDataKeyCachingExample {

    /*
```

```
     * Security thresholds
     *   Max entry age is required.
     *   Max messages (and max bytes) per data key are optional
     */
    private static final int MAX_ENTRY_MSGS = 100;

    public static byte[] encryptWithCaching(String 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:
        // 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("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()`.
 */
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 =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* 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 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. */
  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 }
}
```

### JavaScript Node.js

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

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 =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* 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.
   */
  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

```
# 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"

    # 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_R

    # 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
```

## 캐시 보안 임계값 설정

데이터 키 캐싱을 구현할 때는 캐싱 CMM이 적용하는 보안 임계값을 구성해야 합니다.

보안 임계값을 통해, 캐시된 각 데이터 키가 사용되는 기간과 각 데이터 키에서 보호되는 데이터의 양을 제한할 수 있습니다. 캐싱 CMM은 캐시 항목이 모든 보안 임계값을 준수하는 경우에만 캐시된 데이터 키를 반환합니다. 캐시 항목이 임계값을 초과하는 경우 해당 항목은 현재 작업에 사용되지 않으며 가능한 한 빨리 캐시에서 제거됩니다. 각 데이터 키를 처음 사용한 경우(캐싱 전)에는 이 임계값이 적용되지 않습니다.

일반적으로 비용 및 성능 목표를 달성하는 데 필요한 최소한의 캐싱을 사용합니다.

AWS Encryption SDK 만 키 파생 함수를 사용하여 암호화된 데이터 키를 캐싱합니다. 또한 일부 임계값에 대한 상한선을 설정합니다. 이러한 제한은 데이터 키가 암호화 한도를 초과하여 재사용되지 않도록 합니다. 그러나 일반 텍스트 데이터 키는 (기본적으로 메모리 내에) 캐시되므로, 키가 저장되는 시간을 최소화하도록 합니다. 또한 키가 손상될 경우 노출될 수 있는 데이터를 제한하도록 합니다.

캐시 보안 임계값 설정 예제는 AWS 보안 블로그의 AWS Encryption SDK: 데이터 키 캐싱이 애플리케이션에 적합한지 결정하는 방법을 참조하세요.

> **Note**
>
> 캐싱 CMM은 다음 임계값을 모두 적용합니다. 옵션 값을 지정하지 않으면 캐싱 CMM에 기본값이 사용됩니다.
>
> 데이터 키 캐싱을 일시적으로 비활성화하기 위해 AWS Encryption SDK 의 Java 및 Python 구현에서는 null 암호화 자료 캐시(null 캐시)를 제공합니다. null 캐시는 모든 GET 요청에 대해 누락을 반환하고 PUT 요청에 응답하지 않습니다. 캐시 용량 또는 보안 임계값을 0으로 설정하는 대신 null 캐시를 사용하는 것이 좋습니다. 자세한 내용은 Java 및 Python에서 null 캐시를 참조하세요.

최대 기간(필수)

추가된 시점부터 시작하여 캐시된 항목을 사용할 수 있는 기간을 결정합니다. 이 값은 필수입니다. 0보다 큰 값을 입력합니다. 는 최대 수명 값을 제한하지 AWS Encryption SDK 않습니다.

의 모든 언어 구현은 밀리초를 AWS Encryption SDK for JavaScript사용하는를 제외하고 최대 수명을 초 단위로 AWS Encryption SDK 정의합니다.

애플리케이션이 캐시를 활용할 수 있는 가장 짧은 간격을 사용합니다. 최대 기간 임계값을 키 교체 정책처럼 사용할 수 있습니다. 이를 사용하여 데이터 키의 재사용을 제한하고, 암호화 자료의 노출을 최소화하며, 캐시되는 동안 정책이 변경되었을 수 있는 데이터 키를 제거할 수 있습니다.

최대 메시지 암호화(선택 사항)

캐시된 데이터 키가 암호화할 수 있는 최대 메시지 수를 지정합니다. 이 값은 선택 사항입니다. 1과 2^32개 메시지 사이의 값을 입력합니다. 기본값은 메시지 2^32개입니다.

캐시된 각 키로 보호되는 메시지 수는, 재사용을 통해 값을 가져올 수 있을 만큼 크지만 키가 손상될 경우 노출될 수 있는 메시지 수를 제한할 수 있을 만큼 작게 설정합니다.

최대 바이트 암호화(선택 사항)

캐시된 데이터 키가 암호화할 수 있는 최대 바이트 수를 지정합니다. 이 값은 선택 사항입니다. 0과 2^63 - 1 사이의 값을 입력합니다. 기본값은 2^63 - 1입니다. 값이 0이면 빈 메시지 문자열을 암호화하는 경우에만 데이터 키 캐싱을 사용할 수 있습니다.

이 임계값을 평가할 때 현재 요청의 바이트가 포함됩니다. 처리된 바이트와 현재 바이트가 임계값을 초과하면 캐시된 데이터 키가 더 적은 요청에서 사용되었을 수도 있지만 캐시에서 제거됩니다.

# 데이터 키 캐싱 세부 정보

대부분의 애플리케이션은 사용자 지정 코드를 작성하지 않고도 데이터 키 캐싱의 기본 구현을 사용할 수 있습니다. 이 섹션에서는 기본 구현과, 옵션에 대한 몇 가지 세부 정보를 설명합니다.

주제



## 데이터 키 캐싱의 작동 방식

요청에서 데이터 키 캐싱을 사용하여 데이터를 암호화하거나 복호화하는 경우 AWS Encryption SDK는 먼저 캐시에서 요청과 일치하는 데이터 키를 검색합니다. 유효한 일치 항목을 찾으면 캐시된 데이터 키를 사용하여 데이터를 암호화합니다. 그러지 않으면 캐시가 없을 때와 마찬가지로 새 데이터 키가 생성됩니다.

스트리밍 데이터와 같이 크기를 알 수 없는 데이터에는 데이터 키 캐싱이 사용되지 않습니다. 이를 통해 캐싱 CMM이 최대 바이트 임계값을 적절하게 적용할 수 있습니다. 이 동작을 방지하려면 메시지 크기를 암호화 요청에 추가합니다.

데이터 키 캐싱은 캐시 외에도 캐싱 암호화 자료 관리자(캐싱 CMM)를 사용합니다. 캐싱 CMM은 캐시 및 기본 CMM과 상호 작용하는 특수 암호화 자료 관리자(CMM)입니다. (마스터 키 공급자 또는 키링을 지정하면 AWS Encryption SDK 에서 기본 CMM을 만듭니다.) 캐싱 CMM은 기본 CMM이 반환하는 데이터 키를 캐시합니다. 또한 캐싱 CMM은 사용자가 설정한 캐시 보안 임계값을 적용합니다.

캐시에서 잘못된 데이터 키가 선택되는 것을 방지하기 위해 모든 호환 가능한 캐싱 CMM은 캐시된 암호화 자료의 다음 속성이 구성 요소 요청과 일치해야 합니다.

- 알고리즘 제품군
- 암호화 컨텍스트(비어 있는 경우에도)
- 파티션 이름(캐싱 CMM을 식별하는 문자열)
- (복호화 전용) 암호화된 데이터 키

> **Note**
>
> 는 알고리즘 제품군이 키 유도 함수를 사용하는 경우에만 데이터 키를 AWS Encryption SDK 캐시합니다.

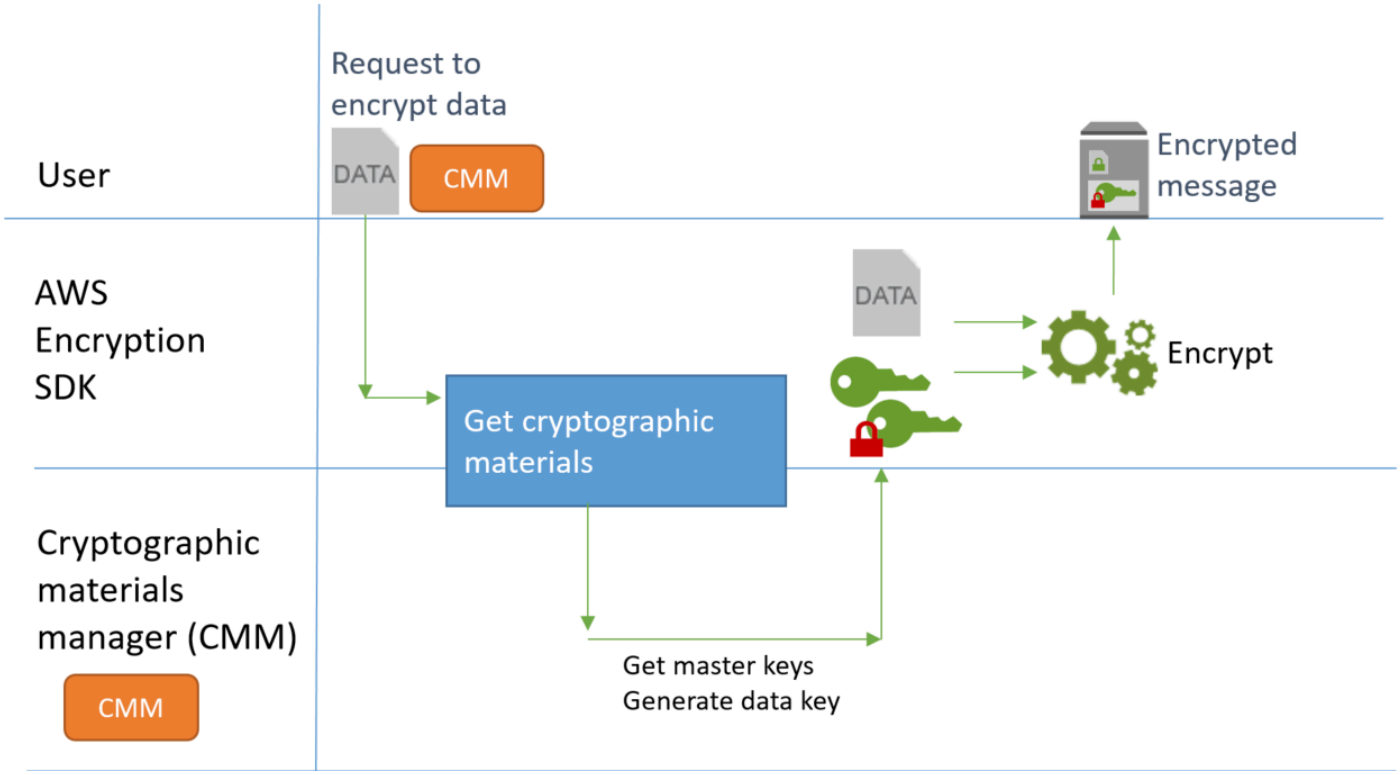
다음 워크플로는 데이터 키 캐싱을 사용하거나 사용하지 않고 데이터 암호화 요청을 처리하는 방법을 보여줍니다. 캐시와 캐싱 CMM을 포함하여 사용자가 생성한 캐싱 구성 요소가 프로세스에서 어떻게 사용되는지 보여줍니다.

## 캐싱을 사용하지 않고 데이터 암호화

캐싱을 사용하지 않고 암호화 자료를 가져오려면 다음을 수행합니다.

1. 애플리케이션이에 데이터 암호화 AWS Encryption SDK 를 요청합니다.

   요청은 마스터 키 공급자 또는 키링을 지정합니다. AWS Encryption SDK 는 사용자 마스터 키 또는 키링과 상호 작용하는 기본 CMM을 만듭니다.
2. 는 CMM에 암호화 자료를 AWS Encryption SDK 요청합니다(암호화 자료 가져오기).
3. CMM은 해당 키링(C 및 JavaScript) 또는 마스터 키 공급자(Java 및 Python)에 암호화 자료를 요청합니다. 여기에는 AWS Key Management Service ()와 같은 암호화 서비스에 대한 호출이 포함될 수 있습니다AWS KMS. CMM은 암호화 자료를 AWS Encryption SDK에 반환합니다.
4. 는 일반 텍스트 데이터 키를 AWS Encryption SDK 사용하여 데이터를 암호화합니다. 암호화된 데이터와 암호화된 데이터 키를 암호화된 메시지에 저장하여 사용자에게 반환합니다.




## 캐싱을 사용하여 데이터 암호화

데이터 키 캐싱을 사용하여 암호화 자료를 가져오려면 다음을 수행합니다.

1. 애플리케이션이에 데이터 암호화 AWS Encryption SDK 를 요청합니다.

   요청은 기본 암호 구성 요소 관리자(CMM)와 연결된 캐싱 암호 구성 요소 관리자(캐싱 CMM)를 지정합니다. 마스터 키 공급자 또는 키링을 지정하면 AWS Encryption SDK 에서 기본 CMM을 만듭니다.
2. SDK는 지정된 캐싱 CMM에 암호화 자료를 요청합니다.
3. 캐싱 CMM은 캐시에서 암호화 자료를 요청합니다.
   a. 캐시가 일치하는 항목을 찾으면 일치하는 캐시 항목의 사용 기간 및 사용 값을 업데이트하고 캐시된 암호화 자료를 캐싱 CMM에 반환합니다.

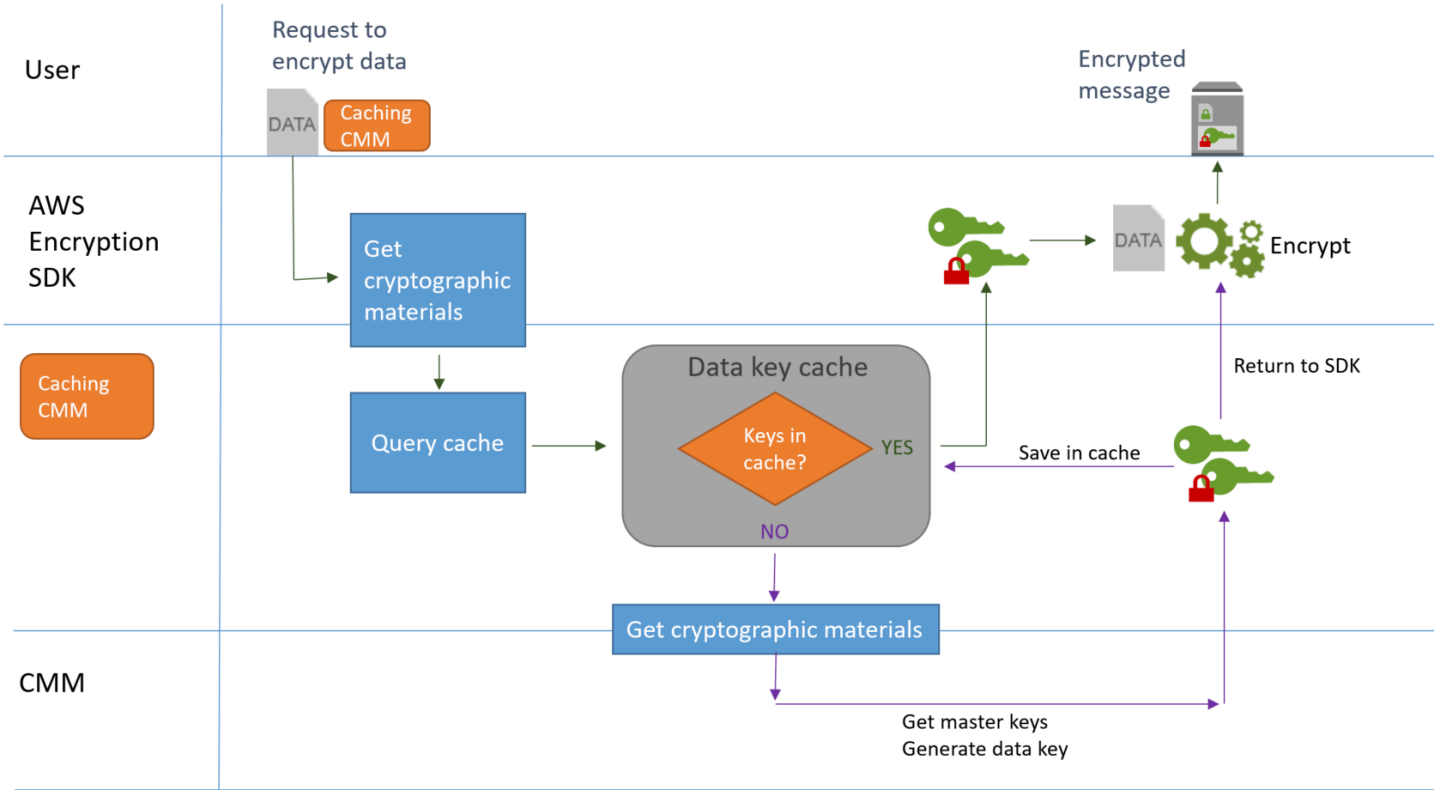
      캐시 항목이 보안 임계값을 준수하는 경우 캐싱 CMM은 해당 항목을 SDK에 반환합니다. 그러지 않으면 항목을 제거하라고 캐시에 지시하고 일치하는 항목이 없는 것처럼 진행합니다.
   b. 캐시에서 유효한 일치 항목을 찾을 수 없는 경우 캐싱 CMM은 기본 CMM에 새 데이터 키를 생성하도록 요청합니다.

      기본 CMM은 해당 키링(C 및 JavaScript) 또는 마스터 키 공급자(Java 및 Python)로부터 암호화 자료를 가져옵니다. 여기에는 AWS Key Management Service와 같은 서비스에 대한 호출이 포함

될 수 있습니다. 기본 CMM은 데이터 키의 일반 텍스트 및 암호화된 사본을 캐싱 CMM에 반환합니다.

캐싱 CMM은 새 암호화 자료를 캐시에 저장합니다.

4. 캐싱 CMM은 암호화 자료를 AWS Encryption SDK에 반환합니다.
5. 는 일반 텍스트 데이터 키를 AWS Encryption SDK 사용하여 데이터를 암호화합니다. 암호화된 데이터와 암호화된 데이터 키를 암호화된 메시지에 저장하여 사용자에게 반환합니다.




## 암호화 자료 캐시 생성

는 데이터 키 캐싱에 사용되는 암호화 자료 캐시에 대한 요구 사항을 AWS Encryption SDK 정의합니다. 또한 구성 가능한 메모리 내 최소 최근 사용(LRU) 캐시인 로컬 캐시도 제공합니다. 로컬 캐시의 인스턴스를 생성하려면 Java 및 Python에서 `LocalCryptoMaterialsCache` 생성자를 생성하거나, JavaScript에서 getLocalCryptographicMaterialsCache 함수 또는 C에서 `aws_cryptosdk_materials_cache_local_new` 생성자를 사용합니다.

로컬 캐시에는 캐시된 항목의 추가, 제거, 일치 및 캐시 유지 관리를 비롯한 기본 캐시 관리를 위한 로직이 포함되어 있습니다. 사용자 지정 캐시 관리 로직을 작성할 필요가 없습니다. 로컬 캐시를 그대로 사용하거나, 사용자 지정하거나, 호환되는 캐시로 대체할 수 있습니다.

로컬 캐시를 생성할 때 용량, 즉 캐시에 저장할 수 있는 최대 항목 수를 설정합니다. 이 설정을 사용하면 데이터 키 재사용이 제한되는 효율적인 캐시를 설계할 수 있습니다.

AWS Encryption SDK for Java 및 는 null 암호화 자료 캐시(NullCryptoMaterialsCache) AWS Encryption SDK for Python 도 제공합니다. NullCryptoMaterialsCache는 모든 GET 작업에 대해 누락을 반환하고 PUT 작업에 응답하지 않습니다. NullCryptoMaterialsCache를 테스트에 사용하거나 캐싱 코드가 포함된 애플리케이션에서 캐싱을 일시적으로 비활성화할 수 있습니다.

에서 AWS Encryption SDK각 암호화 자료 캐시는 캐싱 암호화 자료 관리자(캐싱 CMM)와 연결됩니다. 캐싱 CMM은 캐시에서 데이터 키를 가져와 캐시에 데이터 키를 넣고 사용자가 설정한 보안 임계값을 적용합니다. 캐싱 CMM을 생성할 때, 해당 CMM이 사용할 캐시를 지정하고, CMM이 캐싱할 데이터 키를 생성하는 기본 CMM 또는 마스터 키 공급자를 지정합니다.

## 암호화 자료 캐싱 관리자 생성

데이터 키 캐싱을 활성화하려면 캐시 및 캐싱 암호화 자료 관리자(캐싱 CMM)를 생성합니다. 그런 다음 데이터 암호화 또는 복호화 요청에서 표준 암호화 자료 관리자(CMM), 마스터 키 공급자 또는 키링 대신 캐싱 CMM을 지정합니다.

다음과 같은 두 가지 유형의 CMM이 있습니다. 둘 다 데이터 키(및 관련 암호화 자료)를 가져오지만 다음과 같이 방법이 다릅니다.

- CMM은 키링(C 또는 JavaScript) 또는 마스터 키 공급자(Java와 Python)와 연결됩니다. SDK가 CMM에 암호화 또는 복호화 구성 요소를 요청하면 CMM은 키링 또는 마스터 키 공급자로부터 구성 요소를 가져옵니다. Java 및 Python에서 CMM은 마스터 키를 사용하여 데이터 키를 생성, 암호화 또는 복호화합니다. C 및 JavaScript에서 키링은 암호화 자료를 생성 및 암호화하고 반환합니다.
- 캐싱 CMM은 하나의 캐시(예: 로컬 캐시 및 기본 CMM)와 연결됩니다. SDK가 캐싱 CMM에 암호화 자료를 요청하면 캐싱 CMM은 캐시에서 해당 구성 요소를 가져오려고 시도합니다. 일치하는 항목을 찾을 수 없는 경우 캐싱 CMM은 기본 CMM에 구성 요소를 요청합니다. 그런 다음 새 암호화 자료를 캐싱한 다음 호출자에게 반환합니다.

또한 캐싱 CMM은 각 캐시 항목에 사용자가 설정한 보안 임계값을 적용합니다. 보안 임계값은 캐싱 CMM에서 설정하고 적용하므로 캐시가 민감한 구성 요소용으로 설계되지 않았더라도 호환되는 모든 캐시를 사용할 수 있습니다.

## 데이터 키 캐시 항목에는 무엇이 들어 있나요?

데이터 키 캐싱은 데이터 키 및 관련 암호화 자료를 캐시에 저장합니다. 각 항목에는 아래 나열된 요소가 포함됩니다. 이 정보는 데이터 키 캐싱 기능을 사용할지 여부를 결정할 때와, 캐싱 암호화 자료 관리자(캐싱 CMM)에서 보안 임계값을 설정할 때 유용할 수 있습니다.

암호화 요청의 캐시된 항목

암호화 작업의 결과로 데이터 키 캐시에 추가되는 항목에는 다음 요소가 포함됩니다.

- 일반 텍스트 데이터 키
- 암호화된 데이터 키(하나 이상)
- 암호화 컨텍스트
- 메시지 서명 키(하나가 사용되는 경우)
- 알고리즘 제품군
- 메타데이터(보안 임계값 적용을 위한 사용 카운터 포함)

복호화 요청의 캐시된 항목

복호화 작업의 결과로 데이터 키 캐시에 추가되는 항목에는 다음 요소가 포함됩니다.

- 일반 텍스트 데이터 키
- 서명 확인 키(하나가 사용되는 경우)
- 메타데이터(보안 임계값 적용을 위한 사용 카운터 포함)

## 암호화 컨텍스트: 캐시 항목을 선택하는 방법

모든 요청에 암호화 컨텍스트를 지정하여 데이터를 암호화할 수 있습니다. 하지만 암호화 컨텍스트는 데이터 키 캐싱에서 특별한 역할을 합니다. 이를 통해 데이터 키가 동일한 캐싱 CMM에서 생성된 경우에도 캐시에 데이터 키의 하위 그룹을 만들 수 있습니다.

암호화 컨텍스트는 비밀이 아닌 임의의 데이터를 포함하는 키-값 페어 세트입니다. 암호화하는 동안 암호화 컨텍스트는 암호화된 데이터에 암호적으로 바인딩되므로 데이터를 복호화하는 데 동일한 암호화 컨텍스트가 필요합니다. 에서 AWS Encryption SDK암호화 컨텍스트는 암호화된 데이터 및 데이터 키와 함께 암호화된 메시지에 저장됩니다.

데이터 키 캐시를 사용하는 경우 암호화 컨텍스트를 통해 암호화 작업에 사용할 캐시된 특정 데이터 키를 선택할 수도 있습니다. 암호화 컨텍스트는 데이터 키(캐시 항목 ID의 일부)와 함께 캐시 항목에 저장

됩니다. 캐시된 데이터 키는 암호화 컨텍스트가 일치하는 경우에만 재사용됩니다. 암호화 요청에 특정 데이터 키를 재사용하려면 동일한 암호화 컨텍스트를 지정합니다. 이러한 데이터 키를 피하려면 다른 암호화 컨텍스트를 지정합니다.

암호화 컨텍스트는 항상 선택 사항이지만 권장됩니다. 요청에 암호화 컨텍스트를 지정하지 않는 경우 빈 암호화 컨텍스트가 캐시 항목 식별자에 포함되고 각 요청과 일치합니다.

## 내 애플리케이션이 캐시된 데이터 키를 사용하고 있나요?

데이터 키 캐싱은 특정 애플리케이션 및 워크로드에 매우 효과적인 최적화 전략입니다. 그러나 약간의 위험이 수반되므로, 상황에 얼마나 효과적일 수 있는지 판단한 다음 이점이 위험보다 큰지 판단하는 것이 중요합니다.

데이터 키 캐싱은 데이터 키를 재사용하기 때문에 가장 확실한 효과는 새 데이터 키를 생성하기 위한 호출 횟수를 줄이는 것입니다. 데이터 키 캐싱이 구현되면는 초기 데이터 키를 생성하고 캐시가 AWS KMS `GenerateDataKey` 누락될 때만 작업을 AWS Encryption SDK 호출합니다. 그러나 캐싱은 동일한 암호화 컨텍스트 및 알고리즘 세트를 포함하여 동일한 특성을 가진 수많은 데이터 키를 생성하는 애플리케이션에서만 성능을 눈에 띄게 개선합니다.

의 구현 AWS Encryption SDK 이 실제로 캐시의 데이터 키를 사용하고 있는지 확인하려면 다음 기술을 시도해 보세요.

- 마스터 키 인프라의 로그에서 호출 빈도를 확인하여 새 데이터 키를 만듭니다. 데이터 키 캐싱이 효과적이면 새 키를 만드는 호출 횟수가 눈에 띄게 떨어집니다. 예를 들어 AWS KMS 마스터 키 또는 키링을 사용하는 경우 CloudTrail 로그에서 GenerateDataKey 호출을 검색합니다.
- 다양한 암호화 요청에 대한 응답으로 AWS Encryption SDK 에서 반환되는 암호화된 메시지를 비교합니다. 예를 들어를 사용하는 경우 다른 암호화 호출의 ParsedCiphertext 객체를 AWS Encryption SDK for Java비교합니다. AWS Encryption SDK for JavaScript에서 MessageHeader의 `encryptedDataKeys` 속성 내용을 비교합니다. 데이터 키를 재사용하면 암호화된 메시지의 암호화된 데이터 키가 동일합니다.

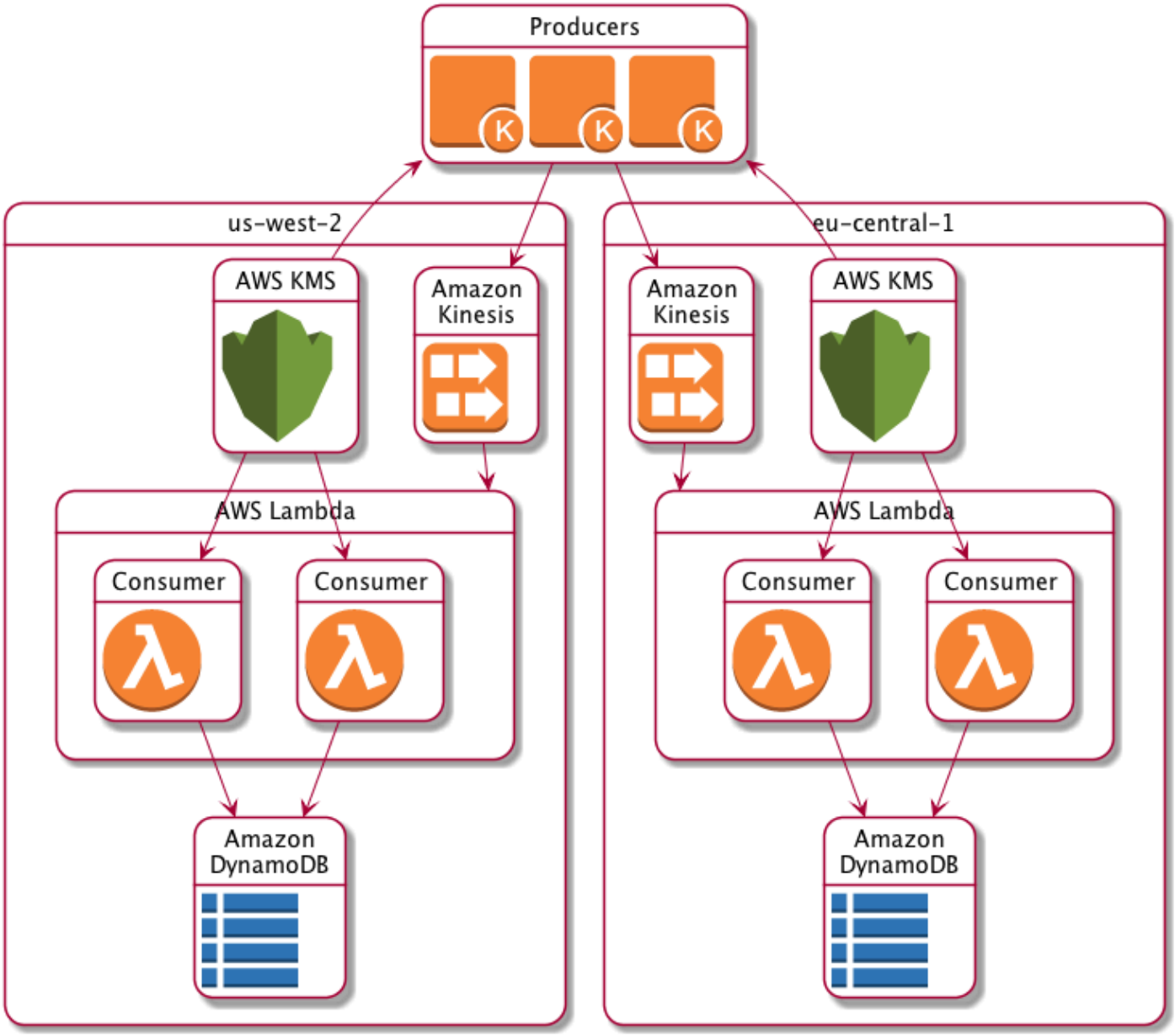
# 데이터 키 캐싱 예제

이 예제에서는 데이터 키 캐싱과 로컬 캐시를 함께 사용하여 다양한 리전에서 여러 디바이스에 의해 생성되는 데이터가 암호화 및 저장되는 애플리케이션의 속도를 가속화합니다.

이 시나리오에서는 여러 데이터 생산자가 데이터를 생성하고 암호화하여 각 리전의 Kinesis 스트림에 씁니다. AWS Lambda 함수(소비자)는 스트림을 복호화하여 일반 텍스트 데이터를 해당 리전의

DynamoDB 테이블에 씁니다. 데이터 생산자와 소비자는 AWS Encryption SDK 및 AWS KMS 마스터 키 공급자를 사용합니다. KMS에 대한 호출을 줄이기 위해 각 생산자와 소비자는 자체 로컬 캐시를 보유하고 있습니다.

Java 및 Python에서 이러한 예제의 소스 코드를 찾을 수 있습니다. 샘플에는 샘플에 대한 리소스를 정의하는 CloudFormation 템플릿도 포함되어 있습니다.




## 로컬 캐시 결과

아래 표에서는 로컬 캐시가 이 예제의 총 KMS 호출 수(리전별 초당)를 원래 값의 1%로 줄이는 것을 보여줍니다.

생산자 요청

| | 클라이언트당 초당 요청 | | | 리전당 클라이언트 | 리전당 초당 평균 요청 |
|---|---|---|---|---|---|
| | 데이터 키 생성(us-west-2) | 데이터 키 암호화(eu-central-1) | 총계(리전별) | | |
| 캐시 없음 | 1 | 1 | 1 | 500 | 500 |
| 로컬 캐시 | 1rps/100회 사용 | 1rps/100회 사용 | 1rps/100회 사용 | 500 | 5 |

소비자 요청

| | 클라이언트당 초당 요청 | | | 리전당 클라이언트 | 리전당 초당 평균 요청 |
|---|---|---|---|---|---|
| | 데이터 키 복호화 | 생산자 | 합계 | | |
| 캐시 없음 | 생산자당 1rps | 500 | 500 | 2 | 1,000 |
| 로컬 캐시 | 생산자당 1rps/100회 사용 | 500 | 5 | 2 | 10 |

## 데이터 키 캐싱 예제 코드

이 코드 샘플은 Java 및 Python에서 로컬 캐시를 사용한 데이터 키 캐싱을 간단하게 구현합니다. 이 코드는 로컬 캐시의 두 인스턴스를 생성합니다. 하나는 데이터를 암호화하는 데이터 생산자용이고 다른 하나는 데이터를 해독하는 데이터 소비자(AWS Lambda 함수)용입니다. 각 언어의 데이터 키 캐싱 구현에 대한 자세한 내용은 AWS Encryption SDK에 대한 Javadoc 및 Python 설명서를 참조하세요.

데이터 키 캐싱은가 AWS Encryption SDK 지원하는 모든 프로그래밍 언어에 사용할 수 있습니다.

에서 데이터 키 캐싱을 사용하는 전체 예제와 테스트된 예제는 다음을 AWS Encryption SDK참조하세요.

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java
- JavaScript 브라우저: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

### 생산자

생산자는 맵을 가져와 JSON으로 변환하고, AWS Encryption SDK 를 사용하여 암호화하고, 사이퍼텍스트 레코드를 각의 Kinesis 스트림으로 푸시합니다 AWS 리전.

이 코드는 캐싱 암호 자료 관리자(캐싱 CMM)를 정의해서 로컬 캐시 및 기본 AWS KMS 마스터 키 공급자와 연결합니다. 캐싱 CMM은 마스터 키 공급자의 데이터 키(및 관련 암호화 자료)를 캐시합니다. 또한 SDK를 대신하여 캐시와 상호 작용하며 사용자가 설정한 보안 임계값을 적용합니다.

암호화 메서드 호출은 일반 암호화 자료 관리자(CMM) 또는 마스터 키 공급자 대신 캐싱 CMM을 지정하므로 암호화에는 데이터 키 캐싱이 사용됩니다.

Java

다음 예제에서는의 버전 2.x를 사용합니다 AWS Encryption SDK for Java. 버전 3.x는 데이터 키 캐싱 CMM을 더 AWS Encryption SDK for Java 이상 사용하지 않습니다. 버전 3.x에서는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 사용할 수도 있습니다.

```
/*
 * 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.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_ENTRY_USES = 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()
                .credentialsProvider(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.

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

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

or in the "license" file accompanying this file. This file is distributed on an "AS
 IS" BASIS,
```

```
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
specific language governing permissions and limitations under the License.
"""
import json
import uuid

from aws_encryption_sdk import 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(
                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
            )
```

## 소비자

데이터 소비자는 Kinesis 이벤트에 의해 트리거되는 AWS Lambda 함수입니다. 각 레코드를 복호화하고 역직렬화하며 일반 텍스트 레코드를 동일 리전의 Amazon DynamoDB 테이블에 씁니다.

생산자 코드와 마찬가지로 소비자 코드는 복호화 메서드를 호출할 때 캐싱 암호 자료 관리자(캐싱 CMM)를 사용하여 데이터 키를 캐싱할 수 있도록 합니다.

Java 코드는 지정된를 사용하여 엄격 모드에서 마스터 키 공급자를 빌드합니다 AWS KMS key. 복호화 시에는 엄격 모드가 반드시 필요하지 않지만 모범 사례입니다. Python 코드는 검색 모드를 사용합니다. 이 모드를 사용하면가 데이터 키를 암호화한 래핑 키를 AWS Encryption SDK 사용하여 복호화할 수 있습니다.

Java

다음 예제에서는 버전 2.x를 사용합니다 AWS Encryption SDK for Java. 버전 3.x는 데이터 키 캐싱 CMM을 더 AWS Encryption SDK for Java 이상 사용하지 않습니다. 버전 3.x에서는 대체 암호화 자료 캐싱 솔루션인 AWS KMS 계층적 키링을 사용할 수도 있습니다.

이 코드는 엄격 모드에서 복호화하기 위한 마스터 키 공급자를 생성합니다. 는 AWS Encryption SDK AWS KMS keys 사용자가 지정한 만 사용하여 메시지를 복호화할 수 있습니다.

```
/*
 * 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)
            .build();

        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

이 Python 코드는 검색 모드에서 마스터 키 공급자를 사용하여 복호화합니다. 이렇게 하면 AWS Encryption SDK 가 데이터 키를 암호화한 래핑 키를 사용하여 복호화할 수 있습니다. 복호화에 사용할 수 있는 래핑 키를 지정하는 엄격 모드가 모범 사례입니다.

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

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

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

or in the "license" file accompanying this file. This file is distributed on an "AS
 IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
specific language governing permissions and limitations under the License.
"""
import base64
import json
import logging
import os

from aws_encryption_sdk import 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
    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!')

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

# 데이터 키 캐싱 예제: CloudFormation template

이 CloudFormation 템플릿은 데이터 키 캐싱 예제를 재현하는 데 필요한 모든 AWS 리소스를 설정합니다.

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": {
                                "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"
                        }
                    ]
                },
                "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": [
                                        "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::Sub": "alias/${KeyAliasSuffix}"
            },
            "TargetKeyId": {
                "Ref": "RegionKinesisCMK"
            }
        }
```

```
        }
    }
}
```

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}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
```

```
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    PythonLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Python consumer
            Runtime: python2.7
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt PythonLambdaRole.Arn
            Handler:
 aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref PythonLambdaS3Key
                S3ObjectVersion: !Ref PythonLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref PythonLambdaOutputTable
    PythonLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref PythonLambdaFunction
            StartingPosition: TRIM_HORIZON
    JavaLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
```

```
    JavaLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: JavaLambdaAccess
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    JavaLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
```

```
            Description: Java consumer
            Runtime: java8
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt JavaLambdaRole.Arn
            Handler:
 com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref JavaLambdaS3Key
                S3ObjectVersion: !Ref JavaLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref JavaLambdaOutputTable
                    CMK_ARN: !GetAtt RegionKinesisCMK.Arn
    JavaLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref JavaLambdaFunction
            StartingPosition: TRIM_HORIZON
    RegionKinesisCMK:
        Type: AWS::KMS::Key
        Properties:
            Description: Used to encrypt data passing through Kinesis Stream in this
 region
            Enabled: true
            KeyPolicy:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            AWS: !Sub arn:aws:iam::${AWS::AccountId}:root
                        Action:
                            # Data plane actions
                            - kms:Encrypt
                            - kms:GenerateDataKey
                            # Control plane actions
                            - kms:CreateAlias
                            - kms:DeleteAlias
```

```
                            - kms:DescribeKey
                            - kms:DisableKey
                            - kms:EnableKey
                            - kms:PutKeyPolicy
                            - kms:ScheduleKeyDeletion
                            - kms:UpdateAlias
                            - kms:UpdateKeyDescription
                        Resource: '*'
                    -
                        Effect: Allow
                        Principal:
                            AWS:
                                - !GetAtt PythonLambdaRole.Arn
                                - !GetAtt JavaLambdaRole.Arn
                        Action: kms:Decrypt
                        Resource: '*'
    RegionKinesisCMKAlias:
        Type: AWS::KMS::Alias
        Properties:
            AliasName: !Sub alias/${KeyAliasSuffix}
            TargetKeyId: !Ref RegionKinesisCMK
```

# 의 버전 AWS Encryption SDK

AWS Encryption SDK 언어 구현에서는 의미 체계 버전 관리를 사용하여 각 릴리스의 변경 규모를 더 쉽게 식별할 수 있습니다. 메이저 버전 번호의 변경(예: 1.x.x에서 2.x.x로 변경)은 코드 변경 및 계획된 배포가 필요할 수 있는 중요한 변경 사항을 나타냅니다. 새 버전의 변경 사항이 모든 사용 사례에 영향을 미치지는 않을 수 있습니다. 릴리스 정보를 검토하여 영향을 받는지 확인하세요. 마이너 버전 번호의 변경(예: x.1.x에서 x.2.x로 변경)은 항상 이하 버전과 호환되지만 더 이상 사용되지 않는 요소가 포함될 수 있습니다.

가능하면 선택한 프로그래밍 언어로의 최신 버전을 사용 AWS Encryption SDK 하세요. 각 버전의 유지 관리 및 지원 정책은 프로그래밍 언어 구현에 따라 다릅니다. 선호하는 프로그래밍 언어로 지원되는 버전에 대한 자세한 내용은 해당 언어의 GitHub 리포지토리에서 `SUPPORT_POLICY.rst` 파일을 참조하세요.

업그레이드에 암호화 또는 복호화 오류를 방지하기 위해 특별한 구성이 필요한 새 기능이 포함된 경우 중간 버전과 자세한 사용 지침을 제공합니다. 예를 들어 버전 1.7.x 및 1.8.x는 1.7.x 이하 버전에서 2.0.x 이상 버전으로 업그레이드하는 데 도움이 되는 전환 버전으로 설계되었습니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.

> **Note**
>
> 버전 번호의 x는 메이저 버전과 마이너 버전의 모든 패치를 나타냅니다. 예를 들어 버전 1.7.x는 1.7.1과 1.7.9를 포함하여 1.7로 시작하는 모든 버전을 나타냅니다.
> 새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

다음 표에서는 각 프로그래밍 언어에 대해 지원되는 버전 간의 주요 차이점에 AWS Encryption SDK 대한 개요를 제공합니다.

## C

모든 변경 사항에 대한 자세한 설명은 GitHub의 aws-encryption-sdk-c 리포지토리에서 CHANGELOG.md 참조하십시오.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Support 단계 |
| | 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 버전 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see 버전 2.0.x. | 정식 출시 (GA) |
| | 2.2 | Improvements to the message decryption process. | |
| | 2.3 | Adds support for AWS KMS multi-Region keys. | |

## C#/.NET

모든 변경 사항에 대한 자세한 설명은 GitHub의 `aws-encryption-sdk-net`리포지토리에서 CHANGELOG.md 참조하십시오.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 3.x | 3.1.0 | Initial release. | End-of-Support |

| | | | |
|---|---|---|---|
| | | | AWS Encryption SDK for .NET 버전 3.x가 지원 종료에 들어갔습니다. 4.x로 업그레이드하세요. |
| 4.x | 4.0 | Adds support for the AWS KMS Hierarchical keyring, the required encryption context CMM, and asymmetric RSA AWS KMS keyrings. | 정식 출시 (GA) |

## 명령줄 인터페이스(CLI)

모든 변경 사항에 대한 자세한 설명은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 AWS 암호화 CLI 버전 및 CHANGELOG.rst를 참조하세요.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Support 단계 |
| | 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 버전 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see 버전 2.0.x. | End-of-Support 단계 |

| | | | |
|---|---|---|---|
| | 2.1 | `--discovery` 파라미터를 제거하고 `--wrapping-keys` 파라미터의 `discovery` 속성으로 바꿉니다.<br><br>AWS Encryption CLI 버전 2.1.0은 다른 프로그래밍 언어의 버전 2.0과 동일합니다. | |
| | 2.2 | Improvements to the message decryption process. | |
| 3.x | 3.0 | Adds support for AWS KMS multi-Region keys. | End-of-Support 단계 |
| 4.x | 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. | 정식 출시 (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

모든 변경 사항에 대한 자세한 설명은 GitHub의 aws-encryption-sdk-java 리포지토리에서 CHANGELOG.rst를 참조하세요.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Support 단계 |
| | 1.3 | Adds support for cryptographic materials manager and data key caching. Moved to deterministic IV generation. | |
| | 1.6.1 | `AwsCrypto.encryptString()` 및를 사용 중지`AwsCrypto.decryptString()` 하고 `AwsCrypto.encryptData()` 및 로 바꿉니다`AwsCrypto.decryptData()` . | |

| | | | |
|---|---|---|---|
| | 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 버전 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see 버전 2.0.x. | 일반 가용성(GA)<br><br>버전 2.x는 2024년에 유지 관리 모드로 전환 AWS Encryption SDK for Java 됩니다. |
| | 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. | |

| | | | |
|---|---|---|---|
| 3.x | 3.0 | 를 재료 공급자 라이브러리(MPL) AWS Encryption SDK for Java 와 통합합니다. https://github.com/aws/aws-cryptographic-material-providers-library<br><br>대칭 및 비대칭 RSA AWS KMS 키링, AWS KMS ECDH 키링, AWS KMS 계층적 키링, 원시 AES 키링, 원시 RSA 키링, 원시 ECDH 키링, 다중 키링 및 필요한 암호화 컨텍스트 CMM에 대한 지원을 추가합니다. | 정식 출시 (GA) |

## Go

모든 변경 사항에 대한 자세한 설명은 GitHub의 aws-encryption-sdk 리포지토리의 Go 디렉터리에 있는 CHANGELOG.md 참조하십시오.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 0.1.x | 0.1.0 | Initial release. | 정식 출시 (GA) |

## JavaScript

모든 변경 사항에 대한 자세한 설명은 GitHub의 aws-encryption-sdk-javascript 리포지토리에서 CHANGELOG.md 참조하세요.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Support 단계 |
| | 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 버전 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see 버전 2.0.x. | End-of-Support 단계 |
| | 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. | Maintenance<br><br>버전 3.x에 대한 지원 AWS Encryption SDK for JavaScript 은 2024년 1월 17일에 종료됩니다. |
| 4.x | 4.0 | Requires version 3 of the AWS Encryption SDK for JavaScript's `kms-#####` to | 정식 출시 (GA) |

| | | | |
|---|---|---|---|
| | | use the AWS KMS keyring. | |

## Python

모든 변경 사항에 대한 자세한 설명은 GitHub의 aws-encryption-sdk-python 리포지토리에서 CHANGELOG.rst를 참조하세요.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Support 단계 |
| | 1.3 | Adds support for cryptographic materials manager and data key caching. Moved to deterministic IV generation. | |
| | 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 버전 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see 버전 2.0.x. | End-of-Support 단계 |
| | 2.2 | Improvements to the message decryption process. | |

| | | | |
|---|---|---|---|
| | 2.3 | Adds support for AWS KMS multi-Region keys. | |
| 3.x | 3.0 | 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. | 정식 출시 (GA) |
| 4.x | 4.0 | 를 재료 공급자 라이브러리(MPL) AWS Encryption SDK for Python 와 통합합니다. https://github.com/aws/aws-cryptographic-material-providers-library | 정식 출시 (GA) |

## Rust

모든 변경 사항에 대한 자세한 설명은 GitHub의 aws-encryption-sdk 리포지토리 Rust 디렉터리에서 CHANGELOG.md 참조하세요.

| 메이저 버전 | 세부 정보 | | SDK 메이저 버전 수명 주기 단계 |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | 정식 출시 (GA) |

# 버전 세부 정보

다음 목록은 지원되는 AWS Encryption SDK버전 간의 주요 차이점을 설명합니다.

주제



## 1.7.x 이하 버전

> **Note**
>
> 의 모든 1.x.x 버전 AWS Encryption SDK 은 end-of-support 단계에 있습니다. 가능한 한 빨리 프로그래밍 언어 AWS Encryption SDK 에 사용할 수 있는 최신 버전의 로 업그레이드합니다. 1.7.x 이전 AWS Encryption SDK 버전에서 업그레이드하려면 먼저 1.7.x로 업그레이드해야 합니다. 자세한 내용은 마이그레이션 AWS Encryption SDK을 참조하세요.

1.7.x AWS Encryption SDK 이전 버전은 Galois/Counter Mode(AES-GCM)의 고급 암호화 표준 알고리즘을 사용한 암호화, HMAC 기반 extract-and-expand 키 유도 함수(HKDF), 서명, 256비트 암호화 키를 포함한 중요한 보안 기능을 제공합니다. 하지만 이러한 버전은 키 커밋을 포함하여 권장되는 모범 사례를 지원하지 않습니다.

## 버전 1.7.x

> **Note**
>
> 의 모든 1.x.x 버전 AWS Encryption SDK 은 end-of-support 단계에 있습니다.

버전 1.7.x는 이전 버전의 사용자가 버전 2.0.x 이상으로 업그레이드 AWS Encryption SDK 할 수 있도록 설계되었습니다. 를 처음 사용하는 경우이 버전을 건너뛰고 프로그래밍 언어에서 사용 가능한 최신 버전으로 시작할 AWS Encryption SDK수 있습니다.

버전 1.7.x는 이하 버전과 완벽하게 호환되며, 중대한 변경 사항을 포함하고 있거나 AWS Encryption SDK의 동작을 변경하지 않습니다. 또한 이상 버전과도 호환되며, 버전 2.0.x와 호환되도록 코드를 업데이트할 수 있습니다. 여기에는 새 기능이 포함되어 있지만 완전히 활성화되지는 않습니다. 또한 준비가 될 때까지 모든 새 기능을 즉시 적용하지 못하도록 하는 구성 값이 필요합니다.

버전 1.7.x에는 다음과 같은 변경 사항이 포함됩니다.

AWS KMS 마스터 키 공급자 업데이트(필수)

버전 1.7.x는 엄격한 또는 검색 모드에서 AWS KMS 마스터 키 공급자를 AWS Encryption SDK for Python 명시적으로 생성하는 AWS Encryption SDK for Java 및에 새로운 생성자를 도입합니다. 이 버전은 AWS Encryption SDK 명령줄 인터페이스(CLI)에 유사한 변경 사항을 추가합니다. 자세한 내용은 AWS KMS 마스터 키 공급자 업데이트을 참조하세요.

- 엄격 모드에서 AWS KMS 마스터 키 공급자는 래핑 키 목록을 필요로 하며, 사용자가 지정한 래핑 키로만 암호화하고 복호화합니다. 엄격 모드는 사용하려는 래핑 키를 사용하고 있음을 보장하는 AWS Encryption SDK 모범 사례입니다.
- 검색 모드에서 AWS KMS 마스터 키 공급자는 어떤 래핑 키도 사용하지 않습니다. 암호화에는 래핑 키를 사용할 수 없습니다. 복호화에는 모든 래핑 키를 사용하여 암호화된 데이터 키를 복호화할 수 있습니다. 다만 복호화에 사용되는 래핑 키를 특정 AWS 계정에 있는 래핑 키로 제한할 수 있습니다. 계정 필터링은 선택 사항이지만 권장되는 모범 사례입니다.

이전 버전의 AWS KMS 마스터 키 공급자를 생성하는 생성자는 버전 1.7.x에서는 더 이상 사용되지 않으며 버전 2.0.x에서는 제거됩니다. 이러한 생성자는 사용자가 지정한 래핑 키를 사용하여 암호화하는 마스터 키 공급자를 인스턴스화합니다. 하지만 지정된 래핑 키에 관계없이 데이터 키를 암호화한 래핑 키를 사용하여 해당 암호화된 데이터 키를 복호화합니다. 사용자는 다른 AWS 계정 및 리전 AWS KMS keys 을 포함하여 사용하지 않을 래핑 키를 사용하여 의도치 않게 메시지를 복호화할 수 있습니다.

AWS KMS 마스터 키의 생성자는 변경되지 않습니다. 암호화 및 복호화 시 AWS KMS 마스터 키는 AWS KMS key 사용자가 지정한 만 사용합니다.

AWS KMS 키링 업데이트(선택 사항)

버전 1.7.x는 AWS KMS 검색 키링을 특정으로 제한하는 AWS Encryption SDK for C 및 AWS Encryption SDK for JavaScript 구현에 새 필터를 추가합니다 AWS 계정. 이 새로운 계정 필터는 선택 사항이지만 권장되는 모범 사례입니다. 자세한 내용은 AWS KMS 키링 업데이트을 참조하세요.

AWS KMS 키링의 생성자는 변경되지 않습니다. 표준 AWS KMS 키링은 엄격한 모드에서 마스터 키 공급자처럼 동작합니다. AWS KMS 검색 키링은 검색 모드에서 명시적으로 생성됩니다.

AWS KMS 복호화에 키 ID 전달

버전 1.7.x부터 암호화된 데이터 키를 복호화할 때는 AWS Encryption SDK 항상를 AWS KMS 복호화 작업 호출 AWS KMS key 에 지정합니다. 는 암호화된 각 데이터 키의 메타데이터 AWS KMS key 에서의 키 ID 값을 AWS Encryption SDK 가져옵니다. 이 기능에는 코드 변경이 필요하지 않습니다.

대칭 암호화 KMS 키로 암호화된 사이퍼텍스트를 해독하는 데의 키 ID를 지정할 AWS KMS key 필요는 없지만 AWS KMS 모범 사례입니다. 이 방법은 키 공급자에서 래핑 키를 지정하는 것과 마찬가지로가 사용하려는 래핑 키를 사용해서 AWS KMS 만 복호화하도록 합니다.

키 커밋으로 사이퍼텍스트 복호화

버전 1.7.x는 키 커밋 유무에 관계없이 암호화된 사이퍼텍스트를 복호화할 수 있습니다. 하지만 키 커밋으로는 사이퍼텍스트를 암호화할 수 없습니다. 이 속성을 사용하면 사이퍼텍스트가 발생하기 전에 키 커밋으로 암호화된 사이퍼텍스트를 복호화할 수 있는 애플리케이션을 완전히 배포할 수 있습니다. 이 버전은 키 커밋 없이 암호화된 메시지를 복호화하므로 사이퍼텍스트를 다시 암호화할 필요가 없습니다.

이 동작을 구현하기 위해 버전 1.7.x에는가 키 커밋으로 암호화 또는 복호화할 수 있는지 여부를 결정하는 새로운 커밋 정책 구성 설정이 포함되어 있습니다. AWS Encryption SDK 버전 1.7.x에서는 커밋 정책에서 유일하게 유효한 값인 `ForbidEncryptAllowDecrypt`가 모든 암호화 및 복호화 작업에 사용됩니다. 이 값은 AWS Encryption SDK 가 키 커밋이 포함된 새 알고리즘 제품군 중 하나를 사용하여 암호화할 수 없도록 합니다. 이를 통해는 키 커밋 유무에 관계없이 사이퍼텍스트를 복호화 AWS Encryption SDK 할 수 있습니다.

버전 1.7.x에는 유효한 커밋 정책 값이 하나뿐이지만 이번 릴리스에 도입된 새 API를 사용할 때는 이 값을 명시적으로 설정할 수 있어야 합니다. 이 값을 명시적으로 설정하면 버전 2.1.x로 업그레이드할 때 커밋 정책이 자동으로 `require-encrypt-require-decrypt`로 변경되는 것을 방지할 수 있습니다. 대신, 단계적으로 커밋 정책을 마이그레이션할 수 있습니다.

키 커밋이 포함된 알고리즘 제품군

버전 1.7.x에는 키 커밋을 지원하는 두 개의 새로운 알고리즘 제품군이 포함되어 있습니다. 하나는 서명이 포함되어 있고 다른 하나는 서명을 포함하고 있지 않습니다. 이전에 지원되던 알고리즘 제품군과 마찬가지로 이 두 가지 새로운 알고리즘 제품군에는 AES-GCM을 사용한 암호화, 256비트 암호화 키, HMAC 기반 추출 및 확장 키 유도 함수(HKDF)가 포함되어 있습니다.

하지만 암호화에 사용되는 기본 알고리즘 제품군은 변경되지 않습니다. 이러한 알고리즘 제품군은 버전 1.7.x에 추가되어 사용자의 애플리케이션이 버전 2.0.x 이상에서 알고리즘을 사용할 수 있도록 지원합니다.

CMM 구현 변경 사항

버전 1.7.x에 키 커밋을 지원하기 위해 기본 암호화 구성 요소 관리자(CMM) 인터페이스가 변경되었습니다. 이 변경 사항은 사용자 지정 CMM을 작성한 경우에만 적용됩니다. 자세한 내용은 사용하는 프로그래밍 언어의 API 설명서 또는 GitHub 리포지토리를 참조하세요.

## 버전 2.0.x

버전 2.0.x는 지정된 래핑 키 및 키 커밋 AWS Encryption SDK을 포함하여에서 제공되는 새로운 보안 기능을 지원합니다. 이러한 기능을 지원하기 위해 버전 2.0.x에는 AWS Encryption SDK의 모든 이하 버전에 대한 주요 변경 사항이 포함되어 있습니다. 버전 1.7.x를 배포하여 이러한 변경 사항에 대비할 수 있습니다. 버전 2.0.x에는 다음과 같은 추가 및 변경 사항과 함께 버전 1.7.x에 도입된 모든 새로운 기능이 포함되어 있습니다.

> **Note**
>
> 의 버전 2.x.x AWS Encryption SDK for Python AWS Encryption SDK for JavaScript및 AWS 암호화 CLI는 end-of-support 단계에 있습니다.
> 원하는 프로그래밍 언어로이 AWS Encryption SDK 버전을 지원하고 유지 관리하는 방법에 대한 자세한 내용은 GitHub 리포지토리의 `SUPPORT_POLICY.rst` 파일을 참조하세요.

AWS KMS 마스터 키 공급자

버전 1.7.x에서 더 이상 사용되지 않는 원래 AWS KMS 마스터 키 공급자 생성자는 버전 2.0.x에서 제거됩니다. 엄격 모드 또는 검색 모드에서 AWS KMS 마스터 키 공급자를 명시적으로 구성해야 합니다.

키 커밋으로 사이퍼텍스트 암호화 및 복호화

버전 2.0.x는 키 커밋 유무에 관계없이 사이퍼텍스트를 암호화 및 복호화할 수 있습니다. 해당 동작은 커밋 정책 설정에 따라 결정됩니다. 기본적으로 항상 키 커밋을 사용하여 암호화하고 키 커밋으로 암호화된 사이퍼텍스트만 복호화합니다. 커밋 정책을 변경하지 않는 한 AWS Encryption SDK는 버전 1.7.x를 포함한 이하 버전의 AWS Encryption SDK로 암호화된 사이퍼텍스트를 복호화하지 않습니다.

> **Important**
>
> 기본적으로 버전 2.0.x는 키 커밋 없이 암호화된 사이퍼텍스트를 복호화하지 않습니다. 애플리케이션에서 키 커밋 없이 암호화된 사이퍼텍스트가 발생한 경우, `AllowDecrypt`로 커밋 정책 값을 설정하세요.

버전 2.0.x에서, 커밋 정책 설정에는 다음과 같은 세 가지 유효한 값이 있습니다.

- `ForbidEncryptAllowDecrypt` - AWS Encryption SDK 는 키 커밋으로 암호화할 수 없습니다. 암호화된 사이퍼텍스트를 키 커밋 사용 여부와 관계없이 복호화할 수 있습니다.
- `RequireEncryptAllowDecrypt` - AWS Encryption SDK 는 키 커밋으로 암호화해야 합니다. 암호화된 사이퍼텍스트를 키 커밋 사용 여부와 관계없이 복호화할 수 있습니다.
- `RequireEncryptRequireDecrypt` (기본값) -는 키 커밋으로 암호화 AWS Encryption SDK 해야 합니다. 키 커밋이 있는 사이퍼텍스트만 복호화합니다.

이전 버전의에서 버전 2.0.x AWS Encryption SDK 로 마이그레이션하는 경우 애플리케이션에서 발생할 수 있는 모든 기존 사이퍼텍스트를 복호화할 수 있도록 커밋 정책을 값으로 설정합니다. 시간이 지남에 따라 이 설정을 조정할 가능성이 높습니다.

## 버전 2.2.x

디지털 서명 및 암호화된 데이터 키 제한에 대한 지원이 추가되었습니다.

> **Note**
>
> 버전 2.x.x 및 AWS Encryption SDK for Python AWS Encryption SDK for JavaScript AWS 암호화 CLI는 end-of-support 단계에 있습니다.
> 원하는 프로그래밍 언어로이 AWS Encryption SDK 버전을 지원하고 유지 관리하는 방법에 대한 자세한 내용은 GitHub 리포지토리의 `SUPPORT_POLICY.rst` 파일을 참조하세요.

### 디지털 서명

복호화 시 디지털 서명 처리를 개선하기 위해 에는 다음 기능이 AWS Encryption SDK 포함되어 있습니다.

- 비스트리밍 모드 - 디지털 서명이 있는 경우 디지털 서명 확인을 포함하여 모든 입력이 처리된 후에만 일반 텍스트를 반환합니다. 이 기능을 사용하면 디지털 서명을 확인하기 전에 일반 텍스트

를 사용할 수 없습니다. 디지털 서명으로 암호화된 데이터(기본 알고리즘 제품군)를 복호화할 때마다 이 기능을 사용하세요. 예를 들어 AWS Encryption CLI는 항상 스트리밍 모드에서 데이터를 처리하기 때문에 디지털 서명으로 사이퍼텍스트를 복호화할 때 `- -buffer` 파라미터를 사용합니다.

- 무서명 전용 복호화 모드 - 이 기능은 서명되지 않은 사이퍼텍스트만 복호화합니다. 복호화 시 사이퍼텍스트에 디지털 서명이 있는 경우 작업이 실패합니다. 이 기능을 사용하면 서명을 확인하기 전에 서명된 메시지의 일반 텍스트를 실수로 처리하는 것을 방지할 수 있습니다.

암호화된 데이터 키 제한

암호화된 메시지의 암호화된 데이터 키의 수를 제한할 수 있습니다. 이 기능을 사용하면 암호화할 때 잘못 구성된 마스터 키 공급자 또는 키링을 탐지하거나 복호화 시 악성 사이퍼텍스트를 식별할 수 있습니다.

신뢰할 수 없는 소스의 메시지를 복호화할 때는 암호화된 데이터 키를 제한해야 합니다. 이렇게 하면 키 인프라에 대하여 불필요하며 비용이 높고 잠재적으로 소모적인 호출을 방지합니다.

## 버전 2.3.x

AWS KMS 다중 리전 키에 대한 지원을 추가합니다. 자세한 내용은 다중 리전 사용 AWS KMS keys을 참조하세요.

> **Note**
>
> AWS Encryption CLI는 버전 3.0.x부터 다중 리전 키를 지원합니다.
> 의 버전 2.x.x AWS Encryption SDK for Python AWS Encryption SDK for JavaScript및 AWS 암호화 CLI는 end-of-support 단계에 있습니다.
> 원하는 프로그래밍 언어로이 AWS Encryption SDK 버전을 지원하고 유지 관리하는 방법에 대한 자세한 내용은 GitHub 리포지토리의 `SUPPORT_POLICY.rst` 파일을 참조하세요.

# 마이그레이션 AWS Encryption SDK

는 여러 상호 운용 가능한 프로그래밍 언어 구현을 AWS Encryption SDK 지원하며, 각 구현은 GitHub의 오픈 소스 리포지토리에서 개발됩니다. 가장 좋은 방법은 각 언어에 AWS Encryption SDK 대해 최신 버전의를 사용하는 것입니다.

버전 2.0.x 이상에서 최신 버전으로 안전하게 업그레이드 AWS Encryption SDK 할 수 있습니다. 그러나의 2.0.x 버전에는 중요한 새로운 보안 기능이 AWS Encryption SDK 도입되었으며, 그 중 일부는 주요 변경 사항입니다. 1.7.x 이하 버전에서 2.0.x 및 이상 버전으로 업그레이드하려면 먼저 최신 1.x 버전으로 업그레이드해야 합니다. 이 섹션의 항목은 변경 사항을 이해하고, 애플리케이션에 맞는 올바른 버전을 선택하고, AWS Encryption SDK의 최신 버전으로 안전하고 성공적으로 마이그레이션하는 데 도움이 되도록 설계되었습니다.

의 중요 버전에 대한 자세한 내용은 섹션을 AWS Encryption SDK참조하세요의 버전 AWS Encryption SDK.

**Important**

1.7.x 이하 버전에서 최신 1.x 버전으로 먼저 업그레이드하지 않고 곧바로 2.0.x 이상 버전으로 업그레이드해서는 안됩니다. 버전 2.0.x 이상으로 직접 업그레이드하고 모든 새 기능을 즉시 활성화하면 AWS Encryption SDK 는 이전 버전의에서 암호화된 사이퍼텍스트를 해독할 수 없습니다 AWS Encryption SDK.

**Note**

AWS Encryption SDK for .NET의 최신 버전은 버전 3.0.x입니다. AWS Encryption SDK for .NET의 모든 버전은의 2.0.x에 도입된 보안 모범 사례를 지원합니다 AWS Encryption SDK. 코드나 데이터를 변경하지 않고도 최신 버전으로 안전하게 업그레이드할 수 있습니다.
AWS Encryption CLI:이 마이그레이션 가이드를 읽을 때 AWS Encryption CLI 1.8.x에 대한 1.7.x 마이그레이션 지침을 사용하고 AWS Encryption CLI 2.1.x에 대한 2.0.x 마이그레이션 지침을 사용합니다. 자세한 내용은 AWS 암호화 CLI 버전을 참조하세요.
새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

신규 사용자

를 처음 사용하는 경우 프로그래밍 언어에 AWS Encryption SDK 맞는 최신 버전의를 AWS Encryption SDK설치합니다. 기본값은의 서명 AWS Encryption SDK, 키 파생 및 키 커밋을 사용한 암호화를 포함하여의 모든 보안 기능을 활성화합니다. AWS Encryption SDK

현재 사용자

가능한 한 빨리 현재 버전에서 사용 가능한 최신 버전으로 업그레이드하는 것이 좋습니다. 의 모든 1.x 버전 AWS Encryption SDK 은 일부 프로그래밍 언어의 이후 버전과 마찬가지로 end-of-support 단계에 있습니다. 사용 중인 프로그래밍 언어의 AWS Encryption SDK 지원 및 유지 관리 상태에 대한 자세한 내용은 지원 및 유지 관리 섹션을 참조하세요.

AWS Encryption SDK 버전 2.0.x 이상에서는 데이터를 보호하는 데 도움이 되는 새로운 보안 기능을 제공합니다. 그러나 AWS Encryption SDK 버전 2.0.x에는 이전 버전과 호환되지 않는 주요 변경 사항이 포함되어 있습니다. 안전하게 전환하려면 먼저 현재 버전에서 사용하는 프로그래밍 언어의 최신 1.x 버전으로 마이그레이션하세요. 최신 1.x 버전이 완전히 배포되고 제대로 작동하면 2.0.x 이상 버전으로 안전하게 마이그레이션할 수 있습니다. 이 2단계 프로세스는 특히 분산 애플리케이션에 중요합니다.

이러한 변경 사항의 기반이 되는 AWS Encryption SDK 보안 기능에 대한 자세한 내용은 AWS 보안 블로그의 향상된 클라이언트 측 암호화: 명시적 KeyIds 및 키 커밋을 참조하세요.

에서를 사용하는 데 도움이 필요하신 AWS Encryption SDK for Java 가요 AWS SDK for Java 2.x? 사전 조건을(를) 참조하세요.

주제

- 마이그레이션 및 배포 방법 AWS Encryption SDK
- AWS KMS 마스터 키 공급자 업데이트
- AWS KMS 키링 업데이트
- 커밋 정책 설정
- 최신 버전으로의 마이그레이션 문제 해결

# 마이그레이션 및 배포 방법 AWS Encryption SDK

1.7.x 이전 AWS Encryption SDK 버전에서 버전 2.0.x 이상으로 마이그레이션할 때는 키 커밋을 사용하여 암호화로 안전하게 전환해야 합니다. 그러지 않으면 애플리케이션에서 복호화할 수 없는 사이퍼텍

스트가 만들어집니다. AWS KMS 마스터 키 공급자를 사용하는 경우 엄격 모드 또는 검색 모드에서 마스터 키 공급자를 생성하는 새 생성자로 업데이트해야 합니다.

> **Note**
>
> 이 항목은 AWS Encryption SDK 의 이하 버전에서 2.0.x 이상 버전으로 마이그레이션하는 사용자를 대상으로 합니다. 를 처음 AWS Encryption SDK사용하는 경우 기본 설정으로 사용 가능한 최신 버전을 즉시 사용할 수 있습니다.

읽어야 하는 사이퍼텍스트를 복호화할 수 없는 심각한 상황을 피하려면 여러 단계를 거쳐 마이그레이션하고 배포하는 것이 좋습니다. 다음 단계를 시작하기 전에 각 단계가 완료되고 완전히 배포되었는지 확인하세요. 이는 여러 호스트가 있는 분산 애플리케이션에 특히 중요합니다.

## 1단계: 애플리케이션을 최신 1.x 버전으로 업데이트합니다.

사용 중인 프로그래밍 언어의 최신 1.x 버전으로 업데이트합니다. 2단계를 시작하기 전에 신중하게 테스트하고 변경 내용을 배포한 다음 업데이트가 모든 대상 호스트에 전파되었는지 확인합니다.

> **Important**
>
> 최신 1.x 버전이 AWS Encryption SDK의 1.7.x 또는 이상 버전인지 확인하세요.

의 최신 1.x 버전 AWS Encryption SDK 은의 레거시 버전 AWS Encryption SDK 과 이전 버전과 호환되며 버전 2.0.x 이상과 호환됩니다. 여기에는 버전 2.0.x에 있는 새 기능이 포함되지만 해당 마이그레이션을 위해 설계된 안전한 기본값을 포함해야 합니다. 필요한 경우 AWS KMS 마스터 키 공급자를 업그레이드하고 키 커밋으로 사이퍼텍스트를 복호화할 수 있는 알고리즘 제품군으로 완전히 배포할 수 있습니다.

- 레거시 AWS KMS 마스터 키 제공자의 생성자를 포함하여 더 이상 사용되지 않는 요소를 교체하세요. Python의 경우 지원 중단 경고를 켭니다. 최신 1.x 버전에서 더 이상 사용되지 않는 코드 요소는 2.0.x 이상 버전에서 제거되었습니다.
- 커밋 정책을 `ForbidEncryptAllowDecrypt`로 명시적으로 설정합니다. 최신 1.x 버전에서 유일하게 유효한 값이지만 이 릴리스에 도입된 API를 사용할 때 해당 설정이 필요합니다. 버전 2.0.x 이상 버전으로 마이그레이션할 때 애플리케이션이 키 커밋 없이 암호화된 사이퍼텍스트를 거부하는 것을 방지합니다. 자세한 내용은 the section called “커밋 정책 설정”을 참조하세요.

- AWS KMS 마스터 키 공급자를 사용하는 경우 레거시 마스터 키 공급자를 엄격 모드 및 검색 모드를 지원하는 마스터 키 공급자로 업데이트해야 합니다. 이 업데이트는 AWS Encryption SDK for Java AWS Encryption SDK for Python, 및 AWS 암호화 CLI에 필요합니다. 검색 모드에서 마스터 키 제공자를 사용하는 경우 사용되는 래핑 키를 특정 AWS 계정의 래핑 키로 제한하는 검색 필터를 구현하는 것이 좋습니다. 이 업데이트는 선택 사항이지만 권장되는 모범 사례입니다. 자세한 내용은 AWS KMS 마스터 키 공급자 업데이트을 참조하세요.
- AWS KMS 검색 키링을 사용하는 경우 복호화에 사용되는 래핑 키를 특정 AWS 계정의 래핑 키로 제한하는 검색 필터를 구현하는 것이 좋습니다. 이 업데이트는 선택 사항이지만 권장되는 모범 사례입니다. 자세한 내용은 AWS KMS 키링 업데이트을 참조하세요.

## 2단계: 애플리케이션을 최신 버전으로 업데이트

최신 1. x 버전을 모든 호스트에 배포했으면 2.0.x 이상 버전으로 업그레이드할 수 있습니다. 버전 2.0.x에는 모든 이전 버전의 주요 변경 사항이 포함되어 있습니다 AWS Encryption SDK. 하지만 1단계에서 권장하는 대로 코드를 변경하면 최신 버전으로 마이그레이션할 때 오류를 피할 수 있습니다.

최신 버전으로 업데이트하기 전에 커밋 정책이 일관되게 `ForbidEncryptAllowDecrypt`로 설정되어 있는지 확인하세요. 그런 다음 데이터 구성에 따라 편할 때에 `RequireEncryptAllowDecrypt`로 마이그레이션한 다음 기본 설정인 `RequireEncryptRequireDecrypt`로 할 수 있습니다. 다음 패턴과 같이 일련의 전환 단계를 수행하는 것이 좋습니다.

1. 커밋 정책을 `ForbidEncryptAllowDecrypt`로 설정한 상태에서 시작하세요. AWS Encryption SDK 는 키 커밋으로 메시지를 복호화할 수 있지만 아직 키 커밋을 사용해 암호화하지는 않습니다.
2. 준비가 완료되면 약정 정책을 `RequireEncryptAllowDecrypt`로 업데이트하세요. 는 키 커밋으로 데이터를 암호화하기 AWS Encryption SDK 시작합니다. 키 커밋 사용 여부와 관계없이 사이퍼텍스트를 복호화할 수 있습니다.

   커밋 정책을 `RequireEncryptAllowDecrypt`로 업데이트하기 전에 최신 1.x 버전이 생성한 사이퍼텍스트를 복호화하는 애플리케이션의 호스트를 포함하여 모든 호스트에 배포되어 있는지 확인합니다. 버전 1.7.x AWS Encryption SDK 이전의 버전은 키 커밋으로 암호화된 메시지를 해독할 수 없습니다.

   또한 아직 키 커밋 없이 사이퍼텍스트를 처리하고 있는지 여부를 측정하는 지표를 애플리케이션에 추가하기에 좋은 타이밍입니다. 이렇게 하면 커밋 정책 설정을 언제 `RequireEncryptRequireDecrypt`로 업데이트해도 안전한지 판단할 수 있습니다. Amazon SQS 대기열의 메시지를 암호화하는 것과 같은 일부 애플리케이션의 경우 이하 버전에서 암호화된 모든 사이퍼텍스트가 다시 암호화되거나 삭제될 때까지 오래 기다려야 할 수 있습니다. 암호화된 S3 객

체와 같은 다른 애플리케이션의 경우 모든 객체를 다운로드하고, 재암호화, 재업로드해야 할 수 있습니다.

3. 키 커밋 없이 암호화된 메시지가 없는 것이 확실하면 약정 정책을 `RequireEncryptRequireDecrypt`로 업데이트할 수 있습니다. 이 값을 사용하면 항상 키 커밋을 통해 데이터가 암호화되고 복호화됩니다. 이 설정은 기본값이므로 명시적으로 설정할 필요는 없지만 이 설정을 사용하는 것이 좋습니다. 명시적 설정은 애플리케이션에서 키 커밋 없이 암호화된 사이퍼텍스트를 발견할 경우 필요할 수 있는 디버깅 및 잠재적 롤백에 도움이 됩니다.

# AWS KMS 마스터 키 공급자 업데이트

최신 1.x 버전의 로 마이그레이션 AWS Encryption SDK한 다음 버전 2.0.x 이상으로 마이그레이션하려면 레거시 AWS KMS 마스터 키 공급자를 엄격 모드 또는 검색 모드에서 명시적으로 생성된 마스터 키 공급자로 바꿔야 합니다. 레거시 마스터 키 공급자는 버전 1.7x에서 더 이상 사용되지 않으며 버전 2.0.x에서 제거되었습니다. 이 변경은 AWS Encryption SDK for Java, AWS Encryption SDK for Python 및 AWS Encryption CLI를 사용하는 애플리케이션 및 스크립트에 반드시 필요합니다. 이 섹션의 예제는 코드를 업데이트하는 방법을 보여줍니다.

> **Note**
>
> Python의 경우 지원 중단 경고를 켭니다. 이렇게 하면 코드에서 업데이트해야 하는 부분을 식별하는 데 도움이 됩니다.

AWS KMS 마스터 키(마스터 키 공급자 아님)를 사용하는 경우이 단계를 건너뛸 수 있습니다. AWS KMS 마스터 키는 더 이상 사용되지 않거나 제거되지 않습니다. 해당 마스터 키는 지정한 래핑 키로만 암호화하고 복호화합니다.

이 섹션의 예제는 변경해야 하는 코드 요소에 초점을 맞춥니다. 업데이트된 코드의 전체 예제는 해당 프로그래밍 언어의 GitHub 리포지토리의 예제 섹션을 참조하세요. 또한 이러한 예제에서는 일반적으로 키 ARNs 사용하여를 나타냅니다 AWS KMS keys. 암호화를 위한 마스터 키 공급자를 생성할 때 유효한 AWS KMS 키 식별자를 사용하여를 나타낼 수 있습니다 AWS KMS key . 복호화를 위한 마스터 키 공급자를 생성할 경우 키 ARN을 사용해야 합니다.

마이그레이션에 대해 자세히 알아보기

모든 AWS Encryption SDK 사용자에 대해에서 커밋 정책을 설정하는 방법에 대해 알아봅니다the section called “커밋 정책 설정”.

AWS Encryption SDK for C 및 AWS Encryption SDK for JavaScript 사용자의 경우의 키링에 대한 선택적 업데이트에 대해 알아봅니다AWS KMS 키링 업데이트.

주제



## 엄격 모드로 마이그레이션

의 최신 1.x 버전으로 업데이트한 후 레거시 마스터 키 공급자를 엄격 모드의 마스터 키 공급자로 AWS Encryption SDK바꿉니다. 엄격 모드에서는 암호화 및 복호화 시 사용할 래핑 키를 지정해야 합니다. 는 지정한 래핑 키만 AWS Encryption SDK 사용합니다. 더 이상 사용되지 않는 마스터 키 공급자는 및 AWS 계정 리전 AWS KMS keys 을 포함하여 데이터 키를 암호화 AWS KMS key 한를 사용하여 데이터를 해독할 수 있습니다.

엄격한 모드의 마스터 키 공급자는 AWS Encryption SDK 버전 1.7.x에 도입되었습니다. 이는 버전 1.7x에서 더 이상 사용되지 않으며 버전 2.0.x에서 제거되는 레거시 마스터 키 공급자를 교체합니다. 엄격 모드에서 마스터 키 공급자를 사용하는 것이 AWS Encryption SDK 모범 사례입니다.

다음 코드는 암호화 및 복호화에 사용할 수 있는 엄격 모드의 마스터 키 공급자를 생성합니다.

Java

이 예제는 AWS Encryption SDK for Java의 버전 1.6.2 이하 버전을 사용하는 애플리케이션의 코드를 나타냅니다.

이 코드는 `KmsMasterKeyProvider.builder()` 메서드를 사용하여 마스터 키 공급자를 래핑 키 AWS KMS key 로 사용하는 AWS KMS 마스터 키 공급자를 인스턴스화합니다.

```
// 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();
```

이 예제는 AWS Encryption SDK for Java 의 버전 1.7.x 이상 버전을 사용하는 애플리케이션의 코드를 나타냅니다. 전체 예제는 BasicEncryptionExample.java를 참조하세요.

이전 예제에서 사용된 `Builder.build()` 및 `Builder.withKeysForEncryption()` 메서드는 버전 1.7.x에서 더 이상 사용되지 않으며 버전 2.0.x에서 제거되었습니다.

엄격 모드 마스터 키 공급자로 업데이트하기 위해 이 코드는 더 이상 사용되지 않는 메서드에 대한 호출을 새 `Builder.buildStrict()` 메서드에 대한 호출로 대체합니다. 이 예제에서는 래핑 키 AWS KMS key 로 하나를 지정하지만 `Builder.buildStrict()` 메서드는 여러의 목록을 가져올 수 있습니다 AWS KMS keys.

```
// Create a master key provider in strict mode
// Replace the example key ARN with a valid one from your AWS ##.
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);
```

Python

이 예제는 AWS Encryption SDK for Python의 버전 1.4.1을 사용하는 애플리케이션의 코드를 나타냅니다. 이 코드는 버전 1.7.x에서 더 이상 사용되지 않으며 버전 2.0.x에서 제거된 `KMSMasterKeyProvider`를 사용합니다. 복호화 시 AWS KMS keys 지정한에 관계없이 데이터 키를 암호화 AWS KMS key 한를 사용합니다.

단, `KMSMasterKey`는 더 이상 사용되지 않거나 제거되지 않았습니다. 암호화 및 복호화 시 AWS KMS key 지정한 만 사용합니다.

```
# 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]
)
```

이 예제는 AWS Encryption SDK for Python의 버전 1.7.x를 사용하는 애플리케이션의 코드를 나타냅니다. 전체 예제는 basic_encryption.py를 참조하세요.

엄격 모드 마스터 키 공급자로 업데이트하기 위해 이 코드는 `KMSMasterKeyProvider()`에 대한 호출을 `StrictAwsKmsMasterKeyProvider()`에 대한 호출로 대체합니다.

```
# Create a master key provider in strict mode
# Replace the example key ARNs with valid values from your AWS ##
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

이 예제에서는 Encryption CLI 버전 1.1.7 이하를 사용하여 AWS 암호화 및 복호화하는 방법을 보여줍니다.

1.1.7 이하 버전에서는 암호화할 때 하나 이상의 마스터 키(또는 래핑 키)를 지정합니다(예: AWS KMS key). 사용자 지정 마스터 키 공급자를 사용하지 않는 한 복호화 시 래핑 키를 지정할 수 없습니다. AWS 암호화 CLI는 데이터 키를 암호화한 모든 래핑 키를 사용할 수 있습니다.

```
\\ Replace the example key ARN with a valid one
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --master-keys key=$keyArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

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

이 예제에서는 AWS Encryption CLI 버전 1.7.x 이상을 사용하여 암호화 및 복호화하는 방법을 보여줍니다. 전체 예제는 AWS 암호화 CLI의 예 섹션을 참조하세요.

`--master-keys` 파라미터는 버전 1.7.x에서 더 이상 사용되지 않으며 버전 2.0.x에서 제거되었습니다. 암호화 및 복호화 명령에 필요한 `--wrapping-keys` 파라미터로 대체되었습니다. 이 파라미터는 엄격 모드 및 검색 모드를 지원합니다. 엄격한 모드는 의도한 래핑 키를 사용하도록 보장하는 AWS Encryption SDK 모범 사례입니다.

엄격 모드로 업그레이드하려면 `--wrapping-keys` 파라미터의 key 속성을 사용하여 암호화 및 복호화 시 래핑 키를 지정하세요.

```
\\ Replace the example key ARN with a valid value
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

## 검색 모드로 마이그레이션

버전 1.7.x부터 마스터 키 공급자에 대해 AWS KMS 엄격 모드를 사용하는 것이 AWS Encryption SDK 가장 좋습니다. 즉, 암호화 및 복호화 시 래핑 키를 지정하는 것입니다. 암호화할 때는 항상 래핑 키를 지정해야 합니다. 그러나 복호화를 AWS KMS keys 위해의 키 ARNs을 지정하는 것이 실용적이지 않은 경우가 있습니다. 예를 들어 별칭을 사용하여 암호화 AWS KMS keys 할 때를 식별하는 경우 복호화할 때 키 ARNs을 나열해야 하는 경우 별칭의 이점을 잃게 됩니다. 또한 검색 모드의 마스터 키 공급자는 원래 마스터 키 공급자처럼 작동하므로 마이그레이션 전략의 일환으로 일시적으로 사용하고 나중에 엄격 모드에서 마스터 키 공급자로 업그레이드할 수 있습니다.

이와 같은 경우에는 검색 모드에서 마스터 키 공급자를 사용할 수 있습니다. 이러한 마스터 키 공급자에서는 래핑 키를 지정할 수 없으므로 암호화에 사용할 수 없습니다. 복호화할 때는 데이터 키를 암호화한 모든 래핑 키를 사용할 수 있습니다. 하지만 동일한 방식으로 동작하는 레거시 마스터 키 공급자와는 달리 검색 모드에서 명시적으로 생성해야 합니다. 검색 모드에서 마스터 키 공급자를 사용하는 경우 사용할 수 있는 래핑 키를 특정 AWS 계정의 것으로만 제한할 수 있습니다. 이 검색 필터는 선택 사항이지만 권장되는 모범 사례입니다. AWS 파티션과 계정에 대한 자세한 내용은 AWS 일반 참조의 Amazon 리소스 이름을 참조하세요.

다음 예제에서는 암호화를 위한 엄격 모드의 AWS KMS 마스터 키 공급자와 복호화를 위한 검색 모드의 AWS KMS 마스터 키 공급자를 생성합니다. 검색 모드의 마스터 키 공급자는 검색 필터를 사용하여 복호화에 사용되는 래핑 키를 aws 파티션 및 특정 예제 AWS 계정으로만 제한합니다. 매우 간단한 이 예제에서는 계정 필터가 필요하지 않지만 한 애플리케이션에서 데이터를 암호화하고 다른 애플리케이션에서 데이터를 복호화할 때 매우 유용한 모범 사례입니다.

Java

이 예제는 AWS Encryption SDK for Java의 버전 1.7.x 이상 버전을 사용하는 애플리케이션의 코드를 나타냅니다. 전체 예제는 DiscoveryDecryptionExample.java를 참조하세요.

암호화를 위한 엄격 모드에서 마스터 키 공급자를 인스턴스화하기 위해 이 예제에서는 `Builder.buildStrict()` 메서드를 사용합니다. 복호화를 위한 검색 모드에서 마스터 키 공급자를 인스턴스화하기 위해서는 `Builder.buildDiscovery()` 메서드를 사용합니다. `Builder.buildDiscovery()` 메서드는 지정된 AWS 파티션 및 계정 AWS KMS keys 에서를 AWS Encryption SDK 로 제한`DiscoveryFilter`하는를 사용합니다.

```
// Create a master key provider in strict mode for encrypting
// Replace the example alias ARN with a valid one from your AWS ##.
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias";

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

이 예제는 AWS Encryption SDK for Python 의 버전 1.7.x 이상 버전을 사용하는 애플리케이션의 코드를 나타냅니다. 전체 예제는 discovery_kms_provider.py를 참조하세요.

암호화를 위한 엄격 모드에서 마스터 키 공급자를 생성하기 위해 이 예제에서는 `StrictAwsKmsMasterKeyProvider`를 사용합니다. 복호화를 위한 검색 모드에서 마스터 키 공급자를 생성하기 위해 지정된 AWS 파티션 및 계정 AWS KMS keys 에서를 AWS Encryption SDK 로 제한`DiscoveryFilter`하는 `DiscoveryAwsKmsMasterKeyProvider`와 함께를 사용합니다.

```
# Create a master key provider in strict mode
# Replace the example key ARN and alias ARNs with valid values from your AWS ##.
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

이 예제에서는 AWS Encryption CLI 버전 1.7.x 이상을 사용하여 암호화 및 복호화하는 방법을 보여줍니다. 버전 1.7.x부터 암호화 및 복호화 시 `--wrapping-keys` 파라미터가 필요합니다. `--wrapping-keys` 파라미터는 엄격 모드 및 검색 모드를 지원합니다. 전체 예제는 the section called “예제” 섹션을 참조하세요.

암호화할 때 이 예제에서는 래핑 키를 지정합니다(필수 사항). 복호화할 때는 값이 `true`인 `--wrapping-keys` 파라미터의 `discovery` 속성을 사용하여 검색 모드를 명시적으로 선택합니다.

이 검색 모드에서 사용할 AWS Encryption SDK 수 있는 래핑 키를 특히 래핑 키로 제한하기 위해 AWS 계정이 예제에서는 `--wrapping-keys` 파라미터의 `discovery-partition` 및 `discovery-account` 속성을 사용합니다. 이러한 선택적 속성은 `discovery` 속성이 `true`로 설정된 경우에만 유효합니다. `discovery-partition` 및 `discovery-account` 속성을 함께 사용해야 하며 둘 다 단독으로는 유효하지 않습니다.

```
\\ Replace the example key ARN with a valid value
$ keyAlias=arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias

\\ 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 \
                                     discovery-account=111122223333 \
                                     discovery-account=444455556666 \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

# AWS KMS 키링 업데이트

의 AWS KMS 키링AWS Encryption SDK for C, AWS Encryption SDK for .NET 및 AWS Encryption SDK for JavaScript는 암호화 및 복호화 시 래핑 키를 지정할 수 있는 지원 모범 사례를 제공합니다. AWS KMS 검색 키링을 생성하는 경우 명시적으로 생성하세요.

> **Note**
>
> AWS Encryption SDK for .NET의 최신 버전은 버전 3.0.x입니다. AWS Encryption SDK for .NET의 모든 버전은의 2.0.x에 도입된 보안 모범 사례를 지원합니다 AWS Encryption SDK. 코드나 데이터를 변경하지 않고도 최신 버전으로 안전하게 업그레이드할 수 있습니다.

의 최신 1.x 버전으로 업데이트할 때 검색 필터를 사용하여 복호화할 때 AWS KMS 검색 키링 또는 AWS KMS 리전 검색 키링이 사용하는 래핑 키를 특정 래핑 키로 제한할 AWS Encryption SDK수 있습니다 AWS 계정. 검색 키링을 필터링하는 것이 AWS Encryption SDK 모범 사례입니다.

이 섹션의 예제는 AWS KMS 리전별 검색 키링에 검색 필터를 추가하는 방법을 보여줍니다.

마이그레이션에 대해 자세히 알아보기

모든 AWS Encryption SDK 사용자에 대해에서 커밋 정책을 설정하는 방법에 대해 알아봅니다the section called “커밋 정책 설정”.

AWS Encryption SDK for Java AWS Encryption SDK for Python및 AWS 암호화 CLI 사용자의 경우에서 마스터 키 공급자에 필요한 업데이트에 대해 알아봅니다the section called “AWS KMS 마스터 키 공급자 업데이트”.

애플리케이션에 다음과 같은 코드가 있을 수 있습니다. 이 예제에서는 미국 서부(오레곤)(us-west-2) 리전의 래핑 키만 사용할 수 있는 AWS KMS 리전 검색 키링을 생성합니다. 이 예제는 1.7.x 이전 AWS Encryption SDK 버전의 코드를 나타냅니다. 하지만 1.7.x 이상 버전에서도 여전히 유효합니다.

C

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

JavaScript Browser

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

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

JavaScript Node.js

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

버전 1.7.x부터 모든 검색 키링에 AWS KMS 검색 필터를 추가할 수 있습니다. 이 검색 필터는가 복호화에 사용할 AWS Encryption SDK 수 AWS KMS keys 있는를 지정된 파티션 및 계정의 로 제한합니다. 이 코드를 사용하기 전에 필요한 경우 파티션을 변경하고 예제 계정 ID를 유효한 것으로 바꾸세요.

C

전체 예제는 kms_discovery.cpp를 참조하세요.

```
std::shared_ptr<KmsKeyring::DiscoveryFilter> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
        .AddAccount("111122223333")
        .AddAccount("444455556666")
        .Build());

struct aws_cryptosdk_keyring *kms_regional_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()

 .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery(discovery_filter))
```

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', '444455556666'], partition:
 'aws' }
})
```

JavaScript Node.js

전체 예제는 kms_filtered_discovery.ts를 참조하세요.

```
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

# 커밋 정책 설정

키 커밋을 사용하면 암호화된 데이터가 항상 동일한 일반 텍스트로 복호화됩니다. 버전 1.7.x부터이 보안 속성을 제공하기 위해는 키 커밋과 함께 새 알고리즘 제품군을 AWS Encryption SDK 사용합니다. 데이터를 암호화하고 복호화할 때 키 커밋 사용 여부를 결정하려면 커밋 정책 구성 설정을 사용합니다. 키 커밋으로 데이터를 암호화하고 복호화하는 것이 AWS Encryption SDK 모범 사례입니다.

커밋 정책 설정은 최신 1.x 버전에서 버전 AWS Encryption SDK 2.0.x 이상으로 마이그레이션하는 마이그레이션 프로세스의 두 번째 단계에서 중요한 부분입니다. 커밋 정책을 설정하고 변경한 후에는 애플리케이션을 프로덕션 환경에 배포하기 전에 철저하게 테스트해야 합니다. 마이그레이션 지침은 마이그레이션 및 배포 방법 AWS Encryption SDK 섹션을 참조하세요.

2.0.x 이상 버전에서 커밋 정책 설정에는 다음과 같은 세 가지 유효한 값이 있습니다. 최신 1.x 버전(1.7x 버전부터)에서는 `ForbidEncryptAllowDecrypt`만 유효합니다.

- `ForbidEncryptAllowDecrypt` -는 키 커밋으로 암호화할 AWS Encryption SDK 수 없습니다. 암호화된 사이퍼텍스트를 키 커밋 사용 여부와 관계없이 복호화할 수 있습니다.

  최신 1.x 버전에서 이는 유일하게 유효한 값입니다. 이를 통해 키 커밋으로 복호화할 준비가 완전히 완료되기 전까지는 키 커밋으로 암호화하지 않도록 합니다. 이 값을 명시적으로 설정하면 2.0.x 이상 버전으로 업그레이드할 때 커밋 정책이 자동으로 `require-encrypt-require-decrypt`로 변경되는 것을 방지할 수 있습니다. 대신, 단계적으로 커밋 정책을 마이그레이션할 수 있습니다.
- `RequireEncryptAllowDecrypt` -는 AWS Encryption SDK 항상 키 커밋으로 암호화합니다. 암호화된 사이퍼텍스트를 키 커밋 사용 여부와 관계없이 복호화할 수 있습니다. 이 값은 버전 2.0.x에 추가되었습니다.
- `RequireEncryptRequireDecrypt` -는 AWS Encryption SDK 항상 키 커밋으로 암호화 및 복호화합니다. 이 값은 버전 2.0.x에 추가되었습니다. 이는 버전 2.0.x 이상에서 기본값입니다.

최신 1.x 버전에서 유일하게 유효한 커밋 정책 값은 `ForbidEncryptAllowDecrypt`입니다. 2.0.x 이상 버전으로 마이그레이션한 후 준비가 되는 대로 커밋 정책을 단계적으로 변경할 수 있습니다. 키 커밋 없이 암호화된 메시지가 없는 것이 확인되기 전에는 커밋 정책을 `RequireEncryptRequireDecrypt`로 업데이트하지 마세요.

다음 예제는 최신 1.x 버전 및 2.0.x 이상 버전에서 커밋 정책을 설정하는 방법을 보여줍니다. 기술은 프로그래밍 언어에 따라 달라집니다.

마이그레이션에 대해 자세히 알아보기

AWS Encryption SDK for Java AWS Encryption SDK for Python및 AWS 암호화 CLI의 경우에서 마스터 키 공급자에 필요한 변경 사항에 대해 알아봅니다the section called “AWS KMS 마스터 키 공급자 업데이트”.

AWS Encryption SDK for C 및의 경우의 키링에 대한 선택적 업데이트에 대해 AWS Encryption SDK for JavaScript알아봅니다AWS KMS 키링 업데이트.

## 커밋 정책 설정 방법

커밋 정책을 설정하는 데 사용하는 방법은 각 언어 구현마다 조금씩 다릅니다. 이 예제에서는 해당 작업 방법을 보여줍니다. 커밋 정책을 변경하기 전에 마이그레이션 및 배포 방법에서 다단계 접근 방식을 검토하세요.

C

버전 1.7.x부터 `aws_cryptosdk_session_set_commitment_policy` 함수를 AWS Encryption SDK for C사용하여 암호화 및 복호화 세션에 커밋 정책을 설정합니다. 설정한 커밋 정책은 해당 세션에서 호출된 모든 암호화 및 복호화 작업에 적용됩니다.

`aws_cryptosdk_session_new_from_keyring` 및 `aws_cryptosdk_session_new_from_cmm` 함수는 버전 1.7.x에서 더 이상 사용되지 않으며 버전 2.0.x에서 제거되었습니다. 이러한 함수는 세션을 반환하는 `aws_cryptosdk_session_new_from_keyring_2` 및 `aws_cryptosdk_session_new_from_cmm_2` 함수로 대체됩니다.

최신 1.x 버전에서 `aws_cryptosdk_session_new_from_keyring_2` 및 `aws_cryptosdk_session_new_from_cmm_2`를 사용한 경우 `COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT` 커밋 정책 값을 사용하여 `aws_cryptosdk_session_set_commitment_policy` 함수를 호출해야 합니다. 2.0.x 이상 버전의 경우 이 함수를 호출하는 것은 선택 사항이며 유효한 값을 모두 사용합니다. 2.0.x 이상 버전의 기본 커밋 정책은 `COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT`입니다.

전체 예를 보려면 string.cpp를 참조하세요.

```
/* 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(
    alloc, AWS_CRYPTOSDK_ENCRYPT, kms_keyring);

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

`require-encrypt-require-decrypt` 값은 AWS Encryption SDK for .NET의 모든 버전에서 기본 커밋 정책입니다. 모범 사례로 명시적으로 설정할 수 있지만 필수 사항은 아닙니다. 그러나 AWS Encryption SDK for .NET을 사용하여 키 커밋 AWS Encryption SDK 없이의 다른 언어 구현으로 암호화된 사이퍼텍스트를 해독하는 경우 커밋 정책 값을 `REQUIRE_ENCRYPT_ALLOW_DECRYPT` 또는 로 변경해야 합니다`FORBID_ENCRYPT_ALLOW_DECRYPT`. 그러지 않으면 사이퍼텍스트 복호화 시도가 실패합니다.

AWS Encryption SDK for .NET에서의 인스턴스에 커밋 정책을 설정합니다 AWS Encryption SDK. `CommitmentPolicy` 파라미터를 사용하여 `AwsEncryptionSdkConfig` 객체를 인스턴스화하고 구성 객체를 사용하여 AWS Encryption SDK 인스턴스를 생성합니다. 그런 다음 구성된 AWS Encryption SDK 인스턴스의 `Encrypt()` 및 `Decrypt()` 메서드를 호출합니다.

이 예에서는 커밋 정책을 `require-encrypt-allow-decrypt`로 설정합니다.

```
// 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"}encryptionSdk
};

var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var keyring = materialProviders.CreateAwsKmsKeyring(createKeyringInput);

// Encrypt your plaintext data
var encryptInput = new EncryptInput
{
    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

AWS 암호화 CLI에서 커밋 정책을 설정하려면 `--commitment-policy` 파라미터를 사용합니다. 이 파라미터는 버전 1.8.x에 도입되었습니다.

최신 1.x 버전의 경우 `--encrypt` 또는 `--decrypt` 명령에서 `--wrapping-keys` 파라미터를 사용할 때는 `forbid-encrypt-allow-decrypt` 값이 있는 `--commitment-policy` 파라미터가 필요합니다. 그러지 않으면 `--commitment-policy` 파라미터가 유효하지 않게 됩니다.

2.1.x 및 이상 버전에서는 `--commitment-policy` 파라미터가 선택 사항이며 키 커밋 없이 암호화된 사이퍼텍스트를 암호화하거나 복호화하지 않는 `require-encrypt-require-decrypt` 값이 기본값입니다. 하지만 유지 관리 및 문제 해결에 도움이 되도록 모든 암호화 및 복호화 호출에서 커밋 정책을 명시적으로 설정하는 것이 좋습니다.

이 예에서는 커밋 정책을 설정합니다. 또한 1.8.x 버전부터 `--master-keys` 파라미터를 대체하는 `--wrapping-keys` 파라미터를 사용합니다. 자세한 내용은 the section called “AWS KMS 마스터 키 공급자 업데이트”을 참조하세요. 전체 예제는 AWS 암호화 CLI의 예 섹션을 참조하세요.

```
\\ 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 \
                     --wrapping-keys key=$keyArn \
                     --commitment-policy forbid-encrypt-allow-decrypt \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext - supports key commitment on 1.7 and later
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --commitment-policy forbid-encrypt-allow-decrypt \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

Java

버전 1.7.x부터 AWS Encryption SDK 클라이언트를 나타내는 객체`AwsCrypto`인의 인스턴스에 커밋 정책을 AWS Encryption SDK for Java설정합니다. 이 커밋 정책 설정은 해당 클라이언트에서 호출된 모든 암호화 및 복호화 작업에 적용됩니다.

`AwsCrypto()` 생성자는의 최신 1.x 버전에서는 더 이상 사용되지 AWS Encryption SDK for Java 않으며 버전 2.0.x에서는 제거됩니다. 새 `Builder` 클래스, `Builder.withCommitmentPolicy()` 메서드, `CommitmentPolicy` 열거 유형으로 대체됩니다.

최신 1.x 버전의 `Builder` 클래스에는 `Builder.withCommitmentPolicy()` 메서드와 `CommitmentPolicy.ForbidEncryptAllowDecrypt` 인수가 필요합니다. 2.0.x 버전부터 `Builder.withCommitmentPolicy()` 메서드는 선택 사항이고 기본값은 `CommitmentPolicy.RequireEncryptRequireDecrypt`입니다.

전체 예제는 SetCommitmentPolicyExample.java를 참조하세요.

```
// 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

버전 1.7.x부터 AWS Encryption SDK 클라이언트를 인스턴스화하는 새 `buildClient` 함수를 호출할 때 커밋 정책을 설정할 AWS Encryption SDK for JavaScript수 있습니다. `buildClient` 함수는 커밋 정책을 나타내는 열거형 값을 사용합니다. 암호화 및 복호화 시 커밋 정책을 적용하는 업데이트된 `encrypt` 및 `decrypt` 함수를 반환합니다.

최신 1.x 버전에서는 `buildClient` 함수에 `CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT` 인수가 필요합니다. 2.0.x 버전부터 커밋 정책 인수는 선택 사항이고 기본값은 `CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT`입니다.

브라우저에 자격 증명을 설정하려면 명령문이 필요하다는 점을 제외하면 Node.js 코드와 브라우저의 코드는 동일합니다.

다음 예제에서는 AWS KMS 키링을 사용하여 데이터를 암호화합니다. 새 `buildClient` 함수는 커밋 정책을 `FORBID_ENCRYPT_ALLOW_DECRYPT`로 설정하며 이는 최신 1.x 버전의 기본값입니다.

buildClient에서 반환되는 업그레이드된 encrypt 및 decrypt 함수는 사용자가 설정한 커밋 정책을 적용합니다.

```
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

버전 1.7.x부터 AWS Encryption SDK 클라이언트를 나타내는 새 객체EncryptionSDKClient인의 인스턴스에 커밋 정책을 AWS Encryption SDK for Python설정합니다. 설정한 커밋 정책은 해당 클라이언트 인스턴스를 사용하는 모든 encrypt 및 decrypt 호출에 적용됩니다.

최신 1.x 버전의 EncryptionSDKClient 생성자에는 CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT 열거형 값이 필요합니다. 2.0.x 버전부터 커밋 정책 인수는 선택 사항이고 기본값은 CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT입니다.

이 예제에서는 새 EncryptionSDKClient 생성자를 사용하고 커밋 정책을 1.7.x 기본값으로 설정합니다. 생성자는 AWS Encryption SDK를 나타내는 클라이언트를 인스턴스화합니다. 이 클라이언트에서 encrypt, decrypt 또는 stream 메서드를 호출하면 설정한 커밋 정책이 적용됩니다. 또한이 예제에서는 암호화 및 복호화 AWS KMS keys 시기를 지정하는 StrictAwsKmsMasterKeyProvider 클래스의 새 생성자를 사용합니다.

전체 예제는 set_commitment.py를 참조하세요.

```
# Instantiate the client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.FORBID_ENCRYPT_AL
```

```
// 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
)
```

Rust

`require-encrypt-require-decrypt` 값은 AWS Encryption SDK for Rust의 모든 버전에서 기본 커밋 정책입니다. 모범 사례로 명시적으로 설정할 수 있지만 필수 사항은 아닙니다. 그러나 AWS Encryption SDK for Rust를 사용하여 키 커밋 AWS Encryption SDK 없이의 다른 언어 구현으로 암호화된 사이퍼텍스트를 해독하는 경우 커밋 정책 값을 `REQUIRE_ENCRYPT_ALLOW_DECRYPT` 또는 로 변경해야 합니다`FORBID_ENCRYPT_ALLOW_DECRYPT`. 그러지 않으면 사이퍼텍스트 복호화 시도가 실패합니다.

AWS Encryption SDK for Rust에서의 인스턴스에 커밋 정책을 설정합니다 AWS Encryption SDK. `comitment_policy` 파라미터를 사용하여 `AwsEncryptionSdkConfig` 객체를 인스턴스화하고 구성 객체를 사용하여 AWS Encryption SDK 인스턴스를 생성합니다. 그런 다음 구성된 AWS Encryption SDK 인스턴스의 `Encrypt()` 및 `Decrypt()` 메서드를 호출합니다.

이 예에서는 커밋 정책을 `forbid-encrypt-allow-decrypt`로 설정합니다.

```
// Configure the commitment policy on the AWS Encryption SDK instance
let esdk_config = AwsEncryptionSdkConfig::builder()
                    .commitment_policy(ForbidEncryptAllowDecrypt)
                    .build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;
```

```
// Create an AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create an AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

// Encrypt your plaintext data
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

// Decrypt your ciphertext
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
    .encryption_context(encryption_context)
```

```
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
commitPolicyForbidEncryptAllowDecrypt :=
 mpltypes.ESDKCommitmentPolicyForbidEncryptAllowDecrypt
encryptionClient, err :=
 client.NewClient(esdktypes.AwsEncryptionSdkConfig{CommitmentPolicy:
 &commitPolicyForbidEncryptAllowDecrypt})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[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 material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS keyring
awsKmsKeyringInput := mpltypes.CreateAwsKmsKeyringInput{
    KmsClient: kmsClient,
    KmsKeyId:  kmsKeyId,
}
awsKmsKeyring, err := matProv.CreateAwsKmsKeyring(context.Background(),
 awsKmsKeyringInput)
if err != nil {
    panic(err)
}

// Encrypt your plaintext data
res, err := forbidEncryptClient.Encrypt(context.Background(),
 esdktypes.EncryptInput{
    Plaintext:         []byte(exampleText),
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}

// Decrypt your ciphertext
decryptOutput, err := forbidEncryptClient.Decrypt(context.Background(),
 esdktypes.DecryptInput{
    Ciphertext:        res.Ciphertext,
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}
```

# 최신 버전으로의 마이그레이션 문제 해결

애플리케이션을 버전 2.0.x 이상으로 업데이트하기 전에의 최신 1.x 버전으로 AWS Encryption SDK업데이트 AWS Encryption SDK 하고 완전히 배포합니다. 이렇게 하면 2.0.x 이상 버전으로 업데이트할 때 발생할 수 있는 대부분의 오류를 방지하는 데 도움이 됩니다. 예를 포함한 자세한 지침은 마이그레이션 AWS Encryption SDK 섹션을 참조하세요.

**Important**

최신 1.x 버전이 AWS Encryption SDK의 1.7.x 또는 이상 버전인지 확인하세요.

**Note**

AWS 암호화 CLI:이 가이드에서 버전 1.7.x에 대한 참조는 AWS 암호화 CLI 버전 1.8.x에 AWS Encryption SDK 적용됩니다. 이 가이드에서 버전 2.0.x에 대한 참조는 AWS Encryption CLI의 2.1.x에 AWS Encryption SDK 적용됩니다.
새로운 보안 기능은 원래 AWS Encryption CLI 버전 1.7.x 및 2.0.x에서 릴리스되었습니다. 그러나 AWS Encryption CLI 버전 1.8.x는 버전 1.7.x를 대체하고 AWS Encryption CLI 2.1.x는 2.0.x를 대체합니다. 자세한 내용은 GitHub의 aws-encryption-sdk-cli 리포지토리에서 관련 보안 권고를 참조하세요.

이 항목은 발생할 수 있는 가장 일반적인 오류를 인식하고 해결하는 데 도움이 되도록 설계되었습니다.

주제

- 더 이상 사용되지 않거나 제거된 객체
- 구성 충돌: 커밋 정책 및 알고리즘 제품군
- 구성 충돌: 커밋 정책 및 사이퍼텍스트
- 키 커밋 검증 실패
- 기타 암호화 오류
- 기타 복호화 오류
- 롤백 고려 사항

## 더 이상 사용되지 않거나 제거된 객체

버전 2.0.x에는 버전 1.7.x에서 더 이상 사용되지 않는 레거시 생성자, 메서드, 함수 및 클래스 제거를 비롯한 몇 가지 주요 변경 사항이 포함되어 있습니다. 컴파일러 오류, 가져오기 오류, 구문 오류 및 기호를 찾을 수 없음 오류(프로그래밍 언어에 따라 다름)를 방지하려면 먼저 프로그래밍 언어에 AWS Encryption SDK 맞는 최신 1.x 버전의 로 업그레이드합니다. (1.7.x 이상 버전이어야 합니다.) 최신 1.x 버전을 사용하는 경우 원본 기호가 제거되기 전에 대체 요소 사용을 시작할 수 있습니다.

2.0.x 이상 버전으로 즉시 업그레이드해야 하는 경우 사용 중인 프로그래밍 언어의 변경 로그를 참조하고 기존 기호를 변경 로그에서 권장하는 기호로 바꾸세요.

## 구성 충돌: 커밋 정책 및 알고리즘 제품군

커밋 정책과 충돌하는 알고리즘 제품군을 지정하는 경우 구성 충돌 오류가 발생하여 암호화 호출이 실패합니다.

이러한 유형의 오류를 방지하려면 알고리즘 제품군을 지정하지 마세요. 기본적으로 AWS Encryption SDK 는 커밋 정책과 호환되는 가장 안전한 알고리즘을 선택합니다. 그러나 서명하지 않은 알고리즘 제품군과 같이 알고리즘 제품군을 지정해야 하는 경우 커밋 정책과 호환되는 알고리즘 제품군을 선택해야 합니다.

| 커밋 정책 | 호환 가능한 알고리즘 제품군 |
| --- | --- |
| ForbidEncryptAllowDecrypt | 다음과 같이 키 커밋이 없는 모든 알고리즘 세트:<br>`AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384` (03 78)(서명 있음)<br><br>`AES_256_GCM_IV12_TAG16_HKDF_SHA256` (01 78)(서명 없음) |
| RequireEncryptAllowDecrypt<br><br>RequireEncryptRequireDecrypt | 다음과 같이 키 커밋이 있는 모든 알고리즘 세트:<br>`AES_256_GCM_HKDF_SHA512_COMMIT_KEY_ECDSA_P384` (05 78)(서명 있음)<br><br>`AES_256_GCM_HKDF_SHA512_COMMIT_KEY` (04 78)(서명 없음) |

알고리즘 세트를 지정하지 않은 상태에서 이 오류가 발생하는 경우, 암호화 자료 관리자(CMM)가 충돌하는 알고리즘 세트를 선택했을 수 있습니다. 기본 CMM은 충돌하는 알고리즘 세트를 선택하지 않지만 사용자 지정 CMM은 충돌하는 알고리즘 세트를 선택할 수 있습니다. 도움이 필요하면 사용자 지정 CMM 문서 섹션을 참조하세요.

## 구성 충돌: 커밋 정책 및 사이퍼텍스트

`RequireEncryptRequireDecrypt` 커밋 정책에서는 AWS Encryption SDK 가 키 커밋 없이 암호화된 메시지를 복호화하는 것을 허용하지 않습니다. 키 커밋 없이 메시지를 복호화 AWS Encryption SDK 하도록에 요청하면 구성 충돌 오류가 반환됩니다.

이 오류를 방지하려면 `RequireEncryptRequireDecrypt` 커밋 정책을 설정하기 전에 키 커밋 없이 암호화된 모든 사이퍼텍스트를 키 커밋으로 복호화하고 다시 암호화하거나 다른 애플리케이션에서 처리해야 합니다. 이 오류가 발생하는 경우 충돌하는 사이퍼텍스트에 대해 오류를 반환하거나 커밋 정책을 일시적으로 `RequireEncryptAllowDecrypt`로 변경할 수 있습니다.

1.7.x 이하 버전에서 최신 1.x(1.7.x 이상 버전) 버전으로 먼저 업그레이드하지 않고 2.0.x 버전으로 업그레이드했기 때문에 이 오류가 발생한 경우 최신 1.x 버전으로 롤백하고 2.0.x 이상 버전으로 업그레이드하기 전에 해당 버전을 모든 호스트에 배포하는 것을 고려하세요. 도움말은 마이그레이션 및 배포 방법 AWS Encryption SDK를 참조하십시오.

## 키 커밋 검증 실패

키 커밋으로 암호화된 메시지를 복호화할 때 키 커밋 검증 실패 오류 메시지가 표시될 수 있습니다. 이는 암호화된 메시지의 데이터 키가 메시지의 고유 데이터 키와 동일하지 않아 복호화 호출이 실패했음을 나타냅니다. 복호화 중에 데이터 키를 검증함으로써 키 커밋은 메시지를 복호화하여 두 개 이상의 일반 텍스트가 생성될 수 있는 메시지를 복호화하지 못하도록 보호합니다.

이 오류는 복호화하려는 암호화된 메시지가 AWS Encryption SDK에서 반환되지 않았음을 나타냅니다. 수동으로 만든 메시지이거나 데이터 손상의 결과일 수 있습니다. 이 오류가 발생하면 애플리케이션에서 메시지를 거부하고 계속하거나 새 메시지 처리를 중지할 수 있습니다.

## 기타 암호화 오류

암호화는 여러 가지 이유로 실패할 수 있습니다. AWS KMS 검색 키링이나 검색 모드의 마스터 키 제공자를 사용하여 메시지를 암호화할 수 없습니다.

암호화에 사용할 권한이 있는 래핑 키가 있는 키링 또는 마스터 키 제공자를 지정해야 합니다. 권한에 대한 도움말은 AWS Key Management Service 개발자 안내서의 키 정책 보기 및에 대한 액세스 결정을 AWS KMS keys참조하세요. AWS KMS key

## 기타 복호화 오류

암호화된 메시지의 복호화 시도가 실패했다는 것은 AWS Encryption SDK 가 메시지의 암호화된 데이터 키를 복호화할 수 없거나 복호화하지 않는다는 뜻입니다.

래핑 키를 지정하는 키링 또는 마스터 키 공급자를 사용한 경우는 지정한 래핑 키만 AWS Encryption SDK 사용합니다. 의도한 래핑 키를 사용하고 있는지, 래핑 키 중 하나 이상에 대한 `kms:Decrypt` 권한이 있는지 확인하세요. 를 폴백 AWS KMS keys으로 사용하는 경우 검색 AWS KMS 키링 또는 검색 모드의 마스터 키 공급자를 사용하여 메시지를 복호화해 볼 수 있습니다. 작업이 성공하면 일반 텍스트를 반환하기 전에 메시지 복호화에 사용된 키가 신뢰할 수 있는 키인지 확인하세요.

## 롤백 고려 사항

애플리케이션이 데이터를 암호화하거나 복호화하는 데 실패하는 경우 일반적으로 코드 기호, 키링, 마스터 키 제공자 또는 커밋 정책을 업데이트하여 문제를 해결할 수 있습니다. 하지만 애플리케이션을 AWS Encryption SDK의 이하 버전으로 롤백하는 것이 최선이라고 판단하는 경우도 있습니다.

롤백해야 하는 경우에는 조심해서 수행하세요. 1.7.x AWS Encryption SDK 이전의 버전은 키 커밋으로 암호화된 사이퍼텍스트를 해독할 수 없습니다.

- 최신 1.x 버전에서 AWS Encryption SDK 의 이하 버전으로 롤백하는 것이 보통 안전합니다. 이하 버전에서 지원되지 않는 기호와 객체를 사용하려면 코드를 변경한 내용을 취소해야 할 수 있습니다.
- 2.0.x 이상 버전에서 키 커밋(커밋 정책을 `RequireEncryptAllowDecrypt`로 설정)을 사용해 암호화를 시작했다면 1.7.x 버전으로 롤백할 수 있지만 그보다 이하 버전으로는 롤백할 수 없습니다. 1.7.x AWS Encryption SDK 이전의 버전은 키 커밋으로 암호화된 사이퍼텍스트를 해독할 수 없습니다.

모든 호스트가 키 커밋으로 복호화하기 전에 실수로 키 커밋을 사용한 암호화를 활성화한 경우 롤백하는 대신 롤아웃을 계속하는 것이 최선일 수 있습니다. 메시지가 일시적이거나 안전하게 삭제할 수 있는 경우에는 메시지 손실과 함께 롤백을 고려해 볼 수 있습니다. 롤백이 필요한 경우 모든 메시지를 복호화하고 다시 암호화하는 도구를 작성하는 것을 고려할 수 있습니다.

# 자주 묻는 질문(FAQ)

자주 묻는 질문(FAQ)



## 는 AWS SDKs 어떻게 AWS Encryption SDK 다릅니까?

AWS SDKs는 AWS Key Management Service ()를 포함하여 Amazon Web Services(AWS)와 상호 작용하기 위한 라이브러리를 제공합니다AWS KMS. AWS Encryption SDK for .NET AWS Encryption SDK과 같은의 일부 언어 구현에는 항상 동일한 프로그래밍 언어의 AWS SDK가 필요합니다. 다른 언어 구현에는 키링 또는 마스터 키 공급자에서 키를 사용하는 AWS KMS 경우에만 해당 AWS SDK가 필요합니다. 자세한 정보는 AWS Encryption SDK 프로그래밍 언어에서 프로그래밍 언어에 대한 주제를 참조하세요.

AWS SDKs를 사용하여 소량의 데이터(대칭 암호화 키로 최대 4,096바이트) 암호화 및 복호화, 클라이언트 측 암호화를 위한 데이터 키 생성 AWS KMS등와 상호 작용할 수 있습니다. 그러나 데이터 키를 생성할 때는 전체 암호화 및 복호화 프로세스를 관리해야 합니다. 여기에는 외부의 데이터 키를 사용한 데이터 암호화 AWS KMS, 일반 텍스트 데이터 키의 안전한 폐기, 암호화된 데이터 키 저장, 데이터 키 복호화 및 데이터 복호화가 포함됩니다. AWS Encryption SDK 에서 이 프로세스를 처리해 줍니다.

는 업계 표준 및 모범 사례를 사용하여 데이터를 암호화하고 해독하는 라이브러리를 AWS Encryption SDK 제공합니다. 데이터 키를 생성하고 지정한 래핑 키로 암호화한 다음 암호화된 데이터와 복호화에

필요한 암호화된 데이터 키가 포함된 이동 가능 데이터 객체인 암호화된 메시지를 반환합니다. 복호화할 때가 되면 암호화된 메시지와 하나 이상의 래핑 키(선택 사항)를 전달하면가 일반 텍스트 데이터를 AWS Encryption SDK 반환합니다.

에서를 래핑 키 AWS KMS keys 로 사용할 수 AWS Encryption SDK있지만 필수는 아닙니다. 직접 생성한 암호화 키와 키 관리자 또는 온프레미스 하드웨어 보안 모듈에서 생성한 암호화 키를 사용할 수 있습니다. AWS 계정이 없 AWS Encryption SDK 더라도를 사용할 수 있습니다.

## 는 Amazon S3 암호화 클라이언트와 어떻게 AWS Encryption SDK 다릅니까?

AWS SDKs의 Amazon S3 암호화 클라이언트는 Amazon Simple Storage Service(Amazon S3)에 저장하는 데이터에 대한 암호화 및 복호화를 제공합니다. 이러한 클라이언트는 Amazon S3와 긴밀하게 연결되어 있으며 Amazon S3에 저장된 데이터에만 사용할 수 있습니다.

는 어디서나 저장할 수 있는 데이터에 대한 암호화 및 복호화를 AWS Encryption SDK 제공합니다. AWS Encryption SDK 및 Amazon S3 암호화 클라이언트는 데이터 형식이 다른 사이퍼텍스트를 생성하기 때문에 호환되지 않습니다.

## 에서 지원하는 암호화 알고리즘 AWS Encryption SDK과 기본 암호화 알고리즘은 무엇입니까?

는 AES-GCM이라고 하는 Galois/Counter Mode(GCM)의 고급 암호화 표준(AES) 대칭 알고리즘을 AWS Encryption SDK 사용하여 데이터를 암호화합니다. 이를 통해 여러 대칭 및 비대칭 알고리즘 중에서 선택하여, 데이터를 암호화하는 데이터 키를 암호화할 수 있습니다.

AES-GCM의 경우 기본 알고리즘 제품군은 256비트 키, 키 파생(HKDF), 디지털 서명 및 키 커밋이 있는 AES-GCM입니다. AWS Encryption SDK 또한는 디지털 서명 및 키 커밋 없이 192비트 및 128비트 암호화 키와 암호화 알고리즘을 지원합니다.

모든 경우에 초기화 벡터(IV)의 길이는 12바이트이고 인증 태그의 길이는 16바이트입니다. 기본적으로 SDK는 데이터 키를 HMAC 기반 extract-and-expand 키 유도 함수(HKDF)의 입력으로 사용하여 AES-GCM 암호화 키를 유도하고 Elliptic Curve Digital Signature Algorithm(ECDSA) 서명도 추가합니다.

사용할 알고리즘 선택에 대한 자세한 내용은 지원 알고리즘 제품군 섹션을 참조하세요.

지원되는 알고리즘에 대한 구현 세부 정보는 알고리즘 참조 섹션을 참조하세요.

# 초기화 벡터(IV)는 어떻게 생성되며 어디에 저장되나요?

는 결정적 메서드를 AWS Encryption SDK 사용하여 프레임마다 다른 IV 값을 구성합니다. 이 절차를 통해 메시지 내에서 IV가 반복되지 않도록 합니다. ( AWS Encryption SDK for Java 및 버전 1.3.0 이전에 AWS Encryption SDK for Python는 각 프레임에 대해 고유한 IV 값을 AWS Encryption SDK 무작위로 생성했습니다.)

IV는이 AWS Encryption SDK 반환하는 암호화된 메시지에 저장됩니다. 자세한 내용은 AWS Encryption SDK 메시지 형식 참조 단원을 참조하십시오.

# 각 데이터 키는 어떻게 생성, 암호화 및 복호화되나요?

방법은 사용하는 키링 또는 마스터 키 공급자에 따라 다릅니다.

의 AWS KMS 키링과 마스터 키 공급자는 AWS KMS GenerateDataKey API 작업을 AWS Encryption SDK 사용하여 각 데이터 키를 생성하고 래핑 키로 암호화합니다. 추가 KMS 키로 데이터 키의 복사본을 암호화하려면 AWS KMS 암호화 작업을 사용합니다. 데이터 키를 복호화하려면 AWS KMS 복호화 작업을 사용합니다. 자세한 내용은 GitHub의 AWS Encryption SDK 사양에서 AWS KMS 키링을 참조하세요.

다른 키링은 각 프로그래밍 언어의 모범 사례 방법을 사용하여 데이터 키를 생성하고 암호화 및 복호화합니다. 자세한 내용은 GitHub의 사양에 있는 프레임워크 섹션에서 키링 또는 마스터 키 공급자의 AWS Encryption SDK 사양을 참조하세요.

# 데이터를 암호화하는 데 사용된 데이터 키를 추적하려면 어떻게 해야 하나요?

에서이 AWS Encryption SDK 작업을 수행합니다. 데이터를 암호화하면 SDK는 데이터 키를 암호화하고, 암호화된 키를 암호화된 데이터와 함께 반환되는 암호화된 메시지에 저장합니다. 데이터를 복호화할 때 AWS Encryption SDK 는 암호화된 메시지에서 암호화된 데이터 키를 추출하여 데이터 복호화에 사용합니다.

# 는 암호화된 데이터 키를 암호화된 데이터와 함께 어떻게 AWS Encryption SDK 저장하나요?

의 암호화 작업은 암호화된 데이터와 암호화된 데이터 키가 포함된 단일 데이터 구조인 암호화된 메시지를 AWS Encryption SDK 반환합니다. 메시지 형식은 최소 두 가지 부분인 헤더와 본문으로 구성됩니

다. 메시지 헤더에는 암호화된 데이터 키와, 메시지 본문 구성 방식에 대한 정보가 포함되어 있습니다. 메시지 본문에는 암호화된 데이터가 포함되어 있습니다. 알고리즘 제품군에 디지털 서명이 포함된 경우 메시지 형식에는 서명이 포함된 바닥글이 포함됩니다. 자세한 내용은 AWS Encryption SDK 메시지 형식 참조 단원을 참조하십시오.

## AWS Encryption SDK 메시지 형식은 내 암호화된 데이터에 얼마나 많은 오버헤드를 추가하나요?

에서 추가하는 오버헤드의 양은 다음을 포함한 여러 요인에 AWS Encryption SDK 따라 달라집니다.

- 일반 텍스트 데이터의 크기
- 지원되는 알고리즘 중 사용되는 알고리즘
- 추가 인증 데이터(AAD) 제공 여부 및 해당 AAD의 길이
- 래핑 키 또는 마스터 키의 수 및 유형
- 프레임 크기(프레임 데이터를 사용하는 경우)

를 기본 구성(하나 AWS KMS key 는 래핑 키(또는 마스터 키), AAD 없음, 프레임 처리되지 않은 데이터, 서명이 있는 암호화 알고리즘)과 AWS Encryption SDK 함께 사용하는 경우 오버헤드는 약 600바이트입니다. 일반적으로, AWS Encryption SDK 는 제공된 AAD를 제외하고 1KB 이하의 오버헤드를 더하는 것으로 가정할 수 있습니다. 자세한 내용은 AWS Encryption SDK 메시지 형식 참조 단원을 참조하십시오.

## 자체 마스터 키 공급자를 사용할 수 있나요?

예. 구현 세부 정보는 사용하는 지원되는 프로그래밍 언어에 따라 달라집니다. 그러나 지원되는 모든 언어를 사용하면 사용자 지정 암호화 자료 관리자(CMMs)Ms), 마스터 키 공급자, 키링, 마스터 키 및 래핑 키를 정의할 수 있습니다.

## 두 개 이상의 래핑 키로 데이터를 암호화할 수 있나요?

예. 키가 다른 리전에 있거나 복호화에 사용할 수 없는 경우 추가 래핑 키(또는 마스터 키)를 사용하여 데이터 키를 암호화하여 중복성을 추가할 수 있습니다.

여러 래핑 키로 데이터를 암호화하려면 여러 래핑 키가 있는 키링 또는 마스터 키 공급자를 만듭니다. 키링으로 작업할 때 여러 래핑 키를 사용하여 단일 키링을 만들거나 다중 키링을 만들 수 있습니다.

여러 래핑 키로 데이터를 암호화하면는 하나의 래핑 키를 AWS Encryption SDK 사용하여 일반 텍스트 데이터 키를 생성합니다. 데이터 키는 고유하며 래핑 키와 수학적으로 관련이 없습니다. 이 작업은 일반 텍스트 데이터 키와, 래핑 키로 암호화된 데이터 키 복사본을 반환합니다. 그러면 암호화 메서드는 다른 래핑 키로 데이터 키를 암호화합니다. 그 결과로 생성되는 암호화된 메시지에는 암호화된 데이터와, 각 래핑 키의 암호화된 데이터 키 1개가 포함됩니다.

암호화된 메시지는 암호화 작업에 사용되는 래핑 키 중 하나를 사용하여 복호화할 수 있습니다. 는 래핑 키를 AWS Encryption SDK 사용하여 암호화된 데이터 키를 해독합니다. 그런 다음 일반 텍스트 데이터 키를 사용하여 데이터를 복호화합니다.

## 로 암호화할 수 있는 데이터 유형은 무엇입니까 AWS Encryption SDK?

의 대부분의 프로그래밍 언어 구현은 원시 바이트(바이트 배열), I/O 스트림(바이트 스트림) 및 문자열을 암호화할 AWS Encryption SDK 수 있습니다. AWS Encryption SDK for .NET은 I/O 스트림을 지원하지 않습니다. 지원되는 프로그래밍 언어 각각에 대한 예제 코드를 제공합니다.

## 는 입력/출력(I/O) 스트림을 어떻게 암호화하고 AWS Encryption SDK 해독하나요?

는 기본 I/O 스트림을 래핑하는 암호화 또는 복호화 스트림을 AWS Encryption SDK 생성합니다. 암호화 또는 복호화 스트림은 읽기 또는 쓰기 호출에서 암호화 작업을 수행합니다. 예를 들어 기본 스트림에서 일반 텍스트 데이터를 읽고 암호화한 후 결과를 반환할 수 있습니다. 또는 기본 스트림에서 사이퍼텍스트를 읽고 복호화한 후 결과를 반환할 수 있습니다. 스트리밍을 지원하는 지원되는 프로그래밍 언어 각각의 스트림을 암호화 및 복호화하는 예제 코드를 제공합니다.

AWS Encryption SDK for .NET은 I/O 스트림을 지원하지 않습니다.

# AWS Encryption SDK 참조

AWS Encryption SDK와 호환되는 자체 암호화 라이브러리를 빌드할 때 이 페이지의 정보를 참조할 수 있습니다. 호환되는 자체 암호화 라이브러리를 빌드하는 경우가 아니라면 이 정보는 필요 없을 것입니다.

지원되는 프로그래밍 언어 중 하나 AWS Encryption SDK 에서를 사용하려면 섹션을 참조하세요프로그래밍 언어.

적절한 AWS Encryption SDK 구현의 요소를 정의하는 사양은 GitHub의 AWS Encryption SDK 사양을 참조하세요.

는 지원되는 알고리즘을 AWS Encryption SDK 사용하여 암호화된 데이터와 해당 암호화된 데이터 키가 포함된 단일 데이터 구조 또는 메시지를 반환합니다. 다음 주제에서는 알고리즘과 데이터 구조에 대해 설명합니다. 이 정보를 사용하여 이 SDK와 호환되는 사이퍼텍스트를 읽고 쓸 수 있는 라이브러리를 빌드하세요.

주제

- AWS Encryption SDK 메시지 형식 참조
- AWS Encryption SDK 메시지 형식 예제
- 에 대한 본문 추가 인증 데이터(AAD) 참조 AWS Encryption SDK
- AWS Encryption SDK 알고리즘 참조
- AWS Encryption SDK 초기화 벡터 참조
- AWS KMS 계층적 키링 기술 세부 정보

# AWS Encryption SDK 메시지 형식 참조

AWS Encryption SDK와 호환되는 자체 암호화 라이브러리를 빌드할 때 이 페이지의 정보를 참조할 수 있습니다. 호환되는 자체 암호화 라이브러리를 빌드하는 경우가 아니라면 이 정보는 필요 없을 것입니다.

지원되는 프로그래밍 언어 중 하나 AWS Encryption SDK 에서를 사용하려면 섹션을 참조하세요프로그래밍 언어.

적절한 AWS Encryption SDK 구현의 요소를 정의하는 사양은 GitHub의 AWS Encryption SDK 사양을 참조하세요.

의 암호화 작업은 암호화된 데이터(암호 텍스트)와 모든 암호화된 데이터 키가 포함된 단일 데이터 구조 또는 암호화된 메시지를 AWS Encryption SDK 반환합니다. 이 데이터 구조를 이해하거나 이를 읽고 쓰는 라이브러리를 구축하려면 메시지 형식을 이해해야 합니다.

메시지 형식은 최소 두 가지 부분인 헤더와 본문으로 구성됩니다. 경우에 따라 메시지 형식에 세 번째 부분인 바닥글이 포함되기도 합니다. 메시지 형식은 네트워크 바이트 순서로 정렬된 바이트 시퀀스를 정의하며 이를 빅 엔디안(big-endian) 형식이라고도 합니다. 메시지 형식은 헤더로 시작하여 본문, 바닥글(있는 경우) 순으로 이어집니다.

AWS Encryption SDK 에서 지원하는 알고리즘 제품군은 두 가지 메시지 형식 버전 중 하나를 사용합니다. 키 커밋이 없는 알고리즘 제품군은 메시지 형식 버전 1을 사용합니다. 키 커밋이 있는 알고리즘 제품군은 메시지 형식 버전 2를 사용합니다.

주제



## 헤더 구조

메시지 헤더에는 암호화된 데이터 키와 메시지 본문 구성 방식에 대한 정보가 포함되어 있습니다. 다음 표에서는 메시지 형식 버전 1 및 2의 헤더를 구성하는 필드에 대해 설명합니다. 표시된 순서대로 바이트가 추가됩니다.

존재하지 않음 값은 해당 버전의 메시지 형식에 필드가 존재하지 않음을 나타냅니다. 굵은 텍스트는 각 버전의 값이 다름을 나타냅니다.

> **Note**
>
> 이 테이블의 모든 데이터를 보려면 가로 또는 세로로 스크롤해야 할 수도 있습니다.

## 헤더 구조

| Field | 메시지 형식 버전 1<br><br>길이(바이트) | 메시지 형식 버전 2<br><br>길이(바이트) |
|---|---|---|
| Version | 1 | 1 |
| Type | 1 | 존재하지 않음 |
| Algorithm ID | 2 | 2 |
| Message ID | 16 | 32 |
| AAD Length | 2<br><br>암호화 컨텍스트가 비어 있는 경우 2바이트 AAD 길이 필드의 값은 0입니다. | 2<br><br>암호화 컨텍스트가 비어 있는 경우 2바이트 AAD 길이 필드의 값은 0입니다. |
| AAD | 변수. 이 필드의 길이는 이전 2바이트(AAD 길이 필드)에 표시됩니다.<br><br>암호화 컨텍스트가 비어 있는 경우 헤더에 AAD 필드가 없습니다. | 변수. 이 필드의 길이는 이전 2바이트(AAD 길이 필드)에 표시됩니다.<br><br>암호화 컨텍스트가 비어 있는 경우 헤더에 AAD 필드가 없습니다. |
| Encrypted Data Key Count | 2 | 2 |
| Encrypted Data Key(s) | 변수. 암호화된 데이터 키의 수와 각 데이터 키의 길이에 따라 결정됩니다. | 변수. 암호화된 데이터 키의 수와 각 데이터 키의 길이에 따라 결정됩니다. |
| Content Type | 1 | 1 |
| Reserved | 4 | 존재하지 않음 |
| IV Length | 1 | 존재하지 않음 |
| Frame Length | 4 | 4 |

| Field | 메시지 형식 버전 1<br><br>길이(바이트) | 메시지 형식 버전 2<br><br>길이(바이트) |
| --- | --- | --- |
| Algorithm Suite Data | 존재하지 않음 | 변수. 메시지를 생성한 알고리즘에 의해 결정됩니다. |
| Header Authentication | 변수. 메시지를 생성한 알고리즘에 의해 결정됩니다. | 변수. 메시지를 생성한 알고리즘에 의해 결정됩니다. |

버전

이 메시지 형식의 버전입니다. 버전은 1 또는 2이며 바이트로 인코딩되어 16진수 표기법으로 `01` 또는 `02`입니다.

유형

이 메시지 형식의 유형입니다. 유형은 구조의 종류를 나타냅니다. 지원되는 유일한 유형은 고객이 인증한 암호화된 데이터라고 설명됩니다. 해당 유형 값은 128이며 바이트로 인코딩되어 16진수 표기법으로 `80`입니다.

메시지 형식 버전 2에는 이 필드가 존재하지 않습니다.

알고리즘 ID

사용된 알고리즘의 식별자입니다. 이 값은 부호 없는 16비트 정수로 해석되는 2바이트 값입니다. 알고리즘에 대한 자세한 내용은 AWS Encryption SDK 알고리즘 참조 섹션을 참조하세요.

메시지 ID

메시지를 식별하는 임의로 생성된 값입니다. 메시지 ID의 특징:

- 암호화된 메시지를 고유하게 식별합니다.
- 메시지 헤더를 메시지 본문에 약하게 바인딩합니다.
- 여러 암호화된 메시지와 함께 데이터 키를 안전하게 재사용할 수 있는 메커니즘을 제공합니다.
- AWS Encryption SDK에서 실수로 데이터 키를 재사용하거나 키가 부족해지는 상황을 방지합니다.

이 값은 메시지 형식 버전 1에서는 128비트이고 버전 2에서는 256비트입니다.

AAD 길이

추가 인증 데이터(AAD)의 길이입니다. 이 값은 AAD가 포함된 바이트 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다.

암호화 컨텍스트가 비어 있는 경우 AAD 길이 필드의 값은 0입니다.

AAD

추가 인증 데이터입니다. AAD는 암호화 컨텍스트의 인코딩으로, 각 키와 값이 UTF-8로 인코딩된 문자열인 키-값 페어의 배열입니다. 암호화 컨텍스트는 바이트 시퀀스로 변환되어 AAD 값으로 사용됩니다. 암호화 컨텍스트가 비어 있는 경우 헤더에 AAD 필드가 없습니다.

서명이 있는 알고리즘을 사용하는 경우 암호화 컨텍스트에는 키-값 페어 `{'aws-crypto-public-key', Qtxt}`가 포함되어야 합니다. Qtxt는 SEC 1 버전 2.0에 따라 압축된 후 base64로 인코딩된 타원 곡선의 점 Q를 나타냅니다. 암호화 컨텍스트에는 추가 값이 포함될 수 있지만 구성된 AAD의 최대 길이는 2^16 - 1바이트입니다.

다음 표에서는 AAD를 구성하는 필드에 대해 설명합니다. 키-값 페어는 키별로 UTF-8 문자 코드에 따라 오름차순으로 정렬됩니다. 표시된 순서대로 바이트가 추가됩니다.

AAD 구조

| Field | 길이(바이트) |
| --- | --- |
| Key-Value Pair Count | 2 |
| Key Length | 2 |
| Key | 변수. 이전 2바이트에 지정된 값(키 길이)과 동일합니다. |
| Value Length | 2 |
| Value | 변수. 이전 2바이트에 지정된 값(값 길이)과 동일합니다. |

키-값 페어 수

AAD의 키-값 페어 수입니다. 이 값은 AAD에서 키-값 페어의 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다. AAD의 최대 키-값 페어 수는 2^16 - 1입니다.

암호화 컨텍스트가 없거나 암호화 컨텍스트가 비어 있는 경우 이 필드는 AAD 구조에 존재하지 않습니다.

키 길이

키-값 페어의 키 길이입니다. 이 값은 키가 포함된 바이트 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다.

키

키-값 페어의 키입니다. UTF-8로 인코딩된 바이트 시퀀스입니다.

값 길이

키-값 페어의 값 길이입니다. 이 값은 값이 포함된 바이트 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다.

값

키-값 페어의 값입니다. UTF-8로 인코딩된 바이트 시퀀스입니다.

암호화된 데이터 키 수

암호화된 데이터 키의 수입니다. 이 값은 암호화된 데이터 키의 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다. 각 메시지의 암호화된 데이터 키의 최대 수는 65,535개(2^16 - 1)입니다.

암호화된 데이터 키(들)

암호화된 데이터 키의 시퀀스입니다. 시퀀스의 길이는 암호화된 데이터 키의 수와 각 데이터 키의 길이에 따라 결정됩니다. 시퀀스에는 암호화된 데이터 키가 하나 이상 포함되어 있습니다.

다음 표에서는 암호화된 각 데이터 키를 구성하는 필드에 대해 설명합니다. 표시된 순서대로 바이트가 추가됩니다.

암호화된 데이터 키 구조

| Field | 길이(바이트) |
| --- | --- |
| Key Provider ID Length | 2 |
| Key Provider ID | 변수. 이전 2바이트에 지정된 값(키 공급자 ID 길이)과 동일합니다. |
| Key Provider Information Length | 2 |

| Field | 길이(바이트) |
| --- | --- |
| Key Provider Information | 변수. 이전 2바이트에 지정된 값(키 공급자 정보 길이)과 동일합니다. |
| Encrypted Data Key Length | 2 |
| Encrypted Data Key | 변수. 이전 2바이트에 지정된 값(암호화된 데이터 키 길이)과 동일합니다. |

키 공급자 ID 길이

키 공급자 식별자의 길이입니다. 이 값은 키 공급자 ID가 포함된 바이트 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다.

키 공급자 ID

키 공급자 식별자입니다. 암호화된 데이터 키의 공급자를 나타내는 데 사용되며 확장 가능하도록 설계되었습니다.

키 공급자 정보 길이

키 공급자 정보의 길이입니다. 이 값은 키 공급자 정보가 포함된 바이트 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다.

키 공급자 정보

키 공급자 정보입니다. 키 공급자에 의해 결정됩니다.

AWS KMS 이 마스터 키 공급자이거나 AWS KMS 키링을 사용하는 경우이 값에는의 Amazon 리소스 이름(ARN)이 포함됩니다 AWS KMS key.

암호화된 데이터 키 길이

암호화된 데이터 키의 길이입니다. 이 값은 암호화된 데이터 키가 포함된 바이트 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다.

암호화된 데이터 키

암호화된 데이터 키입니다. 키 공급자에 의해 암호화된 데이터 암호화 키입니다.

콘텐츠 유형

암호화된 데이터의 유형으로, 프레임 처리되지 않거나 프레임 처리됩니다.

> **Note**
>
> 가능하면 프레임 처리된 데이터를 사용하세요. 는 레거시 용도로만 프레임 처리되지 않은 데이터를 AWS Encryption SDK 지원합니다. 의 일부 언어 구현은 여전히 프레임 처리되지 않은 사이퍼텍스트를 생성할 AWS Encryption SDK 수 있습니다. 지원되는 모든 언어 구현은 프레임 처리된 사이퍼텍스트와 프레임 처리되지 않은 사이퍼텍스트를 복호화할 수 있습니다.

프레임 처리된 데이터는 동일한 길이의 여러 부분으로 나뉘며 각 부분은 개별적으로 암호화됩니다. 프레임 처리된 콘텐츠는 유형 2이며 바이트로 인코딩되어 16진수 표기법으로 `02`입니다.

프레임 처리되지 않은 데이터는 분할되지 않으며 암호화된 단일의 Blob입니다. 프레임 처리되지 않은 콘텐츠는 유형 1이며 바이트로 인코딩되어 16진수 표기법으로 `01`입니다.

예약됨

4바이트의 예약된 시퀀스입니다. 이 값은 0이어야 합니다. 바이트로 인코딩되어 16진수 표기법으로 `00 00 00 00`입니다(즉, 0과 같은 32비트 정수 값의 4바이트 시퀀스).

메시지 형식 버전 2에는 이 필드가 존재하지 않습니다.

IV 길이

초기화 벡터(IV)의 길이입니다. 이 값은 IV를 포함한 바이트 수를 지정하는 부호 없는 8비트 정수로 해석되는 1바이트 값입니다. 이 값은 메시지를 생성한 알고리즘의 IV 바이트 값에 의해 결정됩니다.

메시지 헤더에 결정론적 IV 값을 사용하는 알고리즘 제품군만 지원하는 메시지 형식 버전 2에는 이 필드가 존재하지 않습니다.

프레임 길이

프레임 처리된 데이터의 각 프레임의 길이입니다. 이 값은 각 프레임의 바이트 수를 지정하는 부호 없는 32비트 정수로 해석되는 4바이트 값입니다. 데이터가 프레임 처리되어 있지 않은 경우, 즉 `Content Type` 필드의 값이 1인 경우 이 값은 0이어야 합니다.

> **Note**
>
> 가능하면 프레임 처리된 데이터를 사용하세요. 는 레거시 용도로만 프레임 처리되지 않은 데이터를 AWS Encryption SDK 지원합니다. 의 일부 언어 구현은 여전히 프레임 처리되지 않은 사이퍼텍스트를 생성할 AWS Encryption SDK 수 있습니다. 지원되는 모든 언어 구현

은 프레임 처리된 사이퍼텍스트와 프레임 처리되지 않은 사이퍼텍스트를 복호화할 수 있습니다.

알고리즘 제품군 데이터

메시지를 생성한 알고리즘에 필요한 추가 데이터입니다. 길이와 내용은 알고리즘에 의해 결정됩니다. 길이는 0일 수 있습니다.

메시지 형식 버전 1에는 이 필드가 존재하지 않습니다.

헤더 인증

헤더 인증은 메시지를 생성한 알고리즘에 의해 결정됩니다. 헤더 인증은 전체 헤더에 대해 계산됩니다. IV와 인증 태그로 구성됩니다. 표시된 순서대로 바이트가 추가됩니다.

헤더 인증 구조

| Field | 버전 1.0에서의 길이(바이트) | 버전 2.0에서의 길이(바이트) |
|---|---|---|
| IV | 변수. 메시지를 생성한 알고리즘의 IV 바이트 값으로 결정됩니다. | 해당 사항 없음 |
| Authentication Tag | 변수. 메시지를 생성한 알고리즘의 인증 태그 바이트 값에 의해 결정됩니다. | 변수. 메시지를 생성한 알고리즘의 인증 태그 바이트 값에 의해 결정됩니다. |

IV

헤더 인증 태그를 계산하는 데 사용되는 초기화 벡터(IV)입니다.

메시지 형식 버전 2의 헤더에는 이 필드가 존재하지 않습니다. 메시지 형식 버전 2는 메시지 헤더에 결정론적 IV 값을 사용하는 알고리즘 제품군만 지원합니다.

인증 태그

헤더의 인증 값입니다. 헤더의 전체 내용을 인증하는 데 사용됩니다.

# 본문 구조

메시지 본문에는 사이퍼텍스트라고 하는 암호화된 데이터가 포함되어 있습니다. 본문 구조는 콘텐츠 유형(프레임 처리되지 않음 또는 프레임 처리됨)에 따라 달라집니다. 다음 섹션에서는 각 콘텐츠 유형에 대한 메시지 본문의 형식에 대해 설명합니다. 메시지 본문 구조는 메시지 형식 버전 1 및 2에서 동일합니다.

주제



## 프레임 처리되지 않은 데이터

프레임 처리되지 않은 데이터는 고유한 IV 및 본문 AAD를 사용하여 단일 Blob으로 암호화됩니다.

> **Note**
>
> 가능하면 프레임 처리된 데이터를 사용하세요. 는 레거시 용도로만 프레임 처리되지 않은 데이터를 AWS Encryption SDK 지원합니다. 의 일부 언어 구현은 여전히 프레임 처리되지 않은 사이퍼텍스트를 생성할 AWS Encryption SDK 수 있습니다. 지원되는 모든 언어 구현은 프레임 처리된 사이퍼텍스트와 프레임 처리되지 않은 사이퍼텍스트를 복호화할 수 있습니다.

아래 표에 프레임 처리되지 않은 데이터를 구성하는 필드가 나와 있습니다. 표시된 순서대로 바이트가 추가됩니다.

프레임 처리되지 않은 본문 구조

| Field | 길이(바이트) |
| --- | --- |
| IV | 변수. 헤더의 IV Length 바이트에 지정된 값과 동일합니다. |
| Encrypted Content Length | 8 |
| Encrypted Content | 변수. 이전 8바이트에 지정된 값(암호화된 콘텐츠 길이)과 동일합니다. |
| Authentication Tag | 변수. 사용된 알고리즘 구현에 따라 결정됩니다. |

IV

암호화 알고리즘과 함께 사용할 초기화 벡터(IV)입니다.

암호화된 콘텐츠 길이

암호화된 콘텐츠 또는 사이퍼텍스트의 길이입니다. 이 값은 암호화된 콘텐츠가 포함된 바이트 수를 지정하는 부호 없는 64비트 정수로 해석되는 8바이트 값입니다.

기술적으로, 허용되는 최대값은 2^63 - 1 또는 8엑스비바이트(8EiB)입니다. 그러나 구현된 알고리즘에 따른 제한으로 인해 실제 최대값은 2^36 - 32 또는 64기비바이트(64GiB)입니다.

> **Note**
>
> 이 SDK의 Java 구현에서는 언어 제한으로 인해 이 값을 2^31 - 1 또는 2기비바이트(2GiB)로 추가 제한합니다.

암호화된 콘텐츠

암호화 알고리즘에서 반환된 암호화된 콘텐츠(사이퍼텍스트)입니다.

인증 태그

본문에 대한 인증 값입니다. 메시지 본문을 인증하는 데 사용됩니다.

## 프레임 처리된 데이터

프레임 처리된 데이터에서 일반 텍스트 데이터는 프레임이라는 동일한 길이의 파트로 나뉩니다. 는 고유한 IV 및 본문 AAD를 사용하여 각 프레임을 별도로 AWS Encryption SDK 암호화합니다.

> **Note**
>
> 가능하면 프레임 처리된 데이터를 사용하세요. 는 레거시 용도로만 프레임 처리되지 않은 데이터를 AWS Encryption SDK 지원합니다. 의 일부 언어 구현은 여전히 프레임 처리되지 않은 사이퍼텍스트를 생성할 AWS Encryption SDK 수 있습니다. 지원되는 모든 언어 구현은 프레임 처리된 사이퍼텍스트와 프레임 처리되지 않은 사이퍼텍스트를 복호화할 수 있습니다.

프레임 길이(프레임 내 암호화된 콘텐츠 길이)는 메시지마다 다를 수 있습니다. 프레임의 최대 바이트 수는 2^32 - 1입니다. 메시지의 최대 프레임 수는 2^32 - 1입니다.

프레임에는 일반 프레임과 최종 프레임, 이렇게 두 가지 유형이 있습니다. 모든 메시지는 최종 프레임으로 구성되거나 포함해야 합니다.

메시지의 모든 일반 프레임은 프레임 길이가 동일합니다. 최종 프레임의 프레임 길이는 서로 다를 수 있습니다.

프레임 처리된 데이터의 프레임 구성은 암호화된 콘텐츠의 길이에 따라 다릅니다.

- 프레임 길이와 동일 - 암호화된 콘텐츠 길이가 일반 프레임의 프레임 길이와 동일하면 메시지는 데이터 뒤에 길이가 0인 최종 프레임을 포함하는 일반 프레임으로 구성될 수 있습니다. 또는 메시지는 데이터를 포함하는 최종 프레임으로만 구성될 수 있습니다. 이 경우, 최종 프레임은 일반 프레임과 프레임 길이가 동일합니다.
- 프레임 길이의 배수 - 암호화된 콘텐츠 길이가 일반 프레임의 프레임 길이의 정확한 배수인 경우 메시지는 데이터 뒤에 길이가 0인 최종 프레임을 포함하는 일반 프레임으로 끝날 수 있습니다. 또는 메시지는 데이터를 포함하는 최종 프레임으로 끝날 수 있습니다. 이 경우, 최종 프레임은 일반 프레임과 프레임 길이가 동일합니다.
- 프레임 길이의 배수가 아님 - 암호화된 콘텐츠 길이가 일반 프레임의 프레임 길이의 정확한 배수가 아닌 경우, 최종 프레임에 나머지 데이터가 포함됩니다. 최종 프레임의 프레임 길이는 일반 프레임의 프레임 길이보다 짧습니다.
- 프레임 길이보다 짧음 - 암호화된 콘텐츠 길이가 일반 프레임의 프레임 길이보다 짧은 경우 메시지는 모든 데이터를 포함하는 최종 프레임으로 구성됩니다. 최종 프레임의 프레임 길이는 일반 프레임의 프레임 길이보다 짧습니다.

다음 표에서는 프레임을 구성하는 필드에 대해 설명합니다. 표시된 순서대로 바이트가 추가됩니다.

프레임 본문 구조, 일반 프레임

| Field | 길이(바이트) |
| --- | --- |
| Sequence Number | 4 |
| IV | 변수. 헤더의 IV Length 바이트에 지정된 값과 동일합니다. |
| Encrypted Content | 변수. 헤더의 Frame Length에 지정된 값과 같습니다. |
| Authentication Tag | 변수. 헤더의 Algorithm ID에 지정된 대로 사용된 알고리즘에 따라 결정됩니다. |

시퀀스 번호

프레임 시퀀스 번호입니다. 프레임의 증분 카운터 번호입니다. 이 값은 부호 없는 32비트 정수로 해석되는 4바이트 값입니다.

프레임 데이터는 시퀀스 번호 1에서 시작해야 합니다. 후속 프레임은 순서대로 배치되어야 하며 이전 프레임보다 1씩 증가해야 합니다. 그러지 않으면 복호화 프로세스가 중지되고 오류가 보고됩니다.

IV

프레임의 초기화 벡터(IV)입니다. SDK는 결정론적 방법을 사용하여 메시지의 각 프레임에 대해 서로 다른 IV를 구성합니다. 길이는 사용된 알고리즘 제품군에 따라 지정됩니다.

암호화된 콘텐츠

암호화 알고리즘에서 반환한 프레임의 암호화된 콘텐츠(사이퍼텍스트)입니다.

인증 태그

프레임의 인증 값입니다. 전체 프레임을 인증하는 데 사용됩니다.

프레임 본문 구조, 최종 프레임

| Field | 길이(바이트) |
| --- | --- |
| Sequence Number End | 4 |
| Sequence Number | 4 |
| IV | 변수. 헤더의 IV Length 바이트에 지정된 값과 동일합니다. |
| Encrypted Content Length | 4 |
| Encrypted Content | 변수. 이전 4바이트에 지정된 값(암호화된 콘텐츠 길이)과 동일합니다. |
| Authentication Tag | 변수. 헤더의 Algorithm ID에 지정된 대로 사용된 알고리즘에 따라 결정됩니다. |

시퀀스 번호 끝

최종 프레임을 나타내는 표시기입니다. 해당 값은 4바이트로 인코딩되어 16진수 표기법으로 FF FF FF FF입니다.

시퀀스 번호

프레임 시퀀스 번호입니다. 프레임의 증분 카운터 번호입니다. 이 값은 부호 없는 32비트 정수로 해석되는 4바이트 값입니다.

프레임 데이터는 시퀀스 번호 1에서 시작해야 합니다. 후속 프레임은 순서대로 배치되어야 하며 이전 프레임보다 1씩 증가해야 합니다. 그러지 않으면 복호화 프로세스가 중지되고 오류가 보고됩니다.

IV

프레임의 초기화 벡터(IV)입니다. SDK는 결정론적 방법을 사용하여 메시지의 각 프레임에 대해 서로 다른 IV를 구성합니다. IV 길이의 길이는 알고리즘 제품군에 따라 지정됩니다.

암호화된 콘텐츠 길이

암호화된 콘텐츠의 길이입니다. 이 값은 프레임의 암호화된 콘텐츠가 포함된 바이트 수를 지정하는 부호 없는 32비트 정수로 해석되는 4바이트 값입니다.

암호화된 콘텐츠

암호화 알고리즘에서 반환한 프레임의 암호화된 콘텐츠(사이퍼텍스트)입니다.

인증 태그

프레임의 인증 값입니다. 전체 프레임을 인증하는 데 사용됩니다.

## 바닥글 구조

서명 기능이 있는 알고리즘을 사용하는 경우 메시지 형식에 바닥글이 포함됩니다. 메시지 바닥글에는 메시지 헤더 및 본문에 대해 계산된 디지털 서명이 포함됩니다. 다음 표에서는 바닥글을 구성하는 필드에 대해 설명합니다. 표시된 순서대로 바이트가 추가됩니다. 메시지 바닥글 구조는 메시지 형식 버전 1 및 2에서 동일합니다.

바닥글 구조

| Field | 길이(바이트) |
| --- | --- |
| Signature Length | 2 |

| Field | 길이(바이트) |
| --- | --- |
| Signature | 변수. 이전 2바이트에 지정된 값(서명 길이)과 동일합니다. |

서명 길이

서명의 길이입니다. 이 값은 서명이 포함된 바이트 수를 지정하는 부호 없는 16비트 정수로 해석되는 2바이트 값입니다.

서명

서명입니다.

# AWS Encryption SDK 메시지 형식 예제

AWS Encryption SDK와 호환되는 자체 암호화 라이브러리를 빌드할 때 이 페이지의 정보를 참조할 수 있습니다. 호환되는 자체 암호화 라이브러리를 빌드하는 경우가 아니라면 이 정보는 필요 없을 것입니다.

지원되는 프로그래밍 언어 중 하나 AWS Encryption SDK 에서를 사용하려면 섹션을 참조하세요프로그래밍 언어.

적절한 AWS Encryption SDK 구현의 요소를 정의하는 사양은 GitHub의 AWS Encryption SDK 사양을 참조하세요.

다음 주제에서는 AWS Encryption SDK 메시지 형식의 예를 보여줍니다. 각 예제는 원시 바이트를 16진수 표기법으로 표시한 다음 해당 바이트가 나타내는 내용에 대한 설명을 보여줍니다.

주제

- 프레임 처리된 데이터(메시지 형식 버전 1)
- 프레임 처리된 데이터(메시지 형식 버전 2)
- 프레임 처리되지 않은 데이터(메시지 형식 버전 1)

# 프레임 처리된 데이터(메시지 형식 버전 1)

다음 예제는 메시지 형식 버전 1의 프레임 처리된 데이터에 대한 메시지 형식을 보여줍니다.

```
+--------+
| Header |
+--------+
01                                      Version (1.0)
80                                      Type (128, customer authenticated encrypted
 data)
0378                                    Algorithm ID (see #### ##)
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
```

## 프레임 처리된 데이터(메시지 형식 버전 2)

다음 예제는 메시지 형식 버전 2의 프레임 처리된 데이터에 대한 메시지 형식을 보여줍니다.

```
+--------+
| Header |
+--------+
02                                        Version (2.0)
0578                                      Algorithm ID (see Algorithms reference)
122747eb 21dfe39b 38631c61 7fad7340
```

```
cc621a30 32a11cc3 216d0204 fd148459        Message ID (random 256-bit value)
008e                                       AAD Length (142)
0004                                       AAD Key-Value Pair Count (4)
0005                                       AAD Key-Value Pair 1, Key Length (5)
30546869 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 79707469 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)
41746733 72703845 41345161 36706669        AAD Key-Value Pair 4, Value
 ("QXRnM3JwOEVBNFFhNnBmaTk3MUlTNTk3NHpOMnlZWE5vSmtwRHFPc0dIYkVaVDRqME5OMlFkRStmbTFVY01WdThnPT0=
39373149 53353937 347a4e32 7959584e
6f4a6b70 44714f73 47486245 5a54346a
304e4e32 5164452b 666d3155 634d5675
38673d3d
0001                                       Encrypted Data Key Count (1)
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-2:658956600833:key/b3537ef1-
d8dc-4780-9f5a-55776cbb2f7f")
6573742d 323a3635 38393536 36303038
33333a6b 65792f62 33353337 6566312d
64386463 2d343738 302d3966 35612d35
35373736 63626232 663766
00a7                                       Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010100 7840f38c 275e3109 7416c107        Encrypted Data Key 1, Encrypted Data Key
29515057 1964ada3 ef1c21e9 4c8ba0bd
```

```
bc9d0fb4 14000000 7e307c06 092a8648
86f70d01 0706a06f 306d0201 00306806
092a8648 86f70d01 0701301e 06096086
48016503 04012e30 11040c39 32d75294
06063803 f8460802 0110803b 2a46bc23
413196d2 903bf1d7 3ed98fc8 a94ac6ed
e00ee216 74ec1349 12777577 7fa052a5
ba62e9e4 f2ac8df6 bcb1758f 2ce0fb21
cc9ee5c9 7203bb
02                                      Content Type (2, framed data)
00001000                                Frame Length (4096)
05cd035b 29d5499d 4587570b 87502afe      Algorithm Suite Data (key commitment)
634f7b2c c3df2aa9 88a10105 4a2c7687
76cb339f 2536741f 59a1c202 4f2594ab      Authentication Tag
+------+
| Body |
+------+
ffffffff                                Final Frame, Sequence Number End
00000001                                Final Frame, Sequence Number (1)
00000000 00000000 00000001              Final Frame, IV
00000009                                Final Frame, Encrypted Content Length (9)
fa6e39c6 02927399 3e                    Final Frame, Encrypted Content
f683a564 405d68db eeb0656c d57c9eb0      Final Frame, Authentication Tag
+--------+
| Footer |
+--------+
0067                                    Signature Length (103)
30650230 2a1647ad 98867925 c1712e8f      Signature
ade70b3f 2a2bc3b8 50eb91ef 56cfdd18
967d91d8 42d92baf 357bba48 f636c7a0
869cade2 023100aa ae12d08f 8a0afe85
e5054803 110c9ed8 11b2e08a c4a052a9
074217ea 3b01b660 534ac921 bf091d12
3657e2b0 9368bd
```

## 프레임 처리되지 않은 데이터(메시지 형식 버전 1)

다음 예제에서는 프레임 처리되지 않은 데이터에 대한 메시지 형식을 보여줍니다.

> **Note**
>
> 가능하면 프레임 처리된 데이터를 사용하세요. 는 레거시 용도로만 프레임 처리되지 않은 데이터를 AWS Encryption SDK 지원합니다. 의 일부 언어 구현은 여전히 프레임 처리되지 않은 사이퍼텍스트를 생성할 AWS Encryption SDK 수 있습니다. 지원되는 모든 언어 구현은 프레임 처리된 사이퍼텍스트와 프레임 처리되지 않은 사이퍼텍스트를 복호화할 수 있습니다.

```
+--------+
| Header |
+--------+
01                                      Version (1.0)
80                                      Type (128, customer authenticated encrypted
 data)
0378                                    Algorithm ID (see #### ##)
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, nonframed data)
00000000                                Reserved
0C                                      IV Length (12)
00000000                                Frame Length (0, nonframed 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
```

# 에 대한 본문 추가 인증 데이터(AAD) 참조 AWS Encryption SDK

AWS Encryption SDK와 호환되는 자체 암호화 라이브러리를 빌드할 때 이 페이지의 정보를 참조할 수 있습니다. 호환되는 자체 암호화 라이브러리를 빌드하는 경우가 아니라면 이 정보는 필요 없을 것입니다.

지원되는 프로그래밍 언어 중 하나 AWS Encryption SDK 에서를 사용하려면 섹션을 참조하세요프로그래밍 언어.

적절한 AWS Encryption SDK 구현의 요소를 정의하는 사양은 GitHub의 AWS Encryption SDK 사양을 참조하세요.

각 암호화 작업에 대해 AES-GCM 알고리즘에 추가 인증 데이터(AAD)를 제공해야 합니다. 이는 프레임 처리되었거나 처리되지 않은 본문 데이터 둘 다에 대해서도 마찬가지입니다. AAD와 GCM(Galois/Counter Mode)에서의 AAD 사용 방법에 대한 자세한 내용은 Recommendations for Block Cipher Modes of Operations: Galois/Counter Mode (GCM) and GMAC(블록 암호 운용 방식에 대한 권장 사항: GCM(Galois/Counter Mode) 및 GMAC)를 참조하세요.

다음 표에서는 본문 AAD를 구성하는 필드에 대해 설명합니다. 표시된 순서대로 바이트가 추가됩니다.

본문 AAD 구조

| Field | 길이(바이트) |
| --- | --- |
| Message ID | 16 |
| Body AAD Content | 변수. 다음 목록의 본문 AAD 콘텐츠를 참조하세요. |
| Sequence Number | 4 |
| Content Length | 8 |

메시지 ID

메시지 헤더에 설정된 것과 동일한 Message ID 값입니다.

본문 AAD 콘텐츠

사용된 본문 데이터 유형에 따라 결정되는 UTF-8로 인코딩된 값입니다.

프레임 처리되지 않은 데이터의 경우 `AWSKMSEncryptionClient Single Block` 값을 사용합니다.

프레임 처리된 데이터의 일반 프레임의 경우 `AWSKMSEncryptionClient Frame` 값을 사용합니다.

프레임 처리된 데이터의 최종 프레임의 경우 `AWSKMSEncryptionClient Final Frame` 값을 사용합니다.

시퀀스 번호

부호 없는 32비트 정수로 해석되는 4바이트 값입니다.

프레임 처리된 데이터의 경우 이는 프레임 시퀀스 번호입니다.

프레임 처리되지 않은 데이터의 경우 값 1을 사용합니다. 이 값은 16진수 표기법에서 4바이트 `00 00 00 01`로 인코딩됩니다.

콘텐츠 길이

암호화를 위해 알고리즘에 제공되는 일반 텍스트 데이터의 길이(바이트)입니다. 이 값은 부호 없는 64비트 정수로 해석되는 8바이트 값입니다.

# AWS Encryption SDK 알고리즘 참조

AWS Encryption SDK와 호환되는 자체 암호화 라이브러리를 빌드할 때 이 페이지의 정보를 참조할 수 있습니다. 호환되는 자체 암호화 라이브러리를 빌드하는 경우가 아니라면 이 정보는 필요 없을 것입니다.

지원되는 프로그래밍 언어 중 하나 AWS Encryption SDK 에서를 사용하려면 섹션을 참조하세요프로그래밍 언어.

적절한 AWS Encryption SDK 구현의 요소를 정의하는 사양은 GitHub의 AWS Encryption SDK 사양을 참조하세요.

와 호환되는 사이퍼텍스트를 읽고 쓸 수 있는 자체 라이브러리를 구축하는 경우 AWS Encryption SDK 가 지원되는 알고리즘 제품군을 AWS Encryption SDK 구현하여 원시 데이터를 암호화하는 방법을 이해해야 합니다.

는 다음 알고리즘 제품군을 AWS Encryption SDK 지원합니다. 모든 AES-GCM 알고리즘 제품군에는 12바이트 초기화 벡터와 16바이트 AES-GCM 인증 태그가 있습니다. 기본 알고리즘 제품군은 AWS Encryption SDK 버전 및 선택한 키 커밋 정책에 따라 다릅니다. 자세한 내용은 커밋 정책 및 알고리즘 제품군을 참조하세요.

AWS Encryption SDK 알고리즘 스위트

| 알고리즘 ID | 메시지 형식 버전 | 암호화 알고리즘 | 데이터 키 길이(비트) | 키 유도 알고리즘 | 서명 알고리즘 | 키 커밋 알고리즘 | 알고리즘 제품군 데이터 길이(바이트) |
|---|---|---|---|---|---|---|---|
| 05 78 | 0x02 | AES-GCM | 256 | HKDF(SHA 512 사용) | ECDSA(P-3 84 및 | HKDF(SHA 512 사용) | 32(키 커밋) |

| 알고리즘 ID | 메시지 형식 버전 | 암호화 알고리즘 | 데이터 키 길이(비트) | 키 유도 알고리즘 | 서명 알고리즘 | 키 커밋 알고리즘 | 알고리즘 제품군 데이터 길이(바이트) |
|---|---|---|---|---|---|---|---|
| | | | | | SHA-384 사용) | | |
| 04 78 | 0x02 | AES-GCM | 256 | HKDF(SHA 512 사용) | 없음 | HKDF(SHA 512 사용) | 32(키 커밋) |
| 03 78 | 0x01 | AES-GCM | 256 | HKDF(SHA 384 사용) | ECDSA(P-3 84 및 SHA-384 사용) | 없음 | 해당 사항 없음 |
| 03 46 | 0x01 | AES-GCM | 192 | HKDF(SHA 384 사용) | ECDSA(P-3 84 및 SHA-384 사용) | 없음 | 해당 사항 없음 |
| 02 14 | 0x01 | AES-GCM | 128 | HKDF(SHA 256 사용) | ECDSA(P-2 56 및 SHA-256 사용) | 없음 | 해당 사항 없음 |
| 01 78 | 0x01 | AES-GCM | 256 | HKDF(SHA 256 사용) | 없음 | 없음 | 해당 사항 없음 |
| 01 46 | 0x01 | AES-GCM | 192 | HKDF(SHA 256 사용) | 없음 | 없음 | 해당 사항 없음 |
| 01 14 | 0x01 | AES-GCM | 128 | HKDF(SHA 256 사용) | 없음 | 없음 | 해당 사항 없음 |
| 00 78 | 0x01 | AES-GCM | 256 | 없음 | 없음 | 없음 | 해당 사항 없음 |

| 알고리즘 ID | 메시지 형식 버전 | 암호화 알고리즘 | 데이터 키 길이(비트) | 키 유도 알고리즘 | 서명 알고리즘 | 키 커밋 알고리즘 | 알고리즘 제품군 데이터 길이(바이트) |
|---|---|---|---|---|---|---|---|
| `00 46` | 0x01 | AES-GCM | 192 | 없음 | 없음 | 없음 | 해당 사항 없음 |
| `00 14` | 0x01 | AES-GCM | 128 | 없음 | 없음 | 없음 | 해당 사항 없음 |

알고리즘 ID

알고리즘 구현을 고유하게 식별하는 2바이트 16진수 값입니다. 이 값은 사이퍼텍스트의 메시지 헤더에 저장됩니다.

메시지 형식 버전

메시지 형식의 버전입니다. 키 커밋이 있는 알고리즘 제품군은 메시지 형식 버전 2(0x02)를 사용합니다. 키 커밋이 없는 알고리즘 제품군은 메시지 형식 버전 1(0x01)을 사용합니다.

알고리즘 제품군 데이터 길이

알고리즘 제품군에만 관련된 데이터의 길이(바이트)입니다. 이 필드는 메시지 형식 버전 2(0x02)에서만 지원됩니다. 메시지 형식 버전 2(0x02)에서 이 데이터는 메시지 헤더의 `Algorithm suite data` 필드에 표시됩니다. 키 커밋을 지원하는 알고리즘 제품군은 키 커밋 문자열에 32바이트를 사용합니다. 자세한 정보는 이 목록의 키 커밋 알고리즘을 참조하세요.

데이터 키 길이

데이터 키의 길이(비트)입니다. AWS Encryption SDK 는 256비트, 192비트, 128비트 키를 지원합니다. 데이터 키는 키링 또는 마스터 키로 생성됩니다.

일부 구현에서는 이 데이터 키가 HMAC 기반 추출 및 확장 키 유도 함수(HKDF)의 입력으로 사용됩니다. HKDF의 출력은 암호화 알고리즘에서 데이터 암호화 키로 사용됩니다. 자세한 정보는 이 목록의 키 유도 알고리즘을 참조하세요.

암호화 알고리즘

사용되는 암호화 알고리즘의 이름 및 모드입니다. AWS Encryption SDK 의 알고리즘 제품군은 Galois/Counter Mode(GCM)의 고급 암호화 표준(AES) 암호화 알고리즘을 사용합니다.

키 커밋 알고리즘

키 커밋 문자열을 계산하는 데 사용되는 알고리즘입니다. 출력은 메시지 헤더의 `Algorithm suite data` 필드에 저장되며 키 커밋에 대한 데이터 키를 검증하는 데 사용됩니다.

알고리즘 제품군에 키 커밋을 추가하는 방법에 대한 기술적인 설명은 Cryptology ePrint Archive의 키 커밋 AEAD를 참조하세요.

키 유도 알고리즘

데이터 암호화 키를 추출하는 데 사용되는 HMAC 기반 추출 및 확장 키 유도 함수(HKDF)입니다. 는 RFC 5869에 정의된 HKDF를 AWS Encryption SDK 사용합니다.

키 커밋이 없는 알고리즘 제품군(알고리즘 ID `01xx` - `03xx`)

- 사용되는 해시 함수는 알고리즘 제품군에 따라 SHA-384 또는 SHA-256 중 하나입니다.
- 추출 단계의 경우:
  - 솔트는 사용하지 않습니다. RFC에 따라 솔트는 0으로 구성된 문자열로 설정됩니다. 문자열 길이는 해시 함수 출력의 길이와 같으며, 이는 SHA-384의 경우 48바이트, SHA-256의 경우 32바이트입니다.
  - 입력 키 구성 요소는 키링 또는 마스터 키 공급자에서 받은 데이터 키입니다.
- 확장 단계의 경우:
  - 입력 의사 난수 키는 추출 단계의 출력입니다.
  - 입력 정보는 알고리즘 ID와 메시지 ID를 순서대로 연결한 것입니다.
  - 출력 키 구성 요소의 길이는 데이터 키 길이입니다. 이 출력은 암호화 알고리즘에서 데이터 암호화 키로 사용됩니다.

키 커밋이 있는 알고리즘 제품군(알고리즘 ID `04xx` 및 `05xx`)

- 사용되는 해시 함수는 SHA-512입니다.
- 추출 단계의 경우:
  - 솔트는 256비트 암호화 무작위 값입니다. 메시지 형식 버전 2(0x02)에서 이 값이 `MessageID` 필드에 저장됩니다.
  - 초기 키 구성 요소는 키링 또는 마스터 키 공급자에서 받은 데이터 키입니다.
- 확장 단계의 경우:
  - 입력 의사 난수 키는 추출 단계의 출력입니다.
  - 키 레이블은 빅 엔디안 바이트 순서로 UTF-8 인코딩된 바이트 `DERIVEKEY` 문자열입니다.

- 입력 정보는 알고리즘 ID와 키 레이블을 순서대로 연결한 것입니다.
- 출력 키 구성 요소의 길이는 데이터 키 길이입니다. 이 출력은 암호화 알고리즘에서 데이터 암호화 키로 사용됩니다.

메시지 형식 버전

알고리즘 제품군과 함께 사용되는 메시지 형식의 버전입니다. 자세한 내용은 메시지 형식 참조을 참조하세요.

서명 알고리즘

사이퍼텍스트 헤더 및 본문에 디지털 서명을 생성하는 데 사용되는 서명 알고리즘입니다. 는 다음 세부 정보와 함께 타원 곡선 디지털 서명 알고리즘(ECDSA)을 AWS Encryption SDK 사용합니다.

- 사용되는 타원 곡선은 알고리즘 ID로 지정된 P-384 또는 P-256 곡선입니다. 이러한 곡선은 DSS(디지털 서명 표준)(FIPS PUB 186-4)에 정의되어 있습니다.
- 사용되는 해시 함수는 SHA-384(P-384 곡선 사용) 또는 SHA-256(P-256 곡선 사용)입니다.

# AWS Encryption SDK 초기화 벡터 참조

AWS Encryption SDK와 호환되는 자체 암호화 라이브러리를 빌드할 때 이 페이지의 정보를 참조할 수 있습니다. 호환되는 자체 암호화 라이브러리를 빌드하는 경우가 아니라면 이 정보는 필요 없을 것입니다.

지원되는 프로그래밍 언어 중 하나 AWS Encryption SDK 에서를 사용하려면 섹션을 참조하세요프로그래밍 언어.

적절한 AWS Encryption SDK 구현의 요소를 정의하는 사양은 GitHub의 AWS Encryption SDK 사양을 참조하세요.

는 지원되는 모든 알고리즘 제품군에 필요한 초기화 벡터(IVs)를 AWS Encryption SDK 제공합니다. SDK는 프레임 시퀀스 번호를 사용하여 IV를 구성하므로 동일한 메시지의 두 프레임이 동일한 IV를 가질 수 없습니다.

각 96비트(12바이트) IV는 다음 순서로 연결된 두 개의 빅 엔디안 바이트 배열로 구성됩니다.

- 64비트: 0(향후 사용을 위해 예약됨)
- 32비트: 프레임 시퀀스 번호. 헤더 인증 태그의 경우 이 값은 모두 0입니다.

데이터 키 캐싱이 소개되기 전에는 AWS Encryption SDK 가 항상 새 데이터 키를 사용하여 각 메시지를 암호화하고 모든 IV를 임의로 생성했습니다. 무작위로 생성된 IV는 데이터 키가 재사용된 적이 없으므로 암호학적으로 안전했습니다. SDK가 의도적으로 데이터 키를 재사용하는 데이터 키 캐싱을 도입했을 때, SDK가 IV를 생성하는 방식을 변경했습니다.

메시지 내에서 반복할 수 없는 결정론적 IV를 사용하면 단일 데이터 키로 안전하게 실행할 수 있는 호출 수가 크게 늘어납니다. 또한 캐시된 데이터 키는 항상 키 유도 함수가 있는 알고리즘 제품군을 사용합니다. 의사 무작위 키 유도 함수와 함께 결정적 IV를 사용하여 데이터 키에서 암호화 키를 추출하면 가 암호화 범위를 초과하지 않고 2^32 메시지를 암호화 AWS Encryption SDK 할 수 있습니다.

## AWS KMS 계층적 키링 기술 세부 정보

AWS KMS 계층적 키링은 고유하지 않은 데이터 키를 사용하여 각 메시지를 암호화하고 활성 브랜치 키에서 파생된 고유한 래핑 키로 각 데이터 키를 암호화합니다. 이 키링은 HMAC SHA-256으로 의사 난수 함수를 통해 카운터 모드에서 키 유도를 사용하여 다음 입력으로 32바이트 래핑 키를 도출합니다.

- 16바이트 무작위 솔트
- 활성 브랜치 키
- 키 공급자 식별자 "aws-kms-hierarchy"의 UTF-8 인코딩된 값

계층적 키링은 파생된 래핑 키를 사용하여 16바이트 인증 태그 및 다음 입력과 함께 AES-GCM-256을 사용하여 일반 텍스트 데이터 키의 사본을 암호화합니다.

- 파생된 래핑 키는 AES-GCM 암호 키로 사용됩니다
- 데이터 키는 AES-GCM 메시지로 사용됩니다
- 12바이트 무작위 초기화 벡터(IV)는 AES-GCM IV로 사용됩니다
- 다음과 같은 직렬화된 값을 포함하는 추가 인증 데이터(AAD)

| 값 | 길이(바이트) | 다음으로 해석됨 |
|---|---|---|
| "aws-kms-hierarchy" | 17 | UTF-8 인코딩 |
| 브랜치 키 식별자 | 변수 | UTF-8 인코딩 |
| 브랜치 키 버전 | 16 | UTF-8 인코딩 |
| 암호화 컨텍스트 | 변수 | UTF-8 인코딩 키-값 페어 |

# AWS Encryption SDK 개발자 안내서의 문서 기록

이 주제에서는 AWS Encryption SDK 개발자 가이드에 대한 중요한 업데이트 사항에 대해 설명합니다.

주제



## 최신 업데이트

다음 표에서는 2017년 11월 이후 이 설명서의 중요한 변경 사항을 설명합니다. Amazon은 여기 나와 있는 주요 변경 사항 외에도 설명과 예제를 업데이트하고 고객이 제공한 피드백을 반영하도록 설명서를 자주 업데이트하고 있습니다. 중요한 변경 사항에 대해 알림을 받으려면 RSS 피드를 구독합니다.

| 변경 사항 | 설명 | 날짜 |
| --- | --- | --- |
| 정식 출시 | AWS KMS ECDH 키링 및 원시 ECDH 키링에 대한 설명서가 추가되었습니다. | 2024년 6월 17일 |
| AWS Encryption SDK for Java 버전 3.x | 를 재료 공급자 라이브러리 AWS Encryption SDK for Java 와 통합합니다. 키링 및 필요한 암호화 컨텍스트 CMM에 대한 지원을 추가합니다. | 2023년 12월 6일 |
| AWS Encryption SDK for .NET 버전 4.x | AWS KMS 계층적 키링, 필요한 암호화 컨텍스트 CMM 및 비대칭 RSA AWS KMS 키링에 대한 지원을 추가합니다. | 2023년 10월 12일 |
| 정식 출시 | for .NET에 AWS Encryption SDK 대한 지원을 소개합니다. | 2022년 5월 17일 |
| 문서 변경 | 고객 마스터 키(CMK)라는 AWS Key Management | 2021년 8월 30일 |

| | | |
|---|---|---|
| | Service 용어를 AWS KMS key 및 KMS 키로 바꿉니다. | |
| 정식 출시 | 다중 리전 키에 대한 지원이 추가되었습니다 AWS Key Management Service.(AWS KMS) 다중 리전 키는 AWS KMS 키 ID와 키 구성 요소가 동일하기 때문에 서로 바꿔서 사용할 수 AWS 리전 있는 서로 다른의 키입니다. | 2021년 6월 8일 |
| 정식 출시 | 개선된 메시지 복호화 프로세스에 대한 설명서가 추가 및 업데이트되었습니다. | 2021년 5월 11일 |
| 정식 출시 | AWS Encryption CLI 버전 1.7.x를 대체하는 AWS Encryption CLI 버전 1.8.x 및 AWS Encryption CLI 2.0.x를 대체하는 AWS Encryption CLI 2.1.x의 정식 릴리스에 대한 설명서를 추가하고 업데이트했습니다. | 2020년 10월 27일 |
| 정식 출시 | 모범 사례 가이드, 마이그레이션 가이드, 업데이트된 개념, 업데이트된 프로그래밍 언어 주제, 업데이트된 알고리즘 제품군 참조, 업데이트된 메시지 형식 참조, 새 메시지 형식 예제가 포함된 AWS Encryption SDK 버전 1.7.x 및 2.0.x의 정식 출시 릴리스에 대한 설명서가 추가 및 업데이트되었습니다. | 2020년 9월 24일 |

| | | |
|---|---|---|
| 정식 출시 | AWS Encryption SDK for JavaScript의 정식 출시 릴리스에 대한 설명서가 추가 및 업데이트되었습니다. | 2019년 10월 1일 |
| 미리 보기 릴리스 | AWS Encryption SDK for JavaScript의 공개 베타 릴리스에 대한 설명서가 추가 및 업데이트되었습니다. | 2019년 6월 21일 |
| 정식 출시 | AWS Encryption SDK for C의 정식 출시 릴리스에 대한 설명서가 추가 및 업데이트되었습니다. | 2019년 5월 16일 |
| 미리 보기 릴리스 | AWS Encryption SDK for C의 미리 보기 릴리스에 대한 설명서가 추가되었습니다. | 2019년 2월 5일 |
| 새로운 릴리스 | AWS Encryption SDK에 대한 명령줄 인터페이스 설명서가 추가되었습니다. | 2017년 11월 20일 |

# 이전 업데이트

다음 표에서는 2017년 11월 이전 AWS Encryption SDK 개발자 가이드에서 변경된 중요 사항에 대해 설명합니다.

| 변경 | 설명 | Date |
|---|---|---|
| 새로운 릴리스 | 새 기능에 대한 데이터 키 캐싱 챕터가 추가되었습니다.<br><br>SDK가 무작위 IV 생성에서 결정적 IV 구성으로 변경되었음을 설명하는 the section called | 2017년 7월 31일 |

| 변경 | 설명 | Date |
| --- | --- | --- |
| | "초기화 벡터 참조" 주제가 추가되었습니다.<br><br>새로운 암호화 자료 관리자를 비롯한 개념을 설명하는 the section called "개념" 주제가 추가되었습니다. | |
| 업데이트 | 메시지 형식 참조 설명서가 새 AWS Encryption SDK 참조 섹션으로 확장되었습니다.<br><br>에 대한 섹션을 추가했습니다 AWS Encryption SDK 지원 알고리즘 제품군. | 2017년 3월 21일 |
| 새로운 릴리스 | AWS Encryption SDK 이제는 외에도 Python 프로그래밍 언어를 지원합니다Java. | 2017년 3월 21일 |
| 초기 릴리스 | AWS Encryption SDK 및이 설명서의 최초 릴리스. | 2016년 3월 22일 |

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.