aws

개발자 가이드

AWS 데이터베이스 암호화 SDK

AWS 데이터베이스 암호화 SDK: 개발자 가이드

Table of Contents

AWS Database Encryption SDK란 무엇입니까?

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK는 데이터베이스 설계에 클라이언트 측 암호화를 포함할 수 있는 소프트웨어 라이브러리 세트입니다. AWS Database Encryption SDK는 레코드 수준 암호화 솔루션을 제공합니다. 암호화할 필드와 데이터의 신뢰성을 보장하는 서명에 포함할 필드를 지정합니다. 전송 중 및 유휴 상태의 중요 데이터를 암호화하면 일반 텍스트 데이터를 AWS을 포함한 제3자가 사용할 수 없게 하는 데 도움이 됩니다. AWS Database Encryption SDK는 Apache 2.0 라이선스에 따라 무료 제공됩니다.

이 개발자 안내서에서는 아키텍처 소개, 데이터 보호 방법에 대한 세부 정보, 서버 측 암호화와 데이터의 차이점, 시작하는 데 도움이 되는 애플리케이션의 중요 구성 요소 선택 지침 등 AWS Database Encryption SDK의 개념적 개요를 제공합니다.

AWS Database Encryption SDK는 속성 수준 암호화를 통해 Amazon DynamoDB를 지원합니다.

AWS Database Encryption SDK에는 다음과 같은 이점이 있습니다.

데이터베이스 애플리케이션을 위해 특별히 설계됨

AWS Database Encryption SDK를 사용하기 위해 암호화 전문가가 될 필요는 없습니다. 구현에는 기존 애플리케이션과 함께 작동하도록 설계된 헬퍼 방법이 포함됩니다.

필수 구성 요소를 생성 및 구성한 후, 암호화 클라이언트는 사용자가 데이터베이스에 레코드를 추가할 때 레코드를 투명하게 암호화 및 서명하고, 검색할 때 이를 확인하고 복호화합니다.

보안 암호화 및 서명 포함

AWS Database Encryption SDK에는 고유한 데이터 암호화 키를 사용하여 각 레코드의 필드 값을 암호화한 다음 레코드에 서명하여 필드를 추가 또는 삭제하거나 암호화된 값을 교체하는 등 무단 변경으로부터 보호하는 보안 구현이 포함되어 있습니다.

모든 소스의 암호화 자료 사용

AWS Database Encryption SDK는 키링을 사용하여 레코드를 보호하는 고유한 데이터 암호화 키를 생성, 암호화 및 해독합니다. 키링은 해당 데이터 키를 암호화하는 래핑 키를 결정합니다.

AWS Key Management Service(AWS KMS) 또는 AWS CloudHSM와 같은 암호화 서비스를 포함하여 모든 소스의 래핑 키를 사용할 수 있습니다. AWS Database Encryption SDK에는 AWS 계정 또는 서비스가 AWS 필요하지 않습니다.

암호화 자료 캐싱 지원

AWS KMS 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. 레코드를 암호화하거나 복호화할 AWS KMS 때마다 를 호출하지 않고도 대칭 암호화 KMS 키로 암호화 자료를 보호할 수 있습니다. AWS KMS 계층적 키링은 호출을 최소화해야 하는 애플리케이션에 적합합니다 AWS KMS.

검색 가능한 암호화

전체 데이터베이스를 복호화하지 않고도 암호화된 레코드를 검색할 수 있는 데이터베이스를 설계할 수 있습니다. 위협 모델 및 쿼리 요구 사항에 따라 검색 가능한 암호화를 사용하여 암호화된 데이터베이스에 대해 정확한 일치 검색 또는 보다 사용자 정의된 복잡한 쿼리를 수행할 수 있습니다.

멀티테넌트 데이터베이스 스키마 지원

AWS Database Encryption SDK를 사용하면 각 테넌트를 고유한 암호화 자료로 격리하여 공유 스키마를 사용하여 데이터베이스에 저장된 데이터를 보호할 수 있습니다. 데이터베이스 내에서 암호화 작업을 수행하는 사용자가 여러 명 있는 경우 AWS KMS 키링 중 하나를 사용하여 각 사용자에게 암호화 작업에 사용할 고유한 키를 제공합니다. 자세한 내용은 멀티테넌트 데이터베이스 작업 단원을 참조하십시오.

원활한 스키마 업데이트 지원

AWS Database Encryption SDK를 구성할 때 암호화 및 서명할 필드, 서명할 필드(암호화하지 않음), 무시할 필드를 클라이언트에 알려주는 암호화 작업을 제공합니다. AWS Database Encryption SDK를 사용하여 레코드를 보호한 후에도 데이터 모델을 변경할 수 있습니다. 단일 배포에서 암호화된 필드 추가 또는 제거와 같은 암호화 작업을 업데이트할 수 있습니다.

오픈 소스 리포지토리에서 개발

AWS Database Encryption SDK는 GitHub의 오픈 소스 리포지토리에서 개발됩니다. 이러한 리포지토리를 사용하여 코드를 보고, 문제를 읽고 제출하고, 구현과 관련된 정보를 찾을 수 있습니다.

DynamoDB용 AWS Database Encryption SDK

- GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리는 Java, .NET 및 Rust에서 DynamoDB용 AWS Database Encryption SDK의 최신 버전을 지원합니다.

AWS Database Encryption SDK for DynamoDB는 사양을 작성하는 확인 인식 언어인 Dafny의 제품이며, 이를 구현할 코드와 이를 테스트하기 위한 증명입니다. 그 결과 기능적 정확성을 보장하는 프레임워크에서 AWS Database Encryption SDK for DynamoDB의 기능을 구현하는 라이브러리가 탄생했습니다.

지원 및 유지 관리

AWS Database Encryption SDK는 버전 관리 및 수명 주기 단계를 포함하여 AWS SDK 및 도구가 사용하는 것과 동일한 유지 관리 정책을 사용합니다. 데이터베이스 구현을 위해 사용 가능한 최신 버전의 AWS Database Encryption SDK를 사용하고 새 버전이 출시되면 업그레이드하는 것이 모범 사례입니다.

자세한 내용은 SDK AWS 및 도구 참조 안내서의 SDKs 및 도구 유지 관리 정책을 참조 AWS SDKs.

피드백 보내기

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

AWS Database Encryption SDK에서 잠재적 보안 취약성을 발견한 경우 AWS 보안 팀에 알리세요. 공개적으로 GitHub 문제를 작성하지 마세요.

이 설명서에 대한 피드백을 제공하려면 이 페이지의 피드백 링크를 사용하십시오.

AWS Database Encryption SDK 개념

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

이 주제에서는 AWS Database Encryption SDK에 사용되는 개념과 용어를 설명합니다.

AWS Database Encryption SDK의 구성 요소가 상호 작용하는 방법을 알아보려면 섹션을 참조하세요AWS Database Encryption SDK 작동 방식.

AWS Database Encryption SDK에 대해 자세히 알아보려면 다음 주제를 참조하세요.

- AWS Database Encryption SDK가 봉투 암호화를 사용하여 데이터를 보호하는 방법을 알아봅니다.

- 엔빌로프 암호화의 구성 요소인 레코드를 보호하는 데이터 키와 데이터 키를 보호하는 래핑 키에 대해 알아봅니다.
- 사용하는 래핑 키를 결정하는 키링에 대해 알아봅니다.
- 암호화 프로세스에 무결성을 더하는 암호화 컨텍스트에 대해 알아봅니다.
- 암호화 메서드가 레코드에 추가하는 자료 설명에 대해 알아봅니다.
- AWS Database Encryption SDK에 암호화하고 서명할 필드를 알려주는 암호화 작업에 대해 알아봅니다.

주제

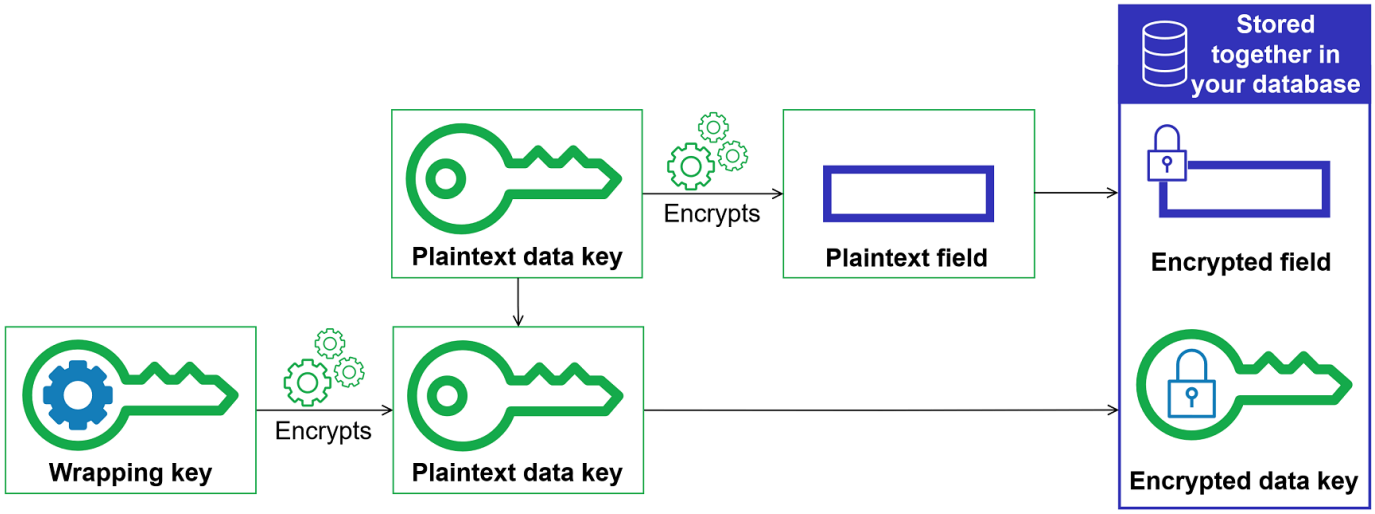
봉투 암호화

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

AWS Database Encryption SDK는 고유한 데이터 키로 각 필드를 암호화합니다. 그러면 지정한 래핑 키에서 각 데이터 키가 암호화됩니다. 암호화된 데이터 키를 자료 설명에 저장합니다.

래핑 키를 지정하려면 키링을 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. AWS Database Encryption SDK는 자료 설명의 암호화된 필드와 함께 모든 암호화된 데이터 키를 저장합니다.

데이터를 복호화하려면 암호화된 데이터 키를 복호화할 수 있는 래핑 키를 적어도 한 개 제공해야 합니다.

여러 알고리즘의 강점 결합

기본적으로 AWS Database Encryption SDK는 AES-GCM 대칭 암호화, HMAC 기반 키 유도 함수(HKDF) 및 ECDSA 서명이 포함된 알고리즘 제품군을 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 제공합니다. 각각의 장점을 결합하려면 퍼블릭 키 암호화를 사용하여 데이터 키를 암호화할 수 있습니다.

가능하면 AWS KMS 키링 중 하나를 사용하는 것이 좋습니다. AWS KMS 키링을 사용할 때 비대칭 RSA를 래핑 키 AWS KMS key 로 지정하여 여러 알고리즘의 강도를 결합하도록 선택할 수 있습니다. 대칭 암호화 KMS 키를 사용할 수도 있습니다.

데이터 키

데이터 키는 AWS Database Encryption SDK가 암호화 작업에 표시된 레코드의 필드를 암호화하는 데 사용하는 `ENCRYPT_AND_SIGN` 암호화 키입니다. 각 데이터 키는 암호화 키 요구 사항을 준수하는 바

이트 배열입니다. AWS Database Encryption SDK는 고유한 데이터 키를 사용하여 각 속성을 암호화합니다.

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

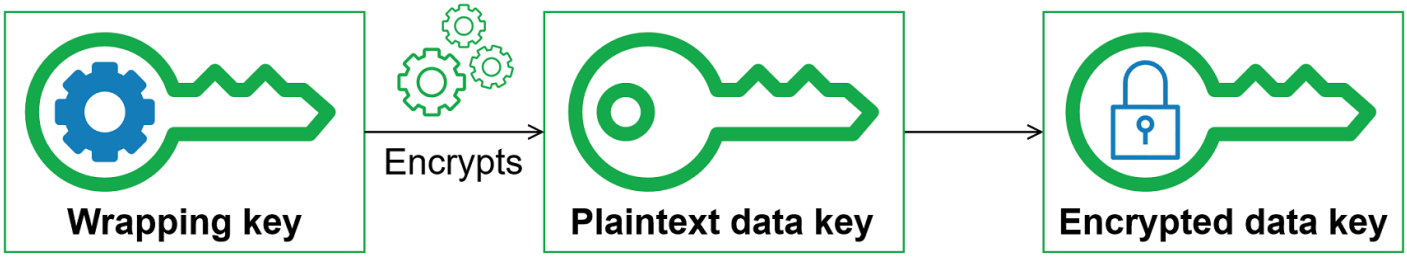
데이터 키를 보호하기 위해 AWS Database Encryption SDK는 래핑 키라고 하는 하나 이상의 키 암호화 키로 암호화합니다. AWS Database Encryption SDK는 일반 텍스트 데이터 키를 사용하여 데이터를 암호화한 후 가능한 한 빨리 메모리에서 제거합니다. 그런 다음 암호화된 데이터 키를 자료 설명에 저장합니다. 자세한 내용은 AWS Database Encryption SDK 작동 방식을 참조하세요.

> **Tip**
>
> AWS Database Encryption SDK에서는 데이터 키와 데이터 암호화 키를 구분합니다. 지원되는 모든 알고리즘 제품군은 키 파생 함수를 사용하는 것이 가장 좋습니다. 키 파생 함수는 데이터 키를 입력으로 사용하고 레코드를 암호화하는 데 실제로 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 메타데이터를 사용하면 복호화 시 AWS Database Encryption SDK가 유효한 래핑 키를 식별할 수 있습니다.

래핑 키

래핑 키는 AWS Database Encryption SDK가 레코드를 암호화하는 데이터 키를 암호화하는 데 사용하는 키-암호화 키입니다. 각각의 데이터 키는 한 개 또는 여러 개의 래핑 키로 암호화될 수 있습니다. 키링을 구성할 때 데이터를 보호하기 위해 사용할 래핑 키를 결정합니다.



AWS Database Encryption SDK는 AWS Key Management Service (AWS KMS) 대칭 암호화 KMS 키(다중 리전 키 포함 AWS KMS) 및 비대칭 RSA KMS 키, 원시 AES-GCM(고급 암호화 표준/갈루아 카

운터 모드) 키, 원시 RSA 키와 같이 일반적으로 사용되는 여러 래핑 키를 지원합니다. 가능하면 KMS 키를 사용하는 것이 좋습니다. 사용해야 하는 래핑 키를 결정하려면 래핑 키 선택을 참조하세요.

엔빌로프 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 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. AWS Database Encryption SDK는 와 통합되어 래핑 키를 보호하고 사용하는 AWS KMS 데 도움이 됩니다.

키링

암호화와 복호화에 사용하는 래핑 키를 지정하려면 키링을 사용합니다. AWS Database Encryption SDK가 제공하는 키링을 사용하거나 자체 구현을 설계할 수 있습니다.

키링은 데이터 키를 생성, 암호화, 복호화합니다. 또한 서명의 해시 기반 메시지 인증 코드(HMAC)를 계산하는 데 사용되는 MAC 키도 생성합니다. 키링을 정의할 때 데이터 키를 암호화하는 래핑 키를 지정할 수 있습니다. 대부분의 키링은 하나 이상의 래핑 키를 지정하거나, 래핑 키를 제공하고 보호하는 서비스를 지정합니다. 암호화할 때 AWS Database Encryption SDK는 키링에 지정된 모든 래핑 키를 사용하여 데이터 키를 암호화합니다. AWS Database Encryption SDK에서 정의하는 키링을 선택하고 사용하는 방법에 대한 도움말은 키링 사용을 참조하세요.

암호화 작업

암호화 작업은 레코드의 각 필드에 수행할 작업을 암호화기에 지시합니다.

암호화 작업 값은 다음 중 하나일 수 있습니다.

- 암호화 및 서명 - 필드를 암호화합니다. 암호화된 필드를 서명에 포함합니다.
- 서명 전용 - 서명에 필드를 포함합니다.
- 암호화 컨텍스트에 서명 및 포함 - 서명 및 암호화 컨텍스트에 필드를 포함합니다.

  기본적으로 파티션 및 정렬 키는 암호화 컨텍스트에 포함된 유일한 속성입니다. AWS KMS 계층적 키링의 브랜치 키 ID 공급자가 암호화 컨텍스트에서 복호화하는 데 필요한 브랜치 키를 식별할 수 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 있도록 추가 필드를 로 정의하는 것이 좋습니다. 자세한 내용은 브랜치 키 ID 공급자를 참조하세요.

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 암호화 작업을 사용하려면 AWS Database Encryption SDK 버전 3.3 이상을 사용해야 합니다. 를 포함하도록 데이터 모델을 업데이트하기 전에 모든 리더에 새 버전을 배포합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

- 아무것도 하지 않음 - 필드를 암호화하거나 서명에 포함하지 않습니다.

중요한 데이터를 저장할 수 있는 필드의 경우 암호화 및 서명을 사용합니다. 기본 키 값(예: DynamoDB 테이블의 파티션 키 및 정렬 키)의 경우 서명 전용 또는 서명을 사용하고 암호화 컨텍스트에 포함합니다. 서명을 지정하고 암호화 컨텍스트 속성에를 포함하는 경우 파티션 및 정렬 속성도 서명이어야 하며 암호화 컨텍스트에를 포함해야 합니다. 자료 설명에는 암호화 작업을 지정할 필요가 없습니다. AWS Database Encryption SDK는 자료 설명이 저장된 필드에 자동으로 서명합니다.

암호화 작업을 신중하게 선택합니다. 확실하지 않은 경우 Encrypt and sign(암호화 및 서명)을 사용합니다. AWS Database Encryption SDK를 사용하여 레코드를 보호한 후에는 기존 , ENCRYPT_AND_SIGN SIGN_ONLY또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 필드를 로 변경하거나 기존 DO_NOTHING 필드에 할당된 암호화 작업을 DO_NOTHING변경할 수 없습니다. 하지만 여전히 데이터 모델에 다른 변경 사항을 적용할 수 있습니다. 예를 들어 단일 배포에서 암호화된 필드를 추가하거나 제거할 수 있습니다.

자료 설명

자료 설명은 암호화된 레코드의 헤더 역할을 합니다. AWS Database Encryption SDK로 필드를 암호화하고 서명하면 암호화 도구는 암호화 자료를 어셈블할 때 자료 설명을 기록하고 암호화 도구가 레코드에 추가하는 새 필드(aws_dbe_head)에 자료 설명을 저장합니다.

자료 설명은 데이터 키의 암호화된 사본과 기타 정보(예: 암호화 알고리즘, 암호화 컨텍스트, 암호화 및 서명 지침)를 포함하는 휴대용 형식의 데이터 구조입니다. 암호화 도구 레코드는 암호화 및 서명을 위해 암호화 자료를 결합할 때 자료 설명을 기록합니다. 나중에 필드를 확인하고 복호화하기 위해 암호화 자료를 조합해야 하는 경우 자료 설명을 지침으로 사용합니다.

암호화된 데이터 키를 암호화된 필드와 함께 저장하면 복호화 작업이 간소화되고 암호화된 데이터와 별도로 암호화된 데이터 키를 저장하고 관리할 필요가 없습니다.

자료 설명에 대한 기술 정보는 자료 설명 형식을 참조하세요.

암호화 컨텍스트

암호화 작업의 보안을 개선하기 위해 AWS Database Encryption SDK는 레코드를 암호화하고 서명하기 위한 모든 요청에 암호화 컨텍스트를 포함합니다.

암호화 컨텍스트는 비밀이 아닌 임의의 추가 인증 데이터를 포함하는 키-값 페어 세트입니다. AWS Database Encryption SDK에는 데이터베이스의 논리적 이름과 암호화 컨텍스트의 기본 키 값(예: DynamoDB 테이블의 파티션 키 및 정렬 키)이 포함됩니다. 필드를 암호화하고 서명하면 암호화 컨텍스트가 암호화된 레코드에 암호화 방식으로 바인딩되므로 필드를 복호화하는 데 동일한 암호화 컨텍스트가 필요합니다.

AWS KMS 키링을 사용하는 경우 AWS Database Encryption SDK는 암호화 컨텍스트를 사용하여 키링이 수행하는 호출에 추가 인증 데이터(AAD)를 제공합니다 AWS KMS.

기본 알고리즘 제품군을 사용할 때마다 암호화 자료 관리자(CMM)은 예약된 이름 `aws-crypto-public-key`과 퍼블릭 확인 키를 나타내는 값으로 구성된 암호화 컨텍스트에 이름-값 페어를 추가합니다. 퍼블릭 확인 키는 자료 설명에 저장됩니다.

암호화 구성 요소 관리자

암호화 자료 관리자(CMM)는 데이터를 암호화, 복호화 및 서명하는 데 사용되는 암호화 자료를 수집합니다. 기본 알고리즘 제품군을 사용할 때마다 암호화 자료에는 일반 텍스트 및 암호화된 데이터 키, 대칭 서명 키, 비대칭 서명 키가 포함됩니다. 사용자는 CMM과 직접 상호 작용하지는 않습니다. 암호화 및 복호화 메서드가 이를 대신 처리합니다.

CMM은 AWS Database Encryption SDK와 키링 간의 연결 역할을 하기 때문에 정책 적용 지원과 같은 사용자 지정 및 확장을 위한 이상적인 지점입니다. CMM을 명시적으로 지정할 수 있지만 필수는 아닙니다. 키링을 지정하면 AWS Database Encryption SDK가 기본 CMM을 생성합니다. 기본 CMM은 사용자가 지정한 키링으로부터 암호화 또는 복호화 자료를 가져옵니다. 여기에는 AWS Key Management Service(AWS KMS)와 같은 암호화 서비스에 대한 호출이 포함될 수 있습니다.

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.

AWS Database Encryption SDK는 봉투 암호화를 사용합니다. 대칭 데이터 키로 데이터를 암호화합니다. 하나 이상의 대칭 또는 비대칭 래핑 키를 사용하여 대칭 데이터 키를 암호화합니다. 데이터 키의 암호화된 사본을 하나 이상 포함하는 자료 설명을 레코드에 추가합니다.

데이터 암호화(대칭 암호화)

AWS Database Encryption SDK는 대칭 데이터 키와 대칭 암호화 알고리즘이 포함된 알고리즘 제품군을 사용하여 데이터를 암호화합니다. AWS Database Encryption SDK는 데이터를 해독하기 위해 동일한 데이터 키와 동일한 알고리즘 제품군을 사용합니다.

데이터 키 암호화(대칭 또는 비대칭 암호화)

암호화 및 복호화 작업에 제공하는 키링에 따라 대칭 데이터 키가 암호화 및 복호화되는 방식이 결정됩니다. 대칭 암호화 KMS 키가 있는 AWS KMS 키링과 같이 대칭 암호화를 사용하는 키링 또는 비대칭 RSA KMS 키가 있는 키링과 같이 AWS KMS 비대칭 암호화를 사용하는 키링을 선택할 수 있습니다.

키 커밋

AWS Database Encryption SDK는 각 사이퍼텍스트를 단일 일반 텍스트로만 해독할 수 있도록 하는 보안 속성인 키 커밋(강력성이라고도 함)을 지원합니다. 이를 위해 키 커밋은 레코드를 암호화한 데이터 키만 복호화에 사용되도록 보장합니다. AWS Database Encryption SDK에는 모든 암호화 및 복호화 작업에 대한 키 커밋이 포함되어 있습니다.

대부분의 최신 대칭 암호(AES 포함)는 AWS Database Encryption SDK가 레코드`ENCRYPT_AND_SIGN`에 표시된 각 일반 텍스트 필드를 암호화하는 데 사용하는 고유한 데이터 키와 같이 단일 보안 키로 일반 텍스트를 암호화합니다. 동일한 데이터 키로 이 레코드를 복호화하면 원본과 동일한 일반 텍스트가 반환됩니다. 다른 키를 사용한 복호화는 일반적으로 실패합니다. 어렵기는 하지만 두 개의 서로 다른 키로 사이퍼텍스트를 복호화하는 것은 기술적으로 가능합니다. 드문 경우지만, 사이퍼텍스트를 부분적으로 복호화할 수 있는 다르지만 여전히 식별할 수 있는 일반 텍스트로 복호화할 수 있는 키를 찾는 것이 가능합니다.

AWS Database Encryption SDK는 항상 하나의 고유한 데이터 키로 각 속성을 암호화합니다. 여러 래핑 키로 해당 데이터 키를 암호화할 수 있지만 래핑 키는 항상 동일한 데이터 키를 암호화합니다. 하지만 정교하고 수동으로 조작된 암호화된 레코드에는 실제로 각각 다른 래핑 키로 암호화된 서로 다른 데이터 키가 포함될 수 있습니다. 예를 들어 한 사용자가 암호화된 레코드를 복호화하여 0x0(false)이 반환됐지만 동일한 암호화된 레코드를 다른 사용자가 복호화하면 0x1(true)이 반환될 수 있습니다.

이 시나리오를 방지하기 위해 AWS Database Encryption SDK에는 암호화 및 복호화 시 키 커밋이 포함됩니다. 암호화 메서드는 사이퍼텍스트를 생성한 고유 데이터 키를 키 커밋, 즉 데이터 키의 파생물을 사용하여 자료 설명에 대해 계산된 해시 기반 메시지 인증 코드(HMAC)에 암호화 방식으로 바인딩합니다. 그런 다음 자료 설명에 키 커밋을 저장합니다. 키 커밋으로 레코드를 복호화하면 AWS

Database Encryption SDK는 데이터 키가 암호화된 레코드의 유일한 키인지 확인합니다. 데이터 키 확인에 실패하면 복호화 작업이 실패합니다.

디지털 서명

AWS Database Encryption SDK는 인증된 암호화 알고리즘, 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 위해 기본 알고리즘 제품군은 타원 곡선 디지털 서명 알고리즘(ECDSA) 서명을 암호화된 레코드에 추가합니다. 기본 알고리즘 제품군은 인증된 암호화 알고리즘인 AES-GCM을 사용하여 `ENCRYPT_AND_SIGN`로 표시된 레코드의 필드를 암호화합니다. 그런 다음 , 및 로 표시된 레코드의 필드를 통해 해시 기반 메시지 인증 코드(HMACs)`ENCRYPT_AND_SIGN``SIGN_ONLY`와 비대칭 ECDSA 서명을 모두 계산합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. 복호화 프로세스는 서명을 사용하여 인증된 사용자가 레코드를 암호화했는지 확인합니다.

기본 알고리즘 제품군을 사용하면 AWS Database Encryption SDK는 암호화된 각 레코드에 대해 임시 프라이빗 키와 퍼블릭 키 페어를 생성합니다. AWS Database Encryption SDK는 퍼블릭 키를 자료 설명에 저장하고 프라이빗 키를 삭제합니다. 이렇게 하면 아무도 퍼블릭 키로 확인하는 다른 서명을 생성할 수 없습니다. 알고리즘은 퍼블릭 키를 자료 설명의 추가 인증 데이터로 암호화된 데이터 키에 바인딩하여 필드를 복호화할 수 있는 사용자가 퍼블릭 키를 변경하거나 서명 확인에 영향을 미치지 않도록 합니다.

AWS Database Encryption SDK에는 항상 HMAC 확인이 포함됩니다. ECDSA 디지털 서명은 기본적으로 활성화되지만 필수는 아닙니다. 데이터를 암호화하는 사용자와 데이터를 복호화하는 사용자가 동일하게 신뢰되는 경우 디지털 서명이 포함되지 않은 알고리즘 제품군을 사용하여 성능을 향상시키는 것을 고려할 수 있습니다. 대체 알고리즘 제품군 선택에 대한 자세한 내용은 알고리즘 제품군 선택을 참조하세요.

> **Note**
>
> 키링이 암호화 도구와 암호 해독기 사이를 구분하지 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 Database Encryption SDK 작동 방식

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK는 데이터베이스에 저장하는 데이터를 보호하도록 특별히 설계된 클라이언트 측 암호화 라이브러리를 제공합니다. 라이브러리에는 변경 없이 확장하거나 사용할 수 있는 보안 구현이 포함되어 있습니다. 사용자 정의 구성 요소 정의 및 사용에 대한 자세한 내용은 데이터베이스 구현을 위한 GitHub 리포지토리를 참조하세요.

이 섹션의 워크플로에서는 AWS Database Encryption SDK가 데이터베이스의 데이터를 암호화 및 서명하고 해독하는 방법을 설명합니다. 이러한 워크플로는 추상 요소와 기본 기능을 사용하여 기본 프로세스를 설명합니다. AWS Database Encryption SDK가 데이터베이스 구현과 작동하는 방식에 대한 자세한 내용은 데이터베이스의 암호화된 항목 주제를 참조하세요.

AWS Database Encryption SDK는 봉투 암호화를 사용하여 데이터를 보호합니다. 각 레코드는 고유한 데이터 키로 암호화됩니다. 데이터 키는 암호화 작업에 `ENCRYPT_AND_SIGN`으로 표시된 각 필드에 대해 고유한 데이터 암호화 키를 파생하는 데 사용됩니다. 그런 다음 데이터 키의 복사본은 지정한 래핑 키로 암호화됩니다. 암호화된 레코드를 해독하기 위해 AWS Database Encryption SDK는 지정한 래핑 키를 사용하여 하나 이상의 암호화된 데이터 키를 해독합니다. 그런 다음 사이버텍스트를 복호화하고 일반 텍스트 항목을 반환할 수 있습니다.

AWS Database Encryption SDK에 사용되는 용어에 대한 자세한 내용은 섹션을 참조하세요AWS Database Encryption SDK 개념.

암호화 및 서명

기본적으로 AWS Database Encryption SDK는 데이터베이스의 레코드를 암호화, 서명, 확인 및 해독하는 레코드 암호화 도구입니다. 암호화하고 서명할 필드에 대한 지침과 레코드에 대한 정보를 가져옵니다. 지정한 래핑 키로 구성된 암호화 자료 관리자로부터 암호화 자료와 사용 방법에 대한 지침을 가져옵니다.

다음 연습에서는 AWS Database Encryption SDK가 데이터 항목을 암호화하고 서명하는 방법을 설명합니다.

1. 암호화 자료 관리자는 AWS 데이터베이스 암호화 SDK에 일반 텍스트 데이터 키 1개, 지정된 래핑 키로 암호화된 데이터 키의 사본, MAC 키와 같은 고유한 데이터 암호화 키를 제공합니다.

   > **Note**
   >
   > 여러 개의 래핑 키로 데이터 키를 암호화할 수 있습니다. 각 래핑 키는 데이터 키의 별도 복사본을 암호화합니다. AWS Database Encryption SDK는 암호화된 모든 데이터 키를 자료 설명에 저장합니다. AWS Database Encryption SDK는 자료 설명을 저장하는 레코드에 새 필드(`aws_dbe_head`)를 추가합니다.
   >
   > 데이터 키의 암호화된 각 복사본에 대해 MAC 키가 파생됩니다. MAC 키는 자료 설명에 저장되지 않습니다. 대신, 복호화 메서드는 래핑 키를 사용하여 MAC 키를 다시 파생시킵니다.

2. 암호화 메서드는 지정한 암호화 작업에서 `ENCRYPT_AND_SIGN`으로 표시된 각 필드를 암호화합니다.
3. 암호화 메서드는 데이터 키에서 `commitKey`을 파생한 다음 이를 사용하여 키 커밋 값을 생성한 다음 데이터 키를 삭제합니다.
4. 암호화 메서드는 레코드에 자료 설명을 추가합니다. 자료 설명에는 암호화된 데이터 키와 암호화된 레코드에 대한 기타 정보가 포함되어 있습니다. 자료 설명에 포함된 정보의 전체 목록은 자료 설명 형식을 참조하세요.
5. 암호화 방법은 1단계에서 반환된 MAC 키를 사용하여 암호화 작업`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`에서 자료 설명, 암호화 컨텍스트 및 `ENCRYPT_AND_SIGN`, `SIGN_ONLY`또는 로 표시된 각 필드의 표준화를 통해 해시 기반 메시지 인증 코드(HMAC) 값을 계산합니다. HMAC 값은 암호화 메서드가 레코드에 추가하는 새 필드(`aws_dbe_foot`)에 저장됩니다.
6. 암호화 방법은 자료 설명, 암호화 컨텍스트 및 , `SIGN_ONLY`또는 `ENCRYPT_AND_SIGN`로 표시된 각 필드의 표준화를 통해 ECDSA 서명을 계

산SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT하고 필드에 ECDSA 서명을 저장합니다aws_dbe_foot.

> **Note**
>
> ECDSA 서명은 기본적으로 활성화되어 있지만 필수는 아닙니다.

7. 암호화 메서드는 암호화되고 서명된 레코드를 데이터베이스에 저장합니다

복호화 및 확인

1. 암호 자료 관리자(CMM)는 일반 텍스트 데이터 키 및 관련 MAC 키를 포함하여 자료 설명에 저장된 복호화 자료를 사용하여 복호화 메서드를 제공합니다.
   - CMM은 지정된 키 링의 래핑 키를 사용하여 암호화된 데이터 키를 복호화하고 일반 텍스트 데이터 키를 반환합니다.
2. 복호화 메서드는 자료 설명의 키 커밋 값을 비교하고 확인합니다.
3. 복호화 메서드는 서명 필드의 서명을 확인합니다.

   정의한 허용된 인증되지 않은 필드 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 목록에서 ENCRYPT_AND_SIGNSIGN_ONLY, 또는 로 표시된 필드를 식별합니다. ??? 복호화 메서드는 1단계에서 반환된 MAC 키를 사용하여 ENCRYPT_AND_SIGN, SIGN_ONLY또는 로 표시된 필드의 HMAC 값을 다시 계산하고 비교합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT. 그런 다음 암호화 컨텍스트에 저장된 퍼블릭 키를 사용하여 ECDSA 서명을 확인합니다.
4. 복호화 메서드는 일반 텍스트 데이터 키를 사용하여 ENCRYPT_AND_SIGN.로 표시된 각 값을 복호화합니다. 그러면 AWS Database Encryption SDK가 일반 텍스트 데이터 키를 삭제합니다.
5. 복호화 메서드는 일반 텍스트 레코드를 반환합니다.

AWS Database Encryption SDK에서 지원되는 알고리즘 제품군

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

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

AWS Database Encryption SDK는 알고리즘 제품군을 사용하여 데이터베이스의 필드를 암호화하고 서명합니다. 지원되는 모든 알고리즘 제품군은 AES-GCM이라고 하는 Galois/Counter Mode(GCM)와 함께 고급 암호화 표준(AES) 알고리즘을 사용하여 원시 데이터를 암호화합니다. AWS Database Encryption SDK는 256비트 암호화 키를 지원합니다. 인증 태그의 길이는 항상 16바이트입니다.

AWS Database Encryption SDK 알고리즘 제품군

Algorithm	암호화 알고리즘	데이터 키 길이(비트)	키 유도 알고리즘	대칭 서명 알고리즘	비대칭 서명 알고리즘	키 커밋
기본값	AES-GCM	256	HKDF(SHA-512 사용)	HMAC-SHA-384	ECDSA(P-384 및 SHA-384 사용)	HKDF(SHA-512 사용)
ECDSA 디지털 서명이 없는 AES-GCM	AES-GCM	256	HKDF(SHA-512 사용)	HMAC-SHA-384	없음	HKDF(SHA-512 사용)

암호화 알고리즘

사용되는 암호화 알고리즘의 이름 및 모드입니다. AWS Database Encryption SDK의 알고리즘 제품군은 Galois/Counter Mode(GCM)와 함께 고급 암호화 표준(AES) 알고리즘을 사용합니다.

데이터 키 길이

데이터 키의 길이(비트)입니다. AWS Database Encryption SDK는 256비트 데이터 키를 지원합니다. 데이터 키는 HMAC 기반 extract-and-expand 키 유도 함수(HKDF)에 대한 입력으로 사용됩니다. HKDF의 출력은 암호화 알고리즘에서 데이터 암호화 키로 사용됩니다.

키 유도 알고리즘

데이터 암호화 키를 추출하는 데 사용되는 HMAC 기반 추출 및 확장 키 유도 함수(HKDF)입니다. AWS Database Encryption SDK는 RFC 5869에 정의된 HKDF를 사용합니다.

- 사용되는 해시 함수는 SHA-512입니다.
- 추출 단계의 경우:
  - 솔트는 사용하지 않습니다. RFC에 따라 솔트는 0으로 구성된 문자열로 설정됩니다.

    - 입력 키 구성 요소는 키링의 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.

대칭 서명 알고리즘

대칭 서명을 생성하는 데 사용되는 해시 기반 메시지 인증 코드(HMAC) 알고리즘입니다. 지원되는 모든 알고리즘 제품군에는 HMAC 확인이 포함됩니다.

AWS Database Encryption SDK는 재료 설명과 `ENCRYPT_AND_SIGN`, `SIGN_ONLY`또는 로 표시된 모든 필드를 직렬화합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. 그런 다음 HMAC를 암호화 해시 함수 알고리즘(SHA-384)과 함께 사용하여 표준화에 서명합니다.

대칭 HMAC 서명은 AWS Database Encryption SDK가 레코드에 추가하는 새 필드(`aws_dbe_foot`)에 저장됩니다.

비대칭 서명 알고리즘

비대칭 디지털 서명을 생성하는 데 사용되는 서명 알고리즘입니다.

AWS Database Encryption SDK는 재료 설명과 `ENCRYPT_AND_SIGN`, `SIGN_ONLY`또는 로 표시된 모든 필드를 직렬화합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. 그런 다음 다음 세부 정보와 함께 타원 곡선 디지털 서명 알고리즘(ECDSA)을 사용하여 표준화에 서명합니다.

- 사용되는 타원 곡선은 디지털 서명 표준(DSS)(FIPS PUB 186-4)에 정의된 P-384입니다. http://doi.org/10.6028/NIST.FIPS.186-4
- 사용되는 해시 함수는 SHA-384입니다.

비대칭 ECDSA 서명은 `aws_dbe_foot` 필드에 대칭 HMAC 서명과 함께 저장됩니다.

ECDSA 디지털 서명은 기본적으로 포함되지만 필수는 아닙니다.

키 커밋

커밋 키를 도출하는 데 사용되는 HMAC 기반 extract-and-expand 키 유도 함수(HKDF)입니다.

- 사용되는 해시 함수는 SHA-512입니다.

- 추출 단계의 경우:
  - 솔트는 사용하지 않습니다. RFC에 따라 솔트는 0으로 구성된 문자열로 설정됩니다.
  - 입력 키 구성 요소는 키링의 데이터 키입니다.
- 확장 단계의 경우:
  - 입력 의사 난수 키는 추출 단계의 출력입니다.
  - 입력 정보는 빅 엔디안 바이트 순서로 `COMMITKEY` 문자열의 UTF-8-encoded 바이트입니다.
  - 출력 키 지정 구성 요소의 길이는 256비트입니다. 이 출력은 커밋 키로 사용됩니다.

커밋 키는 자료 설명에 대해 고유한 256비트 해시 기반 메시지 인증 코드(HMAC) 해시인 레코드 커밋을 계산합니다. 알고리즘 제품군에 키 커밋을 추가하는 방법에 대한 기술적인 설명은 Cryptology ePrint Archive의 키 커밋 AEAD를 참조하세요.

기본 알고리즘 제품군

기본적으로 AWS Database Encryption SDK는 AES-GCM, HMAC 기반 extract-and-expand 키 유도 함수(HKDF), HMAC 확인, ECDSA 디지털 서명, 키 커밋 및 256비트 암호화 키가 있는 알고리즘 제품군을 사용합니다.

기본 알고리즘 제품군에는 HMAC 확인(대칭 서명) 및 ECDSA 디지털 서명(대칭 서명)이 포함됩니다. 이러한 서명은 AWS Database Encryption SDK가 레코드에 추가하는 새 필드(`aws_dbe_foot`)에 저장됩니다. ECDSA 디지털 서명은 권한 부여 정책에서 한 사용자 집합이 데이터를 암호화하고 다른 사용자 집합이 데이터를 복호화하도록 허용하는 경우에 특히 유용합니다.

기본 알고리즘 제품군은 데이터 키를 레코드에 연결하는 HMAC 해시인 키 커밋도 도출합니다. 키 커밋 값은 자료 설명 및 커밋 키에서 계산된 HMAC입니다. 그런 다음 자료 설명에 키 커밋 값을 저장합니다. 키 커밋이 포함된 알고리즘 제품군은 각 사이퍼텍스트가 하나의 일반 텍스트로만 복호화되도록 합니다. 이를 위해 암호화 알고리즘에 대한 입력으로 사용된 데이터 키를 검증합니다. 암호화할 때 알고리즘 제품군은 키 커밋 HMAC를 도출합니다. 암호를 복호화하기 전에 데이터 키가 동일한 키 커밋 HMAC를 생성하는지 확인합니다. 그러지 않으면 복호화 호출이 실패합니다.

ECDSA 디지털 서명이 없는 AES-GCM

기본 알고리즘 제품군은 대부분의 애플리케이션에 적합할 수 있지만 대체 알고리즘 제품군을 선택할 수 있습니다. 예를 들어 일부 신뢰 모델은 ECDSA 디지털 서명이 없는 알고리즘 제품군으로 충족됩니다. 데이터를 암호화하는 사용자와 데이터를 해독하는 사용자가 동일하게 신뢰할 수 있는 경우에만 이 제품군을 사용합니다.

모든 AWS Database Encryption SDK 알고리즘 제품군에는 HMAC 확인(대칭 서명)이 포함됩니다. 유일한 차이점은 ECDSA 디지털 서명이 없는 AES-GCM 알고리즘 제품군에 추가적인 신뢰성 및 거부 방지 계층을 제공하는 비대칭 서명이 없다는 것입니다.

예를 들어 키링, , `wrappingKeyA` `wrappingKeyB`및에 래핑 키가 여러 개 `wrappingKeyC`있고를 사용하여 레코드`wrappingKeyA`를 복호화하는 경우 HMAC 대칭 서명은에 액세스할 수 있는 사용자가 레코드를 암호화했는지 확인합니다`wrappingKeyA`. 기본 알고리즘 제품군을 사용한 경우 HMACs에 대한 동일한 확인을 제공하고 `wrappingKeyA`추가로 ECDSA 디지털 서명을 사용하여에 대한 암호화 권한이 있는 사용자가 레코드를 암호화했는지 확인합니다`wrappingKeyA`.

디지털 서명이 없는 AES-GCM 알고리즘 제품군을 선택하려면 암호화 구성에 다음 코드 조각을 포함합니다.

Java

다음 코드 조각은 ECDSA 디지털 서명이 없는 AES-GCM 알고리즘 제품군을 지정합니다. 자세한 내용은 the section called “암호화 구성” 단원을 참조하십시오.

```
.algorithmSuiteId(
    DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

C# / .NET

다음 코드 조각은 ECDSA 디지털 서명이 없는 AES-GCM 알고리즘 제품군을 지정합니다. 자세한 내용은 the section called “암호화 구성” 단원을 참조하십시오.

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Rust

다음 코드 조각은 ECDSA 디지털 서명이 없는 AES-GCM 알고리즘 제품군을 지정합니다. 자세한 내용은 the section called “암호화 구성” 단원을 참조하십시오.

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

에서 AWS Database Encryption SDK 사용 AWS KMS

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK를 사용하려면 키링을 구성하고 하나 이상의 래핑 키를 지정해야 합니다. 키 인프라가 없는 경우 AWS Key Management Service (AWS KMS)를 사용하는 것이 좋습니다.

AWS Database Encryption SDK는 두 가지 유형의 AWS KMS 키링을 지원합니다. 기존 AWS KMS 키링은 AWS KMS keys을 사용하여 데이터 키를 생성, 암호화 및 복호화합니다. 대칭 암호화(SYMMETRIC_DEFAULT) 또는 비대칭 RSA KMS 키를 사용할 수 있습니다. AWS Database Encryption SDK는 고유한 데이터 키로 모든 레코드를 암호화하고 서명하므로 AWS KMS 키링은 모든 암호화 및 복호화 작업에 AWS KMS 대해를 호출해야 합니다. 호출 수를 최소화해야 하는 애플리케이션의 경우 AWS KMS AWS Database Encryption SDK는 AWS KMS 계층적 키링도 지원합니다. 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. 가능하면 AWS KMS 키링을 사용하는 것이 좋습니다.

와 상호 작용하려면 AWS KMS AWS Database Encryption SDK에의 AWS KMS 모듈이 필요합니다 AWS SDK for Java.

에서 AWS Database Encryption SDK를 사용할 준비를 하려면 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 Amazon 리소스 이름(ARN)이 필요합니다 AWS KMS key. AWS KMS key의 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. 의 지침과 3단계에서 생성한 AWS SDK for Java 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 작용하는 AWS Database Encryption SDK의 코드 샘플은이 단계를 완료했다고 AWS KMS 가정합니다.

AWS Database Encryption SDK 구성

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

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

주제

프로그래밍 언어 선택

DynamoDB용 AWS Database Encryption SDK는 여러 프로그래밍 언어로 제공됩니다. 언어 구현은 서로 다른 방식으로 구현될 수 있지만 완전히 상호 연동되고 동일한 기능을 제공하도록 설계되었습니다. 일반적으로 애플리케이션과 호환되는 라이브러리를 사용합니다.

래핑 키 선택

AWS Database Encryption SDK는 고유한 대칭 데이터 키를 생성하여 각 필드를 암호화합니다. 데이터 키를 구성, 관리 또는 사용할 필요가 없습니다. AWS Database Encryption SDK가 자동으로 수행합니다.

하지만 각 데이터 키를 암호화하려면 래핑 키를 하나 이상 선택해야 합니다. AWS Database Encryption SDK는 AWS Key Management Service(AWS KMS) 대칭 암호화 KMS 키와 비대칭 RSA KMS 키를 지원합니다. 또한 다양한 크기로 제공하는 AES 대칭 키와 RSA 비대칭 키를 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 키링을 사용합니다. 사용하는 키링 유형에 따라 하나의 래핑 키를 지정하거나 동일하거나 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 1개)는 암호화된 필드와 함께 저장된 자료 설명에 저장됩니다. 데이터를 복호화하려면 AWS Database Encryption SDK가 먼저 래핑 키 중 하나를 사용하여 암호화된 데이터 키를 복호화해야 합니다.

가능하면 AWS KMS 키링 중 하나를 사용하는 것이 좋습니다. AWS Database Encryption SDK는 AWS KMS 키링과 AWS KMS 계층적 키링을 제공하므로 호출 횟수가 줄어듭니다 AWS KMS. 키링 AWS KMS key 에서를 지정하려면 지원되는 AWS KMS 키 식별자를 사용합니다. AWS KMS 계층적 키링을 사용하는 경우 키 ARN을 지정해야 합니다. 키의 키 식별자에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 키 식별자를 AWS KMS 참조하세요.

- AWS KMS 키링으로 암호화할 때 대칭 암호화 KMS 키에 유효한 키 식별자(키 ARN, 별칭 이름, 별칭 ARN 또는 키 ID)를 지정할 수 있습니다. 비대칭 RSA KMS 키를 사용하는 경우 ARN 키를 지정해야 합니다.

  암호화할 때 KMS 키의 별칭 이름 또는 별칭 ARN을 지정하면 AWS Database Encryption SDK는 해당 별칭과 현재 연결된 키 ARN을 저장하지만 별칭은 저장하지 않습니다. 별칭을 변경해도 데이터 키를 복호화하는 데 사용되는 KMS 키에는 영향을 주지 않습니다.
- 기본적으로 AWS KMS 키링은 엄격 모드(특정 KMS 키를 지정하는 경우)로 레코드를 복호화합니다. 복호화을 위해 AWS KMS keys 를 식별하려면 키 ARN을 사용해야 합니다.

  AWS KMS 키링으로 암호화하면 AWS Database Encryption SDK는 암호화된 데이터 키와 함께 자료 설명 AWS KMS key 에의 키 ARN을 저장합니다. 엄격 모드에서 복호화할 때 AWS Database Encryption SDK는 래핑 키를 사용하여 암호화된 데이터 키를 복호화하기 전에 키링에 동일한 키 ARN이 나타나는지 확인합니다. 다른 키 식별자를 사용하는 경우 식별자가 동일한 키를 참조 AWS KMS key하더라도 AWS Database Encryption SDK는를 인식하거나 사용하지 않습니다.
- 검색 모드에서 암호를 복호화할 때는 래핑 키를 지정하지 않습니다. 먼저 AWS Database Encryption SDK는 자료 설명에 저장된 키 ARN을 사용하여 레코드를 복호화하려고 시도합니다. 그렇지 않으면 AWS Database Encryption SDK는 누가 해당 KMS 키를 소유하거나 액세스할 수 있는지에 관계없이 레코드를 암호화한 KMS 키를 사용하여 레코드를 AWS KMS 복호화하도록 요청합니다.

원시 AES 키 또는 원시 RSA 키 페어를 키링의 래핑 키로 지정하려면 네임스페이스와 이름을 지정해야 합니다. 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 동일하더라도 AWS Database Encryption SDK는 래핑 키를 인식하거나 사용하지 않습니다.

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`

다음 예제에서는 검색 필터를 생성하는 방법을 보여줍니다. 코드를 사용하기 전에 예제 값을 AWS 계정 및 파티션의 유효한 값으로 바꿉니다.

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
```

C# / .NET

```
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;
```

멀티테넌트 데이터베이스 작업

AWS Database Encryption SDK를 사용하면 각 테넌트를 고유한 암호화 자료로 격리하여 공유 스키마가 있는 데이터베이스에 대한 클라이언트 측 암호화를 구성할 수 있습니다. 멀티테넌트 데이터베이스를 고려할 때는 잠시 시간을 내어 보안 요구 사항과 멀티테넌시가 이에 미치는 영향을 검토하세요. 예를 들어 멀티테넌트 데이터베이스를 사용하면 AWS Database Encryption SDK를 다른 서버 측 암호화 솔루션과 결합하는 기능에 영향을 미칠 수 있습니다.

데이터베이스 내에서 암호화 작업을 수행하는 사용자가 여러 명인 경우 AWS KMS 키링 중 하나를 사용하여 각 사용자에게 암호화 작업에 사용할 고유한 키를 제공할 수 있습니다. 멀티테넌트 클라이언트 측 암호화 솔루션의 데이터 키 관리는 복잡할 수 있습니다. 가능하면 테넌트별로 데이터를 구성하는 것이 좋습니다. 테넌트가 프라이머리 키 값(예: Amazon DynamoDB 테이블의 파티션 키)으로 식별되는 경우 키를 더 쉽게 관리할 수 있습니다.

AWS KMS 키링을 사용하여 고유한 AWS KMS 키링 및 로 각 테넌트를 격리할 수 있습니다 AWS KMS keys. 테넌트당 수행된 호출량 AWS KMS 에 따라 AWS KMS 계층적 키링을 사용하여 호출을 최소화할 수 있습니다 AWS KMS. AWS KMS 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. AWS KMS 계층적 키링을 사용하여 데이터베이스에서 검색 가능한 암호화를 구현해야 합니다.

서명된 비컨 만들기

AWS Database Encryption SDK는 표준 비컨과 복합 비컨을 사용하여 쿼리된 전체 데이터베이스를 해독하지 않고도 암호화된 레코드를 검색할 수 있는 검색 가능한 암호화 솔루션을 제공합니다. 그러나 AWS Database Encryption SDK는 일반 텍스트 서명 필드에서만 구성할 수 있는 서명된 비컨도 지원합니다. 서명된 비컨은 `SIGN_ONLY` 및 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 필드에 대해 복잡한 쿼리를 인덱싱하고 수행하는 복합 비컨의 한 유형입니다.

예를 들어 멀티테넌트 데이터베이스가 있는 경우 특정 테넌트의 키로 암호화된 레코드를 데이터베이스에 쿼리할 수 있는 서명된 비컨을 만들 수 있습니다. 자세한 내용은 멀티테넌트 데이터베이스의 비컨 쿼리 단원을 참조하십시오.

AWS KMS 계층적 키링을 사용하여 서명된 비컨을 생성해야 합니다.

서명된 비컨를 구성하려면 다음 값을 제공해야 합니다.

Java

복합 비컨 구성

다음 예제에서는 서명된 비컨 구성 내에서 로컬로 서명된 부분 목록을 정의합니다.

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
    .name("compoundBeaconName")
    .split(".")
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

비컨 버전 정의

다음 예제에서는 비컨 버전에서 전역적으로 서명된 부분 목록을 정의합니다. 비컨 버전 정의에 대한 자세한 내용은 비컨 사용을 참조하세요.

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
```

```
);
```

C# / .NET

전체 코드 샘플 보기: BeaconConfig.cs

서명된 비컨 구성

다음 예제에서는 서명된 비컨 구성 내에서 로컬로 서명된 부분 목록을 정의합니다.

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Signed = signedPartList,
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

비컨 버전 정의

다음 예제에서는 비컨 버전에서 전역적으로 서명된 부분 목록을 정의합니다. 비컨 버전 정의에 대한 자세한 내용은 비컨 사용을 참조하세요.

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
```

```
};
```

로컬 또는 전역적으로 정의된 목록에서 서명된 부분을 정의할 수 있습니다. 가능하면 비컨 버전의 글로벌 목록에서 서명된 부분을 정의하는 것이 좋습니다. 서명된 부분을 전역적으로 정의하면 각 부분을 한 번 정의한 다음 여러 복합 비컨 구성에서 해당 부분을 재사용할 수 있습니다. 서명된 부분을 한 번만 사용하려는 경우 서명된 비컨 구성의 로컬 목록에서 정의할 수 있습니다. 생성자 목록에서 로컬 부분과 글로벌 부분을 모두 참조할 수 있습니다.

서명된 부분 목록을 전역적으로 정의하는 경우 서명된 비컨이 비컨 구성의 필드를 조합할 수 있는 가능한 모든 방법을 식별하는 생성자 부분 목록을 제공해야 합니다.

> **Note**
>
> 서명된 부분 목록을 전역적으로 정의하려면 AWS Database Encryption SDK 버전 3.2 이상을 사용해야 합니다. 새 부분을 전역적으로 정의하기 전에 모든 리더에 새 버전을 배포합니다. 기존 비컨 구성을 업데이트하여 서명된 부분 목록을 전역적으로 정의할 수 없습니다.

비컨 이름

비컨을 쿼리할 때 사용하는 이름.

서명된 비컨 이름은 암호화되지 않은 필드의 이름과 같을 수 없습니다. 두 비컨이 동일한 비컨 이름을 가질 수는 없습니다.

분할 캐릭터

서명된 비컨을 구성하는 부분을 구분하는 데 사용되는 문자입니다.

분할된 문자는 서명된 비컨을 구성하는 모든 필드의 일반 텍스트 값에 나타날 수 없습니다.

서명된 부분 목록

서명된 비컨에 포함된 서명된 필드를 식별합니다.

각 부분에는 이름, 출처 및 접두사가 포함되어야 합니다. 소스는 파트가 식별하는 `SIGN_ONLY` 또는 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 필드입니다. 소스는 필드 이름이거나 중첩된 필드의 값을 참조하는 인덱스이어야 합니다. 파트 이름이 소스를 식별하는 경우 소스를 생략하면 AWS Database Encryption SDK가 자동으로 이름을 소스로 사용합니다. 가능하면 소스를 부분 이름으로 지정하는 것이 좋습니다. 접두사는 임의의 문자열일 수 있지만 고유해야 합니다. 서명된 비

컨의 서명된 두 부분이 동일한 접두사를 가질 수 없습니다. 부분을 복합 비컨이 제공하는 다른 부분과 구분하는 짧은 값을 사용하는 것이 좋습니다.

가능하면 서명된 부분을 전역적으로 정의하는 것이 좋습니다. 하나의 복합 비컨에서만 사용하려는 경우 서명된 부분을 로컬에서 정의하는 것이 좋습니다. 로컬에서 정의된 파트는 전역적으로 정의된 파트와 동일한 접두사 또는 이름을 가질 수 없습니다.

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
    SignedPart signedPartExample = SignedPart.builder()
        .name("signedFieldName")
        .prefix("S-")
        .build();
    signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

생성자 목록(선택 사항)

서명된 비컨으로 서명된 부분을 조합할 수 있는 다양한 방법을 정의하는 생성자를 식별합니다.

생성자 목록을 지정하지 않으면 AWS Database Encryption SDK가 다음 기본 생성자와 함께 서명된 비컨을 수집합니다.

- 서명된 모든 부분은 서명된 부분 목록에 추가된 순서대로
- 모든 부분이 필요합니다.

Constructors

각 생성자는 서명된 비컨을 조합할 수 있는 한 가지 방법을 정의하는 생성자 부분을 순서대로 나열한 목록입니다. 생성자 부분은 목록에 추가된 순서대로 함께 결합되며 각 부분은 지정된 분할 문자로 구분됩니다.

각 생성자 부분은 서명된 부분의 이름을 지정하고 생성자 내에서 해당 부분이 필수인지 선택적인지 정의합니다. 예를 들어 `Field1`, `Field1.Field2`, 및 `Field1.Field2.Field3`에 대한

서명된 비컨을 조회하고자 한다면, `Field2` 및 `Field3`을 선택 사항으로 표시하고 생성자를 하나 생성합니다.

생성자마다 필수 부분이 하나 이상 있어야 합니다. 쿼리에 `BEGINS_WITH` 연산자를 사용할 수 있도록 각 생성자의 첫 번째 부분을 필수로 설정하는 것이 좋습니다.

생성자의 필수 부분이 모두 레코드에 있으면 생성자는 성공합니다. 새 레코드를 작성하면 서명된 비컨은 생성자 목록을 사용하여 제공된 값에서 비컨을 조합할 수 있는지 여부를 결정합니다. 생성자 목록에 생성자가 추가된 순서대로 비컨을 조합하려고 시도하고 성공한 첫 번째 생성자를 사용합니다. 생성자가 성공하지 못하면 비컨이 레코드에 기록되지 않습니다.

쿼리 결과가 정확한지 확인하려면 모든 리더와 작성자가 동일한 순서의 생성자를 지정해야 합니다.

다음 절차에 따라 생성자 목록을 지정하세요.

1. 서명된 각 부분에 대해 생성자 부분을 만들어 해당 부분이 필요한지 여부를 정의합니다.

   생성자 부분 이름은 서명된 필드의 이름이어야 합니다.

   다음 예제에서는 서명된 필드 하나에 대해 생성자 부분을 만드는 방법을 보여줍니다.

   Java

   ```
   ConstructorPart field1ConstructorPart = ConstructorPart.builder()
           .name("Field1")
           .required(true)
           .build();
   ```

   C# / .NET

   ```
   var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
    = true };
   ```

2. 1단계에서 만든 생성자 부분을 사용하여 서명된 비컨을 조합할 수 있는 가능한 모든 방법에 맞는 생성자를 만듭니다.

   예를 들어 `Field1.Field2.Field3` 및 `Field4.Field2.Field3`에 대해 쿼리하려면 두 개의 생성자를 만들어야 합니다. `Field1` 및 `Field4`은 두 개의 별도 생성자에 정의되어 있으므로 둘 다 필요할 수 있습니다.

Java

```
// Create a list for Field1.Field2.Field3 queries
List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
field123ConstructorPartList.add(field1ConstructorPart);
field123ConstructorPartList.add(field2ConstructorPart);
field123ConstructorPartList.add(field3ConstructorPart);
Constructor field123Constructor = Constructor.builder()
        .parts(field123ConstructorPartList)
        .build();
// Create a list for Field4.Field2.Field1 queries
List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
field421ConstructorPartList.add(field4ConstructorPart);
field421ConstructorPartList.add(field2ConstructorPart);
field421ConstructorPartList.add(field1ConstructorPart);
Constructor field421Constructor = Constructor.builder()
        .parts(field421ConstructorPartList)
        .build();
```

C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
 var field123ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field1ConstructorPart,
 field2ConstructorPart, field3ConstructorPart }
};
// Create a list for Field4.Field2.Field1 queries
var field421ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field4ConstructorPart,
 field2ConstructorPart, field1ConstructorPart }
};
```

3. 2단계에서 만든 모든 생성자를 포함하는 생성자 목록을 만듭니다.

Java

```
List<Constructor> constructorList = new ArrayList<>();
constructorList.add(field123Constructor)
constructorList.add(field421Constructor)
```

C# / .NET

```
var constructorList = new List<Constructor>
{
    field123Constructor,
    field421Constructor
};
```

4. 서명된 `constructorList` 비컨을 만드는 시기를 지정합니다.

AWS Database Encryption SDK의 키 스토어

AWS Database Encryption SDK에서 키 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.

비컨 키

검색 가능한 암호화를 위한 비컨을 생성하는 데 사용되는 데이터 키입니다. 자세한 내용은 검색 가능한 암호화를 참조하세요.

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 검색을 위해 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. AWS Database Encryption SDK는 다음과 같은 키 스토어 작업 구성을 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 발생하고 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);
```

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)?;
```

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);
```

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()?;
```

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.

애플리케이션 컨트롤 플레인의 KeyStore Admin 인터페이스를 통해 `CreateKey` 작업을 사용하는 것이 좋습니다. 이 접근 방식은 키 관리 모범 사례에 부합합니다.

데이터 영역에 브랜치 키를 생성하지 마십시오. 이 방법을 사용하면 다음과 같은 결과가 발생할 수 있습니다.

- 에 대한 불필요한 호출 AWS KMS
- 동시성이 높은 환경에서 AWS KMS 에 대한 여러 동시 호출
- 백업 DynamoDB 테이블에 대한 여러 TransactWriteItems 호출입니다.

`CreateKey` 작업에는 기존 브랜치 키를 덮어쓰지 않도록 `TransactWriteItems` 호출에 조건 확인이 포함됩니다. 그러나 데이터 영역에서 키를 생성해도 리소스 사용량이 비효율적이고 잠재적인 성능 문제가 발생할 수 있습니다.

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
  });
```

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();
```

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"
  }
```

> **Note**
>
> 데이터베이스를 검색 가능한 암호화로 구성하지 않았더라도 `CreateKey` 작업을 수행하면 활성 브랜치 키와 비컨 키가 생성됩니다. 두 키 모두 키 스토어에 저장됩니다. 자세한 내용은 검색 가능한 암호화를 위한 계층적 키링 사용을 참조하세요.

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 79옥틸리온(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.

> **Warning**
>
> 테스트 환경에서 브랜치 키를 삭제하는 것은 되돌릴 수 없습니다. 삭제된 브랜치 키는 복구할 수 없습니다. 테스트 환경에서 ID가 동일한 브랜치 키를 삭제하고 다시 생성하면 다음과 같은 문제가 발생할 수 있습니다.
>
> - 이전 테스트 실행의 재료가 캐시에 남아 있을 수 있습니다.
> - 일부 테스트 호스트 또는 스레드는 삭제된 브랜치 키를 사용하여 데이터를 암호화할 수 있습니다.
> - 삭제된 브랜치로 암호화된 데이터는 해독할 수 없습니다.
>
> 통합 테스트에서 암호화 실패를 방지하려면:
>
> - 새 브랜치 키를 생성하기 전에 계층적 키링 참조를 재설정하거나
> - 각 테스트에 고유한 브랜치 키 IDs 사용

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});
```

Rust

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

키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK는 키링을 사용하여 봉투 암호화를 수행합니다. 키 링은 데이터 키를 생성, 암호화 및 복호화합니다. 키링에 따라 각 암호화된 레코드를 보호하는 고유한 데이터 키의 원본과 해당 데이터 키를 암호화하는 래핑 키가 결정됩니다. 암호화할 때 키 링을 지정하고 암호를 복호화할 때는 동일하거나 다른 키 링을 지정합니다.

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.

키링에 따라 데이터 키, 궁극적으로 데이터를 보호하는 래핑 키가 결정됩니다. 작업에 가장 적합한 가장 안전한 래핑 키를 사용하세요. 가능하면 하드웨어 보안 모듈(HSM) 또는 AWS Key Management Service(AWS KMS)의 KMS 키 또는 AWS CloudHSM의 암호화 키와 같은 키 관리 인프라로 보호되는 래핑 키를 사용합니다.

AWS Database Encryption SDK는 여러 키링과 키링 구성을 제공하며 사용자 지정 키링을 직접 생성할 수 있습니다. 또한 유형이 같거나 다른 키링을 하나 이상 포함하는 다중 키링을 만들 수 있습니다.

주제

- 다중 키링

키링 작동 방식

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

데이터베이스의 필드를 암호화하고 서명하면 AWS Database Encryption SDK는 키링에 암호화 자료를 요청합니다. 키링은 일반 텍스트 데이터 키, 키링의 각 래핑 키로 암호화된 데이터 키의 복사본, 데이터 키와 연결된 MAC 키를 반환합니다. AWS Database Encryption SDK는 일반 텍스트 키를 사용하여 데이터를 암호화한 다음 가능한 한 빨리 메모리에서 일반 텍스트 데이터 키를 제거합니다. 그런 다음 AWS Database Encryption SDK는 암호화된 데이터 키와 암호화 및 서명 지침과 같은 기타 정보를 포함하는 자료 설명을 추가합니다. AWS Database Encryption SDK는 MAC 키를 사용하여 자료 설명 및 `ENCRYPT_AND_SIGN` 또는 로 표시된 모든 필드의 정식화를 통해 해시 기반 메시지 인증 코드(HMACs)를 계산합니다`SIGN_ONLY`.

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 있거나 액세스 권한이 있어야 합니다.

AWS Database Encryption SDK는 암호화된 데이터 키를 자료 설명에서 키링으로 전달하고 키링에 암호화 해제를 요청합니다. 키링은 해당 래핑 키를 사용하여 암호화된 데이터 키 중 하나를 암호화 해제하고 일반 텍스트 데이터 키를 반환합니다. AWS Database Encryption SDK는 일반 텍스트 데이터 키를 이용해 데이터를 복호화합니다. 키 링에 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 암호화하면 다중 키 링은 다중 키 링을 구성하는 모든 키 링의 모든 래핑 키와 데이터 키와 연결된 MAC 키로 암호화된 데이터 키의 복사본을 반환합니다. 다중 키링의 래핑 키 중 하나를 포함하는 키링을 사용하여 데이터를 복호화할 수 있습니다.

AWS KMS 키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS KMS 키링은 대칭 암호화 또는 비대칭 RSAAWS 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 사용하는 것이 좋습니다.

AWS KMS 키링에서 대칭 다중 리전 KMS 키를 사용할 수도 있습니다. 다중 리전을 사용하는 자세한 내용과 예제는 섹션을 AWS KMS keys참조하세요다중 리전 사용 AWS KMS keys. 다중 리전 키에 대한 자세한 내용은 AWS Key Management Service 개발자 가이드의 다중 리전 키 사용을 참조하세요.

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.

레코드를 암호화하려면 생성기 키가 있어야 합니다. AWS KMS 키링에 AWS KMS 키가 하나만 있는 경우 해당 키는 데이터 키를 생성하고 암호화하는 데 사용됩니다.

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
- 다중 리전 사용 AWS KMS keys
- AWS KMS 검색 키링 사용
- AWS KMS 리전 검색 키링 사용

AWS KMS 키링에 필요한 권한

AWS Database Encryption SDK에는가 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 키링을 사용하여 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 인증 및 액세스 제어를 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.

- AWS KMS 키링으로 암호화하려면 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN을 사용하여 데이터를 암호화할 수 있습니다.

  > **Note**
  >
  > 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 키링으로 복호화하려면 키 ARN을 사용하여를 식별해야 합니다 AWS KMS keys. 자세한 내용은 래핑 키 선택을 참조하세요.
- 암호화 및 복호화에 사용되는 키 링에서는 키 ARN을 사용하여 AWS KMS keys를 식별해야 합니다.

복호화 시 AWS Database Encryption SDK는 AWS KMS 키링에서 암호화된 데이터 키 중 하나를 복호화할 수 AWS KMS key 있는를 검색합니다. 특히 AWS Database Encryption SDK는 자료 설명의 각 암호화된 데이터 키에 대해 다음 패턴을 사용합니다.

- AWS Database Encryption SDK는 자료 설명의 메타데이터에서 데이터 키를 암호화 AWS KMS key 한의 키 ARN을 가져옵니다.
- AWS Database Encryption SDK는 복호화 키링에서 일치하는 키 ARN이 AWS KMS key 있는를 검색합니다.
- 키링에서 일치하는 키 ARN이 AWS KMS key 있는를 찾으면 AWS Database Encryption SDK는 KMS 키를 AWS KMS 사용하여 암호화된 데이터 키를 복호화하도록 요청합니다.
- 그러지 않으면 암호화된 다음 데이터 키(있는 경우)로 건너뛰게 됩니다.

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. AWS KMS key 은 대칭 암호화 키(SYMMETRIC_DEFAULT) 또는 비대칭 RSA KMS 키여야 합니다. 대칭 암호화 다중 리전 KMS 키를 사용할 수도 있습니다. 다중 AWS KMS 키링에서 하나 이상의 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.)

암호화 키링 또는 다중 키링의 모든 래핑 키는 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.

다음 예제에서는 `CreateAwsKmsMrkMultiKeyring` 메서드를 사용하여 대칭 암호화 KMS AWS KMS 키를 사용하여 키링을 생성합니다. `CreateAwsKmsMrkMultiKeyring` 메서드는 AWS KMS 클라이언트를 자동으로 생성하고 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리하도록 합니

다. 이 예제에서는 키 ARNs을 사용하여 KMS 키를 식별합니다. 자세한 내용은 AWS KMS 키링 AWS KMS keys 에서 식별 섹션을 참조하세요.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = kmsKeyArn
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
let mat_prov = client::Client::from_conf(provider_config)?;
let kms_keyring = mat_prov
    .create_aws_kms_mrk_multi_keyring()
    .generator(kms_key_id)
    .send()
    .await?;
```

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`

Java

```
 final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsRsaKeyringInput createAwsKmsRsaKeyringInput =
    CreateAwsKmsRsaKeyringInput.builder()
        .kmsClient(KmsClient.create())
        .kmsKeyId(rsaKMSKeyArn)
        .publicKey(publicKey)
        .encryptionAlgorithm(EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256)
        .build();
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsRsaKeyringInput = new CreateAwsKmsRsaKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = rsaKMSKeyArn,
    PublicKey = publicKey,
    EncryptionAlgorithm = EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256
};
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_rsa_keyring = mpl
    .create_aws_kms_rsa_keyring()
    .kms_key_id(rsa_kms_key_arn)
    .public_key(public_key)
```

```
.encryption_algorithm(aws_sdk_kms::types::EncryptionAlgorithmSpec::RsaesOaepSha256)
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .send()
    .await?;
```

다중 리전 사용 AWS KMS keys

AWS Database Encryption SDK에서 다중 리전을 래핑 키 AWS KMS keys 로 사용할 수 있습니다. 하나의에서 다중 리전 키로 암호화하는 경우 다른에서 관련 다중 리전 키를 사용하여 복호화할 AWS 리전수 있습니다 AWS 리전.

다중 리전 KMS 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 AWS Database Encryption SDK에는 AWS KMS multi-Region-aware 키링이 포함되어 있습니다. 이 `CreateAwsKmsMrkMultiKeyring` 방법은 단일 리전 키와 다중 리전 키를 모두 지원합니다.

- 단일 리전 키의 경우 다중 리전 인식 기호는 단일 리전 AWS KMS 키링처럼 작동합니다. 데이터를 암호화한 단일 리전 키로만 사이퍼텍스트 복호화를 시도합니다. AWS KMS 키링 환경을 간소화하려면 대칭 암호화 KMS 키를 사용할 때마다 `CreateAwsKmsMrkMultiKeyring` 메서드를 사용하는 것이 좋습니다.
- 다중 리전 키의 경우 다중 리전 인식 기호는 데이터를 암호화한 것과 동일한 다중 리전 키 또는 사용자가 지정한 리전의 관련 다중 리전 키를 사용하여 사이퍼텍스트를 복호화하려고 시도합니다.

KMS 키를 두 개 이상 사용하는 다중 리전 인식 키링에서는 단일 리전 및 다중 리전 키를 여러 개 지정할 수 있습니다. 그러나 관련 다중 리전 키의 각 집합에서 하나의 키만 지정할 수 있습니다. 키 ID가 같은 키 식별자를 두 개 이상 지정하면 생성자 호출이 실패합니다.

다음 예제에서는 다중 리전 KMS AWS KMS 키를 사용하여 키링을 생성합니다. 이 예제에서는 다중 리전 키를 생성기 키로 지정하고 단일 리전 키를 하위 키로 지정합니다.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
    CreateAwsKmsMrkMultiKeyringInput.builder()
            .generator(multiRegionKeyArn)
            .kmsKeyIds(Collections.singletonList(kmsKeyArn))
            .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = multiRegionKeyArn,
    KmsKeyIds = new List<String> { kmsKeyArn }
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

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

let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(multiRegion_key_arn)
    .kms_key_ids(vec![key_arn.to_string()])
    .send()
    .await?;
```

다중 리전 AWS KMS 키링을 사용하는 경우 엄격 모드에서 사이퍼텍스트를 해독하거나 검색 모드에서 사이퍼텍스트를 해독할 수 있습니다. 엄격 모드에서 사이퍼텍스트를 복호화하려면 사이퍼텍스트를 복호화하는 리전에서 관련 다중 리전 키의 키 ARN을 사용하여 다중 리전 인식 기호를 인스턴스화합니다. 다른 리전(예: 레코드가 암호화된 리전)에 있는 관련 다중 리전 키의 키 ARN을 지정하면 다중 리전 인식 기호가 해당 AWS KMS key에 대한 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.

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 Database Encryption SDK는 암호화에 사용되는 다중 리전 키에 대한 교차 리전 호출을 시도하지 않습니다.

AWS KMS 검색 키링 사용

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

AWS Database Encryption SDK는 표준 AWS KMS 검색 키링과 AWS KMS 다중 리전 키에 대한 검색 키링을 제공합니다. AWS Database 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.

복호화할 때 검색 키링을 사용하면 AWS Database Encryption SDK가 해당 키를 소유하거나 액세스할 수 AWS KMS key 있는 사용자와 관계없이 암호화한를 사용하여 암호화된 데이터 키를 복호화 AWS KMS 하도록 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. 검색 키링을 사용하여 단독 또는 다중 키링으로 데이터를 암호화하는 경우 암호화 작업이 실패합니다

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

- 인증 - AWS KMS 검색 키링은 호출자가 복호화에 사용할 권한이 AWS KMS key 있는 한 자료 설명에서 데이터 키를 암호화하는 데 사용된 모든 AWS KMS key 를 사용할 수 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 AWS Database Encryption SDK가 다른 AWS 계정 및 리전 AWS KMS keys 에서에 의해 암호화된 데이터 키를 포함하여 암호화된 모든 데이터 키를 복호화하려고 하고 AWS KMS keys 호출자가 복호화에 사용할 권한이 없기 때문에 검색 키링이 다른 키링보다 눈에 띄게 느릴 수 있습니다.

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

다음 코드 예제에서는 AWS Database Encryption SDK가 사용할 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.

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);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

AWS KMS 리전 검색 키링 사용

AWS KMS 리전 검색 키링은 KMS 키의 ARN을 지정하지 않는 키링입니다. 대신 AWS Database Encryption SDK가 특히 KMS 키만 사용하여 복호화할 수 있습니다 AWS 리전.

AWS KMS 리전 검색 키링을 사용하여 복호화할 때 AWS Database Encryption SDK는 지정된 AWS KMS key 의에서 암호화된 모든 데이터 키를 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 필드를 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.

AWS Database Encryption SDK의 리전 검색 키링은 지정된 리전의 KMS 키로만 복호화를 시도합니다. 검색 키링을 사용하는 경우 AWS KMS 클라이언트에서 리전을 구성합니다. 이러한 AWS Database Encryption SDK 구현 AWS KMS 은 KMS 키를 리전별로 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.

예를 들어 다음 코드는 검색 필터를 사용하여 AWS KMS 리전 검색 키링을 생성합니다. 이 키링은 AWS Database Encryption SDK를 미국 서부(오레곤) 리전(us-west-2)의 계정 111122223333에 있는 KMS 키로 제한합니다.

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);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter,
    Regions = us-west-2
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
    .discovery_filter(discovery_filter)
    .regions(us-west-2)
    .send()
    .await?;
```

AWS KMS 계층적 키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

> **Note**
>
> 2023년 7월 24일부터 개발자 시험판 중에 생성된 브랜치 키는 지원되지 않습니다. 새 브랜치 키를 생성하여 개발자 미리 보기 중에 생성한 키 스토어를 계속 사용합니다.

AWS KMS 계층적 키링을 사용하면 레코드를 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 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.

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 캐시에 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 식별되는 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**
>
> AWS Database Encryption SDK의 계층적 키링에 대한 모든 언급은 AWS KMS 계층적 키링을 참조합니다.

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 래핑 키를 생성하는 데 유효한 브랜치 키 구성 요소가 있는지 확인합니다. 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 레코드를 암호화 및 서명합니다. AWS Database Encryption SDK에서 레코드를 암호화하고 서명하는 방법에 대한 자세한 내용은 암호화 및 서명을 참조하세요.

복호화 및 확인

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

1. 복호화 메서드는 암호화된 레코드의 자료 설명 필드에서 암호화된 데이터 키를 식별하고 이를 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 키링은 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 이용해 레코드를 복호화하고 검증하는 방식입니다. AWS Database Encryption SDK에서 레코드를 복호화하고 확인하는 방법에 대한 자세한 내용은 복호화 및 확인을 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 Database Encryption SDK에는가 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 키링을 사용하여 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”

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 스레딩 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}
};
```

Rust

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

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 고치기 위한 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
    }
};
```

Rust

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

StormTracking 캐시

StormTracking은 멀티스레드가 많은 환경을 지원하도록 설계되었습니다. 브랜치 키 자료 항목이 만료되면 StormTracking 캐시는 브랜치 키 자료 항목이 미리 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.

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: `fanOut` 초과 시 스레드가 절전 상태로 유지되는 시간(초)을 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
    }
```

```
};
```

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()?)
```

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 쓰고 있는 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);
```

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?;
```

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 };
```

Rust

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

3. 2단계에서 계층적 키링으로 `sharedCache` 객체를 전달합니다.

공유 캐시를 사용하여 계층적 키링을 생성할 때 선택적으로를 정의`partitionID`하여 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 제한이 600초인 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);
```

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?;
```

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. 기본적으로 파티션 및 정렬 키만 암호화 컨텍스트에 포함됩니다. 그러나 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 암호화 작업을 사용하여 암호화 컨텍스트에 추가 필드를 포함할 수 있습니다.

  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,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

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()
    .branch_key_id(branch_key_id)
    .key_store(branch_key_store_name)
    .ttl_seconds(600)
    .send()
    .await?;
```

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 1단계에서 생성한 두 브랜치 키에 대한 표시 이름을 생성하고를 호출`CreateDynamoDbEncryptionBranchKeyIdSupplier`하여 AWS Database Encryption SDK for DynamoDB 클라이언트를 사용하여 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;
```

Rust

```
// Create friendly names for each branch_key_id
pub struct ExampleBranchKeyIdSupplier {
    branch_key_id_for_tenant1: String,
```

```
    branch_key_id_for_tenant2: String,
}

impl ExampleBranchKeyIdSupplier {
    pub fn new(tenant1_id: &str, tenant2_id: &str) -> Self {
        Self {
            branch_key_id_for_tenant1: tenant1_id.to_string(),
            branch_key_id_for_tenant2: tenant2_id.to_string(),
        }
    }
}

// Create the branch key ID supplier
let dbesdk_config = DynamoDbEncryptionConfig::builder().build()?;
let dbesdk = dbesdk_client::Client::from_conf(dbesdk_config)?;
let supplier = ExampleBranchKeyIdSupplier::new(tenant1_branch_key_id,
 tenant2_branch_key_id);

let branch_key_id_supplier = dbesdk
    .create_dynamo_db_encryption_branch_key_id_supplier()
    .ddb_key_branch_key_id_supplier(supplier)
    .send()
    .await?
    .branch_key_id_supplier
    .unwrap();
```

2. 계층적 키링 생성

   다음 예제에서는 1단계에서 생성한 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);
```

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()
    .branch_key_id_supplier(branch_key_id_supplier)
    .key_store(key_store)
    .ttl_seconds(600)
    .send()
    .await?;
```

검색 가능한 암호화를 위한 계층적 키링 사용

검색 가능한 암호화를 사용하면 전체 데이터베이스를 복호화하지 않고도 암호화된 레코드를 검색할 수 있습니다. 이는 암호화된 필드의 일반 텍스트 값을 비컨으로 인덱싱하여 수행됩니다. 검색 가능한 암호화를 구현하려면 계층적 키링을 사용해야 합니다.

키 스토어 CreateKey 작업은 브랜치 키와 비컨 키를 모두 생성합니다. 브랜치 키는 레코드 암호화 및 복호화 작업에 사용됩니다. 비컨 키는 비컨을 생성하는 데 사용됩니다.

브랜치 키와 비컨 키는 키 스토어 서비스를 생성할 때 지정한 AWS KMS key 것과 동일한 로 보호됩니다. CreateKey 작업이를 호출 AWS KMS 하여 브랜치 키를 생성한 후 kms:GenerateDataKeyWithoutPlaintext를 다시 호출하여 다음 요청을 사용하여 비컨 키를 생성합니다.

```
{
   "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
   },
   "KeyId": "the KMS key ARN",
   "NumberOfBytes": "32"
}
```

두 키를 모두 생성한 후 CreateKey 작업은 ddb:TransactWriteItems를 호출하여 브랜치 키 저장소에 브랜치 키와 비컨 키를 유지하는 두 개의 새 항목을 작성합니다.

표준 비컨을 구성하면 AWS Database Encryption SDK가 키 스토어에서 비컨 키를 쿼리합니다. 그런 다음 HMAC 기반 추출 및 확장 키 파생 함수(HKDF) 를 사용하여 비컨 키를 표준 비컨의 이름과 결합하여 지정된 비컨에 대한 HMAC 키를 생성합니다.

브랜치 키와 달리 키 스토어에는 당 하나의 비컨 키 버전만 branch-key-id 있습니다. 비컨 키는 절대 교체되지 않습니다.

비컨 키 소스 정의하기

표준 및 복합 비컨의 비컨 버전을 정의할 때는 비컨 키를 식별하고 비컨 키 자료에 대한 캐시 제한 수명(TTL)을 정의해야 합니다. 비컨 키 자료는 브랜치 키와는 별도의 로컬 캐시에 저장됩니다. 다음 스니펫은 단일 테넌트 데이터베이스용으로 keySource를 정의하는 방법을 보여줍니다. 연결된 branch-key-id에 의한 비컨 키로 비컨 키를 식별합니다.

Java

```
keySource(BeaconKeySource.builder()
        .single(SingleKeyStore.builder()
```

```
                .keyId(branch-key-id)
                .cacheTTL(6000)
                .build())
        .build())
```

C# / .NET

```
KeySource = new BeaconKeySource
{
    Single = new SingleKeyStore
    {
       KeyId = branch-key-id,
       CacheTTL = 6000
    }
}
```

Rust

```
 .key_source(BeaconKeySource::Single(
    SingleKeyStore::builder()
        // `keyId` references a beacon key.
        // For every branch key we create in the keystore,
        // we also create a beacon key.
        // This beacon key is not the same as the branch key,
        // but is created with the same ID as the branch key.
        .key_id(branch_key_id)
        .cache_ttl(6000)
        .build()?,
))
```

멀티테넌트 데이터베이스의 비컨 소스 정의

멀티테넌트 데이터베이스를 사용하는 경우 keySource를 구성할 때 다음 값을 지정해야 합니다.

- keyFieldName

  지정된 테넌트에 대한 비컨을 생성하는 데 사용된 비컨 키와 branch-key-id 관련된 필드를 저장하는 필드의 이름을 정의합니다. keyFieldName은 임의의 문자열일 수 있지만 데이터베이스의 다른 모든 필드에 고유해야 합니다. 데이터베이스에 새 레코드를 쓰는 경우 해당 레코드에 대한 비컨을 생성하는 데 사용되는 비컨 키를 식별하는 branch-key-id가 이 필드에 저장됩니다.

비컨 쿼리에 이 필드를 포함하고 비컨을 재계산하는 데 필요한 적절한 비컨 키 자료를 식별해야 합니다. 자세한 내용은 멀티테넌트 데이터베이스의 비컨 쿼리 단원을 참조하십시오.

- cacheTTL

  로컬 비컨 캐시 내의 비컨 키 자료 항목이 만료되기 전에 사용할 수 있는 시간(초) 입니다. 이 값은 0보다 커야 합니다. 캐시 제한 TTL이 만료되면 해당 항목이 로컬 캐시에서 제거됩니다.

- (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.

다음 예제에서는 브랜치 키 ID 공급자의 캐시 제한 TLL이 600초이고 입력 용량이 1000인 계층적 키링을 생성합니다.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                        .entryCapacity(1000)
                        .build())
                .build();
final IKeyring 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 = 1000 }
   }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
    let mat_prov = client::Client::from_conf(provider_config)?;
    let kms_keyring = mat_prov
        .create_aws_kms_hierarchical_keyring()
        .branch_key_id(branch_key_id)
        .key_store(key_store)
        .ttl_seconds(600)
        .send()
        .await?;
```

AWS KMS ECDH 키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

Important

AWS KMS ECDH 키링은 Material Providers Library 버전 1.5.0 이상에서만 사용할 수 있습니다.

AWS KMS ECDH 키링은 비대칭 키 계약을 사용하여 두 당사자 간에 공유 대칭 래핑 키를 AWS KMS keys 도출합니다. 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 다

음 키링은 공유 보안 암호를 사용하여 데이터 암호화 키를 보호하는 공유 래핑 키를 파생합니다. AWS Database Encryption SDK가 (KDF_CTR_HMAC_SHA384)를 사용하여 공유 래핑 키를 도출하는 키 도출 함수는 키 도출에 대한 NIST 권장 사항을 준수합니다.

키 파생 함수는 64바이트의 키 지정 구성 요소를 반환합니다. 양 당사자가 올바른 키 구성 요소를 사용하도록 AWS Database Encryption SDK는 처음 32바이트를 커밋 키로 사용하고 마지막 32바이트를 공유 래핑 키로 사용합니다. 복호화 시 키링이 암호화된 레코드의 자료 설명 필드에 저장된 동일한 커밋 키와 공유 래핑 키를 재현할 수 없는 경우 작업이 실패합니다. 예를 들어 Alice의 프라이빗 키와 Bob의 퍼블릭 키로 구성된 키링으로 레코드를 암호화하는 경우 Bob의 프라이빗 키와 Alice의 퍼블릭 키로 구성된 키링은 동일한 커밋 키와 공유 래핑 키를 재현하고 레코드를 해독할 수 있습니다. Bob의 퍼블릭 키가 KMS 키 페어가 아닌 경우 Bob은 원시 ECDH 키링을 생성하여 레코드를 복호화할 수 있습니다.

AWS KMS ECDH 키링은 AES-GCM을 사용하여 대칭 키로 레코드를 암호화합니다. 그런 다음 AES-GCM을 사용하여 파생된 공유 래핑 키로 데이터 키를 봉투 암호화합니다. 각 AWS KMS ECDH 키링에는 하나의 공유 래핑 키만 있을 수 있지만, 다중 키링에는 단독으로 또는 다른 키링과 함께 여러 AWS KMS ECDH 키링을 포함할 수 있습니다.

주제

- AWS KMS ECDH 키링에 필요한 권한
- AWS KMS ECDH 키링 생성
- AWS KMS ECDH 검색 키링 생성

AWS KMS ECDH 키링에 필요한 권한

AWS Database Encryption SDK에는 AWS 계정이 필요하지 않으며 AWS 서비스에 의존하지 않습니다. 그러나 AWS KMS ECDH 키링을 사용하려면 키링의 AWS KMS keys 에 대한 AWS 계정과 다음과 같은 최소 권한이 필요합니다. 권한은 사용하는 키 계약 스키마에 따라 달라집니다.

- `KmsPrivateKeyToStaticPublicKey` 키 계약 스키마를 사용하여 레코드를 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 레코드를 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();
```

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.

```
// 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?;
```

AWS KMS ECDH 검색 키링 생성

복호화할 때는 AWS Database Encryption SDK가 사용할 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 암호화된 레코드의 자료 설명 필드에 저장된 수신자의 퍼블릭 키와 일치하는 모든 레코드를 복호화할 수 있는 AWS KMS ECDH 키링을 생성할 수도 있습니다.

> **Important**
>
> `KmsPublicKeyDiscovery` 키 계약 스키마를 사용하여 레코드를 복호화하는 경우 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 암호화된 레코드의 자료 설명 필드에 저장된 수신자의 퍼블릭 키와 일치하는 모든 레코드를 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 암호화된 레코드의 자료 설명 필드에 저장된 수신자의 퍼블릭 키와 일치하는 모든 레코드를 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();
```

Rust

```
// 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?;
```

Raw AES 키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK를 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.

키 네임스페이스 및 이름

키링에서 AES 키를 식별하기 위해 Raw AES 키링은 사용자가 제공한 키 네임스페이스와 키 이름을 사용합니다. 이 값은 비밀이 아닙니다. AWS Database Encryption SDK가 레코드에 추가하는 자료 설명에 일반 텍스트로 표시됩니다. HSM 또는 키 관리 시스템의 키 네임스페이스와 해당 시스템에서 AES 키를 식별하는 키 이름을 사용하는 것이 좋습니다.

> **Note**
>
> 키 네임스페이스와 키 이름은 `JceMasterKey`의 공급자 ID(또는 공급자) 및 키 ID 필드와 동일합니다.

특정 필드를 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.

Java

```
final CreateRawAesKeyringInput keyringInput = CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

C# / .NET

```
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
var keyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = AESWrappingKey,
```

```
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

var matProv = new MaterialProviders(new MaterialProvidersConfig());
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

Raw RSA 키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

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 페어에 속하는지 확인하세요.

Raw RSA 키링은 RSA 비대칭 암호화 키와 함께 사용될 AWS Encryption SDK for Java 때의 JceMasterKey와 동일하고 상호 작용합니다.

> **Note**
>
> Raw RSA 키링은 비대칭 KMS 키를 지원하지 않습니다. 비대칭 RSA KMS 키를 사용하려면 AWS KMS 키링을 구성합니다.

네임스페이스 및 이름

키링에서 RSA 키 자료를 식별하기 위해 Raw RSA 키링은 사용자가 제공한 키 네임스페이스와 키 이름을 사용합니다. 이 값은 비밀이 아닙니다. AWS Database Encryption SDK가 레코드에 추가하는 자료 설명에 일반 텍스트로 표시됩니다. 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`의 공급자 ID(또는 공급자) 및 키 ID 필드와 동일합니다.

특정 레코드를 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 Java 예제에서는 RSA 키 쌍의 공개 및 개인 키를 사용하여 원시 RSA 키링을 생성하고 SHA-256 패딩 모드를 사용하는 MGF1 및 SHA-256을 사용하는 OAEP를 생성하는 방법을 보여줍니다. `RSAPublicKey` 및 `RSAPrivateKey` 변수는 사용자가 제공하는 키 자료를 나타냅니다.

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);
```

C# / .NET

```
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 keyringInput = new CreateRawRsaKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    PaddingScheme = PaddingScheme.OAEP_SHA512_MGF1,
    PublicKey = publicKey,
```

```
    PrivateKey = privateKey
};

// Create the keyring
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name("RSA_2048_06")
    .key_namespace("HSM_01")
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
    .public_key(RSA_public_key)
    .private_key(RSA_private_key)
    .send()
    .await?;
```

원시 ECDH 키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

Important

원시 ECDH 키링은 Material Providers Library 버전 1.5.0에서만 사용할 수 있습니다.

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. AWS Database Encryption SDK가 (KDF_CTR_HMAC_SHA384)를 사용하여 공유 래핑 키를 도출하는 키 도출 함수는 키 도출에 대한 NIST 권장 사항을 준수합니다.

키 파생 함수는 64바이트의 키 지정 구성 요소를 반환합니다. 양 당사자가 올바른 키 구성 요소를 사용하도록 AWS Database Encryption SDK는 처음 32바이트를 커밋 키로 사용하고 마지막 32바이트를 공유 래핑 키로 사용합니다. 복호화 시 키링이 암호화된 레코드의 자료 설명 필드에 저장된 동일한 커밋 키와 공유 래핑 키를 재현할 수 없는 경우 작업이 실패합니다. 예를 들어 Alice의 프라이빗 키와 Bob의 퍼블릭 키로 구성된 키링으로 레코드를 암호화하는 경우 Bob의 프라이빗 키와 Alice의 퍼블릭 키로 구성된 키링은 동일한 커밋 키와 공유 래핑 키를 재현하고 레코드를 해독할 수 있습니다. Bob의 퍼블릭 키가 페어에서 가져온 AWS KMS key 경우 Bob은 AWS KMS ECDH 키링을 생성하여 레코드를 해독할 수 있습니다.

원시 ECDH 키링은 AES-GCM을 사용하여 대칭 키로 레코드를 암호화합니다. 그런 다음 AES-GCM을 사용하여 파생된 공유 래핑 키로 데이터 키를 봉투 암호화합니다. 각 원시 ECDH 키링에는 하나의 공유 래핑 키만 있을 수 있지만, 단독으로 또는 다른 키링과 함께 여러 원시 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. AWS Database Encryption SDK는 다음과 같은 타원 큐브 사양을 지원합니다.

- `ECC_NIST_P256`
- `ECC_NIST_P384`
- `ECC_NIST_P512`

원시 ECDH 키링 생성

원시 ECDH 키링은 `RawPrivateKeyToStaticPublicKey`, 및 `EphemeralPrivateKeyToStaticPublicKey`의 세 가지 주요 계약 스키마를 지원합니다`PublicKeyDiscovery`. 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 레코드를 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);
}
```

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.

```
// 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?;
```

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 레코드만 암호화할 수 있습니다.
EphemeralPrivateKeyToStaticPublicKey 키 계약 스키마로 암호화된 레코드를 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);
}
```

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.

```
// 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?;
```

PublicKeyDiscovery

복호화할 때는 AWS Database Encryption SDK가 사용할 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 암호화된 레코드의 자료 설명 필드에 저장된 수신자의 퍼블릭 키와 일치하는 모든 레코드를 복호화할 수 있는 원시 ECDH 키링을 생성할 수도 있습니다. 이 키 계약 스키마는 레코드만 복호화할 수 있습니다.

> **Important**
>
> `PublicKeyDiscovery` 키 계약 스키마를 사용하여 레코드를 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 암호화된 레코드의 자료 설명 필드에 저장된 수신자의 퍼블릭 키와 일치하는 모든 레코드를 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 암호화된 레코드의 자료 설명 필드에 저장된 수신자의 퍼블릭 키와 일치하는 모든 레코드를 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);
}
```

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.

```
// 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?;
```

다중 키링

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

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 키링의 생성기 키는 일반 텍스트 키를 생성하고 암호화합니다. 그런 다음 AWS KMS 키링 AWS KMS keys 의 모든 추가 키와 다중 키링의 모든 하위 키링의 모든 래핑 키는 동일한 일반 텍스트 키를 암호화합니다.

복호화할 때 AWS Database Encryption SDK는 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.

다중 키링을 만들려면 먼저 하위 키링을 인스턴스화하세요. 이 예제에서는 AWS KMS 키링과 원시 AES 키링을 사용하지만 지원되는 모든 키링을 다중 키링에 결합할 수 있습니다.

Java

```
// 1. Create 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);

// 2. Create the AWS KMS keyring.
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

C# / .NET

```
// 1. Create the raw AES keyring.
var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

var matProv = new MaterialProviders(new MaterialProvidersConfig());
```

```
var createRawAesKeyringInput = new CreateRawAesKeyringInput
{
    KeyName = "keyName",
    KeyNamespace = "myNamespaces",
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};
var rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// 2. Create the AWS KMS keyring.
//    We create a MRK multi keyring, as this interface also supports
//    single-region KMS keys,
//    and creates the KMS client for us automatically.
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = keyArn
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
// 1. Create the raw AES keyring
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

// 2. Create the AWS KMS keyring
let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(key_arn)
    .send()
    .await?;
```

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.

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);
```

C# / .NET

.NET `CreateMultiKeyringInput` 생성자를 사용하면 생성기 키링과 자식 키링을 정의할 수 있습니다. 결과 `CreateMultiKeyringInput` 객체는 변경할 수 없습니다.

```
var createMultiKeyringInput = new CreateMultiKeyringInput
{
    Generator = awsKmsMrkMultiKeyring,
    ChildKeyrings = new List<IKeyring> { rawAesKeyring }
};
var multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

Rust

```
let multi_keyring = mpl
    .create_multi_keyring()
    .generator(aws_kms_mrk_multi_keyring)
    .child_keyrings(vec![raw_aes_keyring.clone()])
    .send()
    .await?;
```

이제 다중 키링을 사용하여 데이터를 암호화 및 복호화할 수 있습니다.

검색 가능한 암호화

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

검색 가능한 암호화를 사용하면 전체 데이터베이스를 복호화하지 않고도 암호화된 레코드를 검색할 수 있습니다. 이는 필드에 기록된 일반 텍스트 값과 데이터베이스에 실제로 저장된 암호화된 값 사이의 맵을 생성하는 비컨을 사용하여 수행됩니다. AWS Database Encryption SDK는 레코드에 추가하는 새 필드에 비컨을 저장합니다. 사용하는 비컨의 유형에 따라 암호화된 데이터에 대해 정확히 일치하는 검색을 수행하거나 보다 맞춤화된 복합 쿼리를 수행할 수 있습니다.

> **Note**
>
> AWS Database Encryption SDK의 검색 가능한 암호화는 검색 가능한 대칭 암호화와 같이 학술 연구에 정의된 검색 가능한 대칭 암호화와 다릅니다.

비컨은 필드의 일반 텍스트와 암호화된 값 사이에 맵을 생성하는 잘린 해시 기반 메시지 인증 코드(HMAC) 태그입니다. 검색 가능한 암호화를 위해 구성된 암호화된 필드에 새 값을 작성하면 AWS Database Encryption SDK가 일반 텍스트 값을 통해 HMAC를 계산합니다. 이 HMAC 출력은 해당 필드의 일반 텍스트 값과 일대일(1:1) 일치합니다. 여러 개의 고유한 일반 텍스트 값이 잘린 동일한 HMAC 태그에 매핑되도록 HMAC 출력이 잘립니다. 이러한 오탐은 일반 텍스트 값에 대한 구별 정보를 식별하는 권한이 없는 사용자의 능력을 제한합니다. 비컨을 쿼리하면 AWS Database Encryption SDK는 이러한 오탐을 자동으로 필터링하고 쿼리의 일반 텍스트 결과를 반환합니다.

각 비컨에 대해 생성된 평균 오탐 수는 잘린 후 남은 비컨 길이에 따라 결정됩니다. 구현에 적합한 비컨 길이를 결정하는 데 도움이 필요하면 비컨 길이 결정을 참조하세요.

> **Note**
>
> 검색 가능한 암호화는 채워지지 않은 새 데이터베이스에 구현되도록 설계되었습니다. 기존 데이터베이스에 구성된 모든 비컨은 데이터베이스에 업로드된 새 레코드만 매핑하며, 비컨이 기존 데이터를 매핑할 방법은 없습니다.

주제

비컨이 내 데이터 세트에 적합한가?

비컨을 사용하여 암호화된 데이터에 대한 쿼리를 수행하면 클라이언트측 암호화된 데이터베이스와 관련된 성능 비용을 줄일 수 있습니다. 비컨을 사용할 때 쿼리의 효율성과 데이터 분포에 대해 공개되는 정보의 양 사이에는 본질적인 균형이 있습니다. 비컨은 필드의 암호화된 상태를 변경하지 않습니다. AWS Database Encryption SDK로 필드를 암호화하고 서명하면 필드의 일반 텍스트 값이 데이터베이스에 노출되지 않습니다. 데이터베이스는 필드의 무작위화되고 암호화된 값을 저장합니다.

비컨은 비컨이 계산되는 암호화된 필드와 함께 저장됩니다. 즉, 인증되지 않은 사용자가 암호화된 필드의 일반 텍스트 값을 볼 수 없더라도 비컨에 대한 통계 분석을 수행하여 데이터 세트 분포에 대해 자세히 알아보고, 극단적인 경우에는 비컨이 매핑하는 일반 텍스트 값을 식별할 수 있습니다. 비컨을 구성하는 방식으로 이러한 위험을 완화할 수 있습니다. 특히 올바른 비컨 길이를 선택하면 데이터 세트의 기밀을 유지하는 데 도움이 될 수 있습니다.

보안과 성능 비교

- 비컨 길이가 짧을수록 보안이 더 많이 보존됩니다.
- 비컨 길이가 길수록 성능이 더 많이 보존됩니다.

검색 가능한 암호화는 모든 데이터 세트에 대해 원하는 수준의 성능과 보안을 모두 제공하지 못할 수 있습니다. 비컨을 구성하기 전에 위협 모델, 보안 요구 사항 및 성능 요구 사항을 검토하세요.

검색 가능한 암호화가 데이터 세트에 적합한지 결정할 때 다음 데이터 세트 고유성 요구 사항을 고려하세요.

배포

비컨이 보존하는 보안 수준은 데이터 세트의 배포에 따라 달라집니다. 검색 가능한 암호화를 위해 암호화된 필드를 구성하면 AWS Database Encryption SDK는 해당 필드에 기록된 일반 텍스트 값을 통해 HMAC를 계산합니다. 주어진 필드에 대해 계산된 모든 비컨은 동일한 키를 사용하여 계산됩니다. 단, 각 테넌트에 대해 고유한 키를 사용하는 멀티테넌트 데이터베이스는 예외입니다. 즉, 동일한 일반 텍스트 값을 필드에 여러 번 기록하면 해당 일반 텍스트 값의 모든 인스턴스에 대해 동일한 HMAC 태그가 생성됩니다.

매우 일반적인 값을 포함하는 필드에서 비컨을 생성하지 않아야 합니다. 일리노이주의 모든 거주자의 주소를 저장하는 데이터베이스를 예로 들어 보겠습니다. 암호화된 City 필드를 기반으로 비컨을 구성하면 일리노이주 인구 중 시카고에 거주하는 인구가 많기 때문에 "시카고"를 기준으로 계산된 비컨이 과다 표시될 수 있습니다. 인증되지 않은 사용자는 암호화된 값과 비컨 값만 읽을 수 있더라도 비컨이 이 배포를 보존한다면 시카고 거주자에 대한 데이터가 들어 있는 레코드를 식별할 수 있을 것입니다. 배포에 대해 드러나는 식별 정보의 양을 최소화하려면 비컨을 충분히 잘라야 합니다. 이 고르지 않은 분포를 숨기는 데 필요한 비컨 길이로 인해 성능 비용이 많이 들기 때문에 애플리케이션의 요구 사항을 충족하지 못할 수 있습니다.

데이터 세트의 분포를 주의 깊게 분석하여 비컨을 잘라야 하는 정도를 결정해야 합니다. 잘린 후 남은 비컨 길이는 분포에 대해 식별할 수 있는 통계 정보의 양과 직접적인 상관 관계가 있습니다. 데이터 세트에 대해 드러나는 식별 정보의 양을 충분히 최소화하려면 더 짧은 비컨 길이를 선택해야 할 수도 있습니다.

성능과 보안의 균형을 효과적으로 유지하는 고르지 않게 분산된 데이터 집합의 비컨 길이를 계산할 수 없는 경우도 있습니다. 예를 들어, 희귀 질환에 대한 의료 검사 결과를 저장하는 필드에서 비컨을 구성해서는 안 됩니다. NEGATIVE 결과가 데이터셋 내에서 훨씬 더 널리 퍼질 것으로 예상되므로, POSITIVE 결과가 얼마나 희귀한지에 따라 결과를 쉽게 식별할 수 있습니다. 필드에 가능한 값이 두 개뿐인 경우 분포를 숨기기가 매우 어렵습니다. 분포를 숨길 만큼 짧은 비컨 길이를 사용하면 모든 일반 텍스트 값이 동일한 HMAC 태그에 매핑됩니다. 더 긴 비컨 길이를 사용하면 어떤 비컨이 일반 텍스트 POSITIVE 값에 매핑되는지 명확히 알 수 있습니다.

상관관계

상관 관계가 있는 값을 포함한 필드에서 별개의 비컨을 생성하지 않는 것이 좋습니다. 상관 관계가 있는 필드로 구성된 비컨은 각 데이터 세트가 인증되지 않은 사용자에게 배포되는 과정에서 노출되는 정보의 양을 충분히 최소화하기 위해 비컨 길이를 줄여야 합니다. 엔트로피와 상관 관계가 있는 값의 공동 분포 등 데이터 세트를 주의 깊게 분석하여 비컨을 얼마나 잘라야 하는지 결정해야 합니다. 결과 비컨 길이가 성능 요구 사항을 충족하지 못하면 비컨이 데이터 세트에 적합하지 않을 수 있습니다.

예를 들어 우편번호가 한 도시에만 연결될 가능성이 높으므로 City 및 ZIPCode 필드로 분리된 두 개의 비컨을 만들면 안 됩니다. 일반적으로 비컨에서 생성되는 오탐은 승인되지 않은 사용자가 데이터세트에 대한 식별 가능한 정보를 식별하는 능력을 제한합니다. 그러나 City 및 ZIPCode 필드 사이의 상관 관계를 통해 권한이 없는 사용자도 어떤 결과가 오탐인지 쉽게 식별하고 서로 다른 우편 번호를 구별할 수 있습니다.

또한 동일한 일반 텍스트 값을 포함하는 필드에서 비컨을 구성하지 않아야 합니다. 예를 들어, 두 개의 필드는 동일한 값을 가질 가능성이 높으므로 mobilePhone 및 preferredPhone 필드에

서 비컨을 생성해서는 안 됩니다. 두 필드 모두에서 고유한 비컨을 구성하는 경우 AWS Database Encryption SDK는 서로 다른 키 아래에 각 필드에 대한 비컨을 생성합니다. 그 결과 동일한 일반 텍스트 값에 대해 서로 다른 두 개의 HMAC 태그가 생성됩니다. 서로 다른 두 개의 비컨은 동일한 오탐을 가질 가능성이 낮으며, 인증되지 않은 사용자가 서로 다른 전화번호를 구별할 수도 있습니다.

데이터 세트에 상관 관계가 있는 필드가 포함되어 있거나 분포가 고르지 않은 경우에도 비컨 길이를 줄이면 데이터 세트의 기밀성을 유지하는 비컨을 구성할 수 있습니다. 하지만 비컨 길이가 데이터 세트의 모든 고유한 값이 다수의 오탐을 생성하여 데이터 세트에 대해 드러나는 식별 정보의 양을 효과적으로 최소화할 수 있다는 보장은 없습니다. 비컨 길이는 생성된 오탐의 평균 횟수만 추정합니다. 데이터 세트가 고르지 않게 분산될수록 생성되는 평균 오탐 수를 결정할 수 있는 비컨 길이의 효율성이 떨어집니다.

비컨을 구성하는 필드의 분포를 신중하게 고려하고 보안 요구 사항을 충족하기 위해 비컨 길이를 얼마나 줄여야 하는지 생각해야 합니다. 이 장의 다음 주제에서는 비컨이 균일하게 분산되어 있으며 상관 관계가 있는 데이터를 포함하지 않는다고 가정합니다.

검색 가능한 암호화 시나리오

다음의 예제는 간단한 검색 가능한 암호화 솔루션을 보여줍니다. 애플리케이션에서 이 예제에 사용된 예제 필드는 비컨에 대한 배포 및 상관 관계 고유성 권장 사항을 충족하지 않을 수 있습니다. 이 장의 검색 가능한 암호화 개념에 대해 읽을 때 이 예제를 참조용으로 사용할 수 있습니다.

회사의 직원 데이터를 추적하는 `Employees`이라는 데이터베이스를 예제로 들어 보겠습니다. 데이터베이스의 각 레코드에는 EmployeeID, LastName, FirstName, 및 Address라는 필드가 포함되어 있습니다. `Employees` 데이터베이스의 각 필드는 프라이머리 키 `EmployeeID`로 식별됩니다.

다음은 데이터베이스의 일반 텍스트 레코드의 예제입니다.

```
{
    "EmployeeID": 101,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

LastName 및 FirstName 필드를 암호화 작업과 같이 ENCRYPT_AND_SIGN으로 표시한 경우 이러한 필드의 값은 데이터베이스에 업로드되기 전에 로컬로 암호화됩니다. 업로드되는 암호화된 데이터는 완전히 무작위화되며 데이터베이스는 이 데이터를 보호 대상으로 인식하지 않습니다. 일반적인 데이터 입력만 탐지합니다. 즉, 데이터베이스에 실제로 저장되는 레코드는 다음과 같이 보일 수 있습니다.

```
{
    "PersonID": 101,
    "LastName": "1d76e94a2063578637d51371b363c9682bad926cbd",
    "FirstName": "21d6d54b0aaabc411e9f9b34b6d53aa4ef3b0a35",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

데이터베이스에서 LastName 필드가 정확히 일치하는지 쿼리해야 하는 경우 LastName 필드에 기록된 일반 텍스트 값을 데이터베이스에 저장된 암호화된 값에 매핑하도록 LastName이라는 표준 비컨을 구성합니다.

이 비컨은 LastName 필드의 일반 텍스트 값을 기반으로 HMAC를 계산합니다. 각 HMAC 출력은 잘려서 더 이상 일반 텍스트 값과 정확히 일치하지 않습니다. 예를 들어, Jones에 대한 전체 해시와 잘린 해시는 다음과 같이 보일 수 있습니다.

전체 해시

2aa4e9b404c68182562b6ec761fcca5306de527826a69468885e59dc36d0c3f824bdd44cab45526f

잘린 해시

b35099d408c833

표준 비컨을 구성한 후 LastName 필드에서 동등 검색을 수행할 수 있습니다. 예를 들어, Jones에 대해 검색하려는 경우 LastName 비컨을 사용하여 다음 쿼리를 수행합니다.

```
LastName = Jones
```

AWS Database Encryption SDK는 오탐을 자동으로 필터링하고 쿼리의 일반 텍스트 결과를 반환합니다.

비컨

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

비컨은 필드에 기록된 일반 텍스트 값과 데이터베이스에 실제로 저장된 암호화된 값 사이의 맵을 생성하는 잘린 해시 기반 메시지 인증 코드(HMAC) 태그입니다. 비컨은 필드의 암호화된 상태를 변경하지 않습니다. 비컨은 필드의 일반 텍스트 값에 대한 HMAC를 계산하여 암호화된 값과 함께 저장합니다. 이 HMAC 출력은 해당 필드의 일반 텍스트 값과 일대일(1:1) 일치합니다. 여러 개의 고유한 일반 텍스트 값이 잘린 동일한 HMAC 태그에 매핑되도록 HMAC 출력이 잘립니다. 이러한 오탐은 일반 텍스트 값에 대한 구별 정보를 식별하는 권한이 없는 사용자의 능력을 제한합니다.

비컨은 암호화 작업`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`에서 `ENCRYPT_AND_SIGN`, `SIGN_ONLY`또는 로 표시된 필드로만 구성할 수 있습니다. 비컨 자체는 서명되거나 암호화되지 않습니다. `DO_NOTHING`로 표시된 필드로는 비컨을 구성할 수 없습니다.

구성하는 비컨의 유형에 따라 수행할 수 있는 쿼리 유형이 결정됩니다. 검색 가능한 암호화를 지원하는 두 가지 유형의 비컨이 있습니다. 표준 비컨은 동등 검색을 수행합니다. 복합 비컨은 기본적 일반 텍스트 문자열과 표준 비컨을 결합하여 복잡한 데이터베이스 작업을 수행합니다. 비컨을 구성한 후 암호화된 필드를 검색하려면 먼저 각 비컨에 대한 보조 인덱스를 구성해야 합니다. 자세한 내용은 비컨을 사용한 보조 인덱스 구성 단원을 참조하십시오.

주제

- 표준 비컨
- 복합 비컨

표준 비컨

표준 비컨은 데이터베이스에서 검색 가능한 암호화를 구현하는 가장 간단한 방법입니다. 암호화된 필드 또는 가상 필드 하나에 대해서만 동등 검색을 수행할 수 있습니다. 표준 비컨을 구성하는 방법을 알아보려면 표준 비컨 구성을 참조하세요.

표준 비컨을 구성하는 필드를 비컨 소스라고 합니다. 비컨이 매핑해야 하는 데이터의 위치를 식별합니다. 비컨 소스는 암호화된 필드 또는 가상 필드일 수 있습니다. 각 표준 비컨의 비컨 소스는 고유해야 합니다. 동일한 비컨 소스로 두 개의 비컨을 구성할 수 없습니다.

표준 비컨을 사용하여 암호화된 필드 또는 가상 필드를 동등하게 검색할 수 있습니다. 또는 복합 비컨을 구성하여 더 복잡한 데이터베이스 작업을 수행하는 데 사용할 수 있습니다. 표준 비컨을 구성하고 관리하는 데 도움이 되도록 AWS Database Encryption SDK는 표준 비컨의 용도를 정의하는 다음과 같은 선택적 비컨 스타일을 제공합니다. 자세한 내용은 비컨 스타일 정의를 참조하세요.

암호화된 단일 필드에 대해 등식 검색을 수행하는 표준 비컨을 생성하거나 가상 필드를 생성하여 여러 `ENCRYPT_AND_SIGN`, `SIGN_ONLY`및 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 필드의 연결에 대해 등식 검색을 수행하는 표준 비컨을 생성할 수 있습니다.

가상 필드

가상 필드는 하나 이상의 소스 필드로 구성된 개념적 필드입니다. 가상 필드를 생성해도 레코드에 새 필드가 기록되지는 않습니다. 가상 필드는 데이터베이스에 명시적으로 저장되지 않습니다. 표준 비컨 구성에서 필드의 특정 세그먼트를 식별하는 방법 또는 레코드 내의 여러 필드를 연결하여 특정 쿼리를 수행하는 방법에 대한 지침을 비컨에 제공하는 데 사용됩니다. 가상 필드에는 하나 이상의 암호화된 필드가 필요합니다.

> **Note**
>
> 다음 예제는 가상 필드로 수행할 수 있는 변환 및 쿼리 유형을 보여줍니다. 애플리케이션에서 이 예제에 사용된 예제 필드는 비컨에 대한 배포 및 상관 관계 고유성 권장 사항을 충족하지 않을 수 있습니다.

예를 들어, `FirstName` 및 `LastName` 필드의 연결에 대해 동등 검색을 수행하려는 경우 다음 가상 필드 중 하나를 만들 수 있습니다.

- `FirstName` 필드의 첫 번째 문자와 그 뒤에 `LastName` 필드가 오는 가상 `NameTag` 필드(모두 소문자)입니다. 이 가상 필드를 사용하면 `NameTag=mjones`을 쿼리할 수 있습니다.
- `LastName` 필드와 그 뒤에 `FirstName` 필드로 구성된 가상 `LastFirst` 필드입니다. 이 가상 필드를 사용하면 `LastFirst=JonesMary`을 쿼리할 수 있습니다.

또는 암호화된 필드의 특정 세그먼트에서 동등 검색을 수행하려는 경우 쿼리하려는 세그먼트를 식별하는 가상 필드를 만드세요.

예를 들어 IP 주소의 처음 세 세그먼트를 사용하여 암호화된 `IPAddress` 필드를 쿼리하려면 다음 가상 필드를 만듭니다.

- `Segments('.', 0, 3)`로 구성된 가상 `IPSegment` 필드. 이 가상 필드를 사용하면 `IPSegment=192.0.2`을 쿼리할 수 있습니다. 쿼리는 `IPAddress` 값이 "192.0.2"로 시작하는 모든 레코드를 반환합니다.

가상 필드는 고유해야 합니다. 정확히 동일한 소스 필드로 두 개의 가상 필드를 구성할 수는 없습니다.

가상 필드 및 가상 필드를 사용하는 비컨을 구성하는 데 도움이 필요하면 가상 필드 만들기를 참조하세요.

복합 비컨

복합 비컨은 쿼리 성능을 향상시키고 더 복잡한 데이터베이스 작업을 수행할 수 있도록 인덱스를 생성합니다. 복합 비컨을 사용하여 리터럴 일반 텍스트 문자열과 표준 비컨을 결합하여 암호화된 레코드에 대해 복잡한 쿼리(예: 단일 인덱스에서 서로 다른 두 레코드 유형을 쿼리하거나 정렬 키로 필드 조합을 쿼리하는 등)를 수행할 수 있습니다. 복합 비컨 솔루션 예제에 대한 자세한 내용은 비컨 유형 선택을 참조하세요.

복합 비컨은 표준 비컨 또는 표준 비컨과 서명된 필드의 조합으로 구성할 수 있습니다. 부분 목록으로 구성됩니다. 모든 복합 비컨에는 비컨에 포함된 `ENCRYPT_AND_SIGN` 필드를 식별하는 암호화된 부분 목록이 포함되어야 합니다. 모든 `ENCRYPT_AND_SIGN` 필드는 표준 비컨으로 식별되어야 합니다. 더 복잡한 복합 비컨에는 비컨에 포함된 일반 텍스트 `SIGN_ONLY` 또는 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 필드를 식별하는 서명된 부분 목록과 복합 비컨이 필드를 조합할 수 있는 가능한 모든 방법을 식별하는 생성자 부분 목록이 포함될 수도 있습니다.

> **Note**
>
> AWS Database Encryption SDK는 일반 텍스트 `SIGN_ONLY` 및 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 필드에서 완전히 구성할 수 있는 서명된 비컨도 지원합니다. 서명된 비컨은 서명되었지만 암호화되지 않은 필드에 대해 복잡한 쿼리를 인덱싱하고 수행하는 복합 비컨의 한 유형입니다. 자세한 내용은 서명된 비컨 만들기 단원을 참조하십시오.

복합 비컨 구성에 대한 도움말은 복합 비콘 구성을 참조하세요.

복합 비컨을 구성하는 방식에 따라 수행할 수 있는 쿼리 유형이 결정됩니다. 예를 들어, 일부 암호화되고 서명된 부분을 선택 사항으로 설정하여 쿼리의 유연성을 높일 수 있습니다. 복합 비컨이 수행할 수 있는 쿼리 유형에 대한 자세한 내용은 비컨 쿼리 섹션을 참조하세요.

비컨 계획 수립

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

비컨은 채워지지 않은 새 데이터베이스에 구현되도록 설계되었습니다. 기존 데이터베이스에 구성된 모든 비컨은 데이터베이스에 기록된 새 레코드만 매핑합니다. 비컨은 필드의 일반 텍스트 값에서 계산됩니다. 필드가 암호화되면 비컨이 기존 데이터를 매핑할 방법이 없습니다. 비컨으로 새 레코드를 작성한 후에는 비컨의 구성을 업데이트할 수 없습니다. 하지만 레코드에 추가하는 새 필드에 대해 새 비컨을 추가할 수 있습니다.

검색 가능한 암호화를 구현하려면 AWS KMS 계층 키링을 사용하여 레코드 보호에 사용되는 데이터 키를 생성, 암호화 및 복호화해야 합니다. 자세한 내용은 검색 가능한 암호화를 위한 계층적 키링 사용 단원을 참조하십시오.

검색 가능한 암호화를 위한 비컨을 구성하려면 먼저 암호화 요구 사항, 데이터베이스 액세스 패턴 및 위협 모델을 검토하여 데이터베이스에 가장 적합한 솔루션을 결정해야 합니다.

구성하는 비컨 유형에 따라 수행할 수 있는 쿼리 유형이 결정됩니다. 표준 비컨 구성에서 지정하는 비컨 길이에 따라 해당 비컨에 대해 생성되는 예상 오탐 수가 결정됩니다. 비컨을 구성하기 전에 수행해야 하는 쿼리 유형을 식별하고 계획하는 것이 좋습니다. 비컨을 사용한 후에는 구성을 업데이트할 수 없습니다.

비컨을 구성하기 전에 다음 작업을 검토하고 완료하는 것이 좋습니다.

- 비컨이 데이터 세트에 적합한지 판단
- 비컨 유형 선택
- 비컨 길이 선택
- 비컨 이름 선택

데이터베이스의 검색 가능한 암호화 솔루션을 계획할 때 다음 비컨 고유성 요구 사항을 기억합니다.

- 모든 표준 비컨에는 고유한 비컨 소스가 있어야 합니다.

  동일한 암호화된 필드나 가상 필드에서 여러 표준 비컨을 구성할 수 없습니다.

  하지만 단일 표준 비컨을 사용하여 여러 복합 비컨을 구성할 수 있습니다.

- 소스 필드가 기존 표준 비컨과 겹치는 가상 필드를 만들지 마세요.

  다른 표준 비컨을 만드는 데 사용된 소스 필드가 포함된 가상 필드에서 표준 비컨을 구성하면 두 비컨의 보안이 저하될 수 있습니다.

  자세한 내용은 가상 필드에 대한 보안 고려 사항 단원을 참조하십시오.

멀티테넌트 데이터베이스 고려 사항

멀티테넌트 데이터베이스에 구성된 비컨을 쿼리하려면 레코드를 암호화한 테넌트와 관련된 `branch-key-id`을 쿼리에 저장하는 필드를 포함해야 합니다. 비컨 키 소스를 정의할 때 이 필드를 정의합니다. 쿼리가 성공하려면 이 필드의 값이 비컨을 재계산하는 데 필요한 적절한 비컨 키 자료를 식별해야 합니다.

비컨을 구성하기 전에 쿼리의 `branch-key-id`에 비컨을 어떻게 포함할지 결정해야 합니다. 쿼리에 `branch-key-id`을 포함할 수 있는 다양한 방법에 대한 자세한 내용은 멀티테넌트 데이터베이스의 비컨 쿼리 섹션을 참조하세요.

비컨 유형 선택

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

검색 가능한 암호화를 사용하면 암호화된 필드의 일반 텍스트 값을 비컨으로 매핑하여 암호화된 레코드를 검색할 수 있습니다. 구성하는 비컨 유형에 따라 수행할 수 있는 쿼리 유형이 결정됩니다.

비컨을 구성하기 전에 수행해야 하는 쿼리 유형을 식별하고 계획하는 것이 좋습니다. 비컨을 구성한 후 암호화된 필드를 검색하려면 먼저 각 비컨에 대한 보조 인덱스를 구성해야 합니다. 자세한 내용은 비컨을 사용한 보조 인덱스 구성 단원을 참조하십시오.

비컨은 필드에 기록된 일반 텍스트 값과 데이터베이스에 실제로 저장된 암호화된 값 사이의 맵을 만듭니다. 두 표준 비컨에 동일한 기본 일반 텍스트가 포함되어 있더라도 두 표준 비컨의 값을 비교할 수는 없습니다. 두 개의 표준 비컨은 동일한 일반 텍스트 값에 대해 서로 다른 두 개의 HMAC 태그를 생성합니다. 따라서 표준 비컨은 다음 쿼리를 수행할 수 없습니다.

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)

- CONTAINS(*beacon1*, *beacon2*)

복합 비컨의 서명된 부분을 비교하는 경우에만 위의 쿼리를 수행할 수 있습니다. 단, 복합 비컨과 함께 사용하여 조합된 비컨에 포함된 암호화되거나 서명된 필드의 전체 값을 식별할 수 있는 CONTAINS 연산자는 예외입니다. 서명된 부분을 비교할 때 선택적으로 암호화된 부분의 접두사를 포함할 수 있지만 필드의 암호화된 값은 포함할 수 없습니다. 표준 및 복합 비컨이 수행할 수 있는 쿼리 유형에 대한 자세한 내용은 비컨 쿼리를 참조하세요.

데이터베이스 액세스 패턴을 검토할 때 다음과 같은 검색 가능한 암호화 솔루션을 고려하세요. 다음 예제는 다양한 암호화 및 쿼리 요구 사항을 충족하도록 구성할 비컨을 정의합니다.

표준 비컨

표준 비컨은 평등 검색만 수행할 수 있습니다. 표준 비컨을 사용하여 다음 쿼리를 수행할 수 있습니다.

암호화된 단일 필드 쿼리

암호화된 필드의 특정 값이 포함된 레코드를 식별하려면 표준 비컨을 만드세요.

예제

다음 예제에서는 프로덕션 시설에 대한 검사 데이터를 추적하는 UnitInspection라는 이름의 데이터베이스를 고려합니다. 데이터베이스의 각 레코드에는 work_id, inspection_date, inspector_id_last4, 및 unit라는 필드가 있습니다. 전체 검사기 ID는 0~99,999,999 사이의 숫자입니다. 하지만 데이터세트가 균일하게 분포되도록 하기 위해 inspector_id_last4에는 검사기 ID의 마지막 4자리만 저장됩니다. 데이터베이스의 각 필드는 프라이머리 키 work_id로 식별됩니다. inspector_id_last4 및 unit 필드는 암호화 작업에서 ENCRYPT_AND_SIGN으로 표시됩니다.

다음은 UnitInspection 데이터베이스의 일반 텍스트 항목의 예입니다.

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

레코드의 단일 암호화된 필드 쿼리

inspector_id_last4 필드를 암호화해야 하지만 정확히 일치하는지 쿼리해야 하는 경우 inspector_id_last4 필드에서 표준 비컨을 생성합니다. 그런 다음 표준 비컨을 사용하여 보조

인덱스를 만듭니다. 이 보조 인덱스를 사용하여 암호화된 `inspector_id_last4` 필드를 쿼리할 수 있습니다.

표준 비컨 구성에 대한 도움말은 표준 비컨 구성을 참조하세요.

가상 필드 쿼리

가상 필드는 하나 이상의 소스 필드로 구성된 개념적 필드입니다. 암호화된 필드의 특정 세그먼트에 대해 동등 검색을 수행하거나 여러 필드의 연결에 대해 동등 검색을 수행하려면 가상 필드에서 표준 비컨을 구성합니다. 모든 가상 필드는 하나 이상의 암호화된 소스 필드를 포함해야 합니다.

예제

다음 예제는 `Employees` 데이터베이스의 가상 필드를 만듭니다. 다음은 `Employees` 데이터베이스의 일반 텍스트 레코드의 예제입니다.

```
{
    "EmployeeID": 101,
    "SSN": 000-00-0000,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

암호화된 필드의 세그먼트 쿼리

이 예제에서는 SSN 필드가 암호화됩니다.

사회보장번호의 마지막 4자리를 사용하여 SSN 필드를 쿼리하려면 쿼리하려는 세그먼트를 식별하는 가상 필드를 만드십시오.

`Last4SSN`로 구성된 가상 `Suffix(4)` 필드를 사용하면 `Last4SSN=0000`를 쿼리할 수 있습니다. 이 가상 필드를 사용하여 표준 비컨을 구성합니다. 그런 다음 표준 비컨을 사용하여 보조 인덱스를 만듭니다. 이 보조 인덱스를 사용하여 가상 필드를 쿼리할 수 있습니다. 이 쿼리는 지정한 마지막 4자리 숫자로 끝나는 SSN 값을 가진 모든 레코드를 반환합니다.

여러 필드의 연결 쿼리

> **Note**
>
> 다음 예제는 가상 필드로 수행할 수 있는 변환 및 쿼리 유형을 보여줍니다. 애플리케이션에서 이 예제에 사용된 예제 필드는 비컨에 대한 배포 및 상관 관계 고유성 권장 사항을 충족하지 않을 수 있습니다.

`FirstName` 및 `LastName` 필드 연결에 대해 동일 검색을 수행하려는 경우 `FirstName` 필드의 첫 번째 문자와 그 뒤에 오는 `LastName` 필드(모두 소문자)로 구성되는 가상 `NameTag` 필드를 생성할 수 있습니다. 이 가상 필드를 사용하여 표준 비컨을 구성합니다. 그런 다음 표준 비컨을 사용하여 보조 인덱스를 만듭니다. 이 보조 인덱스를 사용하여 가상 필드에 대한 `NameTag=mjones`를 쿼리할 수 있습니다.

원본 필드 중 하나 이상을 암호화해야 합니다. `FirstName` 또는 `LastName` 둘 중 하나를 암호화하거나 둘 다 암호화할 수 있습니다. 모든 일반 텍스트 소스 필드는 암호화 작업`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`에서 `SIGN_ONLY` 또는 로 표시되어야 합니다.

가상 필드 및 가상 필드를 사용하는 비컨을 구성하는 데 도움이 필요하면 가상 필드 만들기를 참조하세요.

복합 비컨

복합 비컨은 기본적 일반 텍스트 문자열과 표준 비컨에서 인덱스를 생성하여 복잡한 데이터베이스 작업을 수행합니다. 복합 비컨을 사용하여 다음 쿼리를 수행할 수 있습니다.

단일 인덱스에서 암호화된 필드 조합 쿼리

단일 인덱스에서 암호화된 필드 조합을 쿼리해야 하는 경우, 암호화된 각 필드에 대해 구성된 개별 표준 비컨을 결합하여 단일 인덱스를 형성하는 복합 비컨을 만듭니다.

복합 비컨을 구성한 후에는 복합 비컨을 파티션 키로 지정하여 정확히 일치 쿼리를 수행하거나 정렬 키를 사용하여 더 복잡한 쿼리를 수행하는 보조 인덱스를 만들 수 있습니다. 복합 비컨을 정렬 키로 지정하는 보조 인덱스는 정확히 일치하는 쿼리와 보다 맞춤화된 복합 쿼리를 수행할 수 있습니다.

예제

다음 예제에서는 프로덕션 시설에 대한 검사 데이터를 추적하는 UnitInspection라는 이름의 데이터베이스를 고려합니다. 데이터베이스의 각 레코드에는 work_id, inspection_date, inspector_id_last4, 및 unit라는 필드가 있습니다. 전체 검사기 ID는 0~99,999,999 사이의 숫자입니다. 하지만 데이터세트가 균일하게 분포되도록 하기 위해 inspector_id_last4에는 검사기 ID의 마지막 4자리만 저장됩니다. 데이터베이스의 각 필드는 프라이머리 키 work_id로 식별됩니다. inspector_id_last4 및 unit 필드는 암호화 작업에서 ENCRYPT_AND_SIGN으로 표시됩니다.

다음은 UnitInspection 데이터베이스의 일반 텍스트 항목의 예입니다.

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

암호화된 필드 조합에서 동등 검색 수행

inspector_id_last4.unit에서 정확히 일치하는 항목을 UnitInspection 데이터베이스에 쿼리하려면 먼저 inspector_id_last4 및 unit 필드에 대해 고유한 표준 비컨을 만듭니다. 그런 다음 두 개의 표준 비컨으로 복합 비컨을 만듭니다.

복합 비컨을 구성한 후 복합 비컨을 파티션 키로 지정하는 보조 인덱스를 만듭니다. 이 보조 인덱스를 사용하여 inspector_id_last4.unit에 대해 정확히 일치하는 항목을 쿼리할 수 있습니다. 예를 들어, 이 비컨을 쿼리하여 검사기가 특정 단위에 대해 수행한 검사 목록을 찾을 수 있습니다.

암호화된 필드 조합에 대해 복잡한 쿼리 수행

inspector_id_last4 및 inspector_id_last4.unit에서 UnitInspection 데이터베이스를 쿼리하려면 먼저 inspector_id_last4 및 unit 필드에 대해 고유한 표준 비컨을 만듭니다. 그런 다음 두 개의 표준 비컨으로 복합 비컨을 만듭니다.

복합 비컨를 구성한 후 복합 비컨을 정렬 키로 지정하는 보조 인덱스를 만듭니다. 이 보조 인덱스를 사용하여 특정 검사기로 시작하는 항목을 UnitInspection 데이터베이스에 쿼리하거나 특정 검사기에서 검사한 특정 장치 ID 범위 내의 모든 장치 목록을 데이터베이스에 쿼리할 수 있습니다. inspector_id_last4.unit에서 정확히 일치하는 검색을 수행할 수도 있습니다.

복합 비컨 구성에 대한 도움말은 복합 비콘 구성을 참조하세요.

단일 인덱스에서 암호화된 필드와 일반 텍스트 필드의 조합 쿼리

단일 인덱스에서 암호화된 필드와 일반 텍스트 필드를 조합하여 쿼리해야 하는 경우 개별 표준 비컨과 일반 텍스트 필드를 결합하여 단일 인덱스를 형성하는 복합 비컨을 만듭니다. 복합 비컨을 구성하는 데 사용되는 일반 텍스트 필드는 암호화 작업SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT에서 SIGN_ONLY 또는 로 표시되어야 합니다.

복합 비컨을 구성한 후에는 복합 비컨을 파티션 키로 지정하여 정확히 일치 쿼리를 수행하거나 정렬 키를 사용하여 더 복잡한 쿼리를 수행하는 보조 인덱스를 만들 수 있습니다. 복합 비컨을 정렬 키로 지정하는 보조 인덱스는 정확히 일치하는 쿼리와 보다 맞춤화된 복합 쿼리를 수행할 수 있습니다.

예제

다음 예제에서는 프로덕션 시설에 대한 검사 데이터를 추적하는 UnitInspection라는 이름의 데이터베이스를 고려합니다. 데이터베이스의 각 레코드에는 work_id, inspection_date, inspector_id_last4, 및 unit라는 필드가 있습니다. 전체 검사기 ID는 0~99,999,999 사이의 숫자입니다. 하지만 데이터세트가 균일하게 분포되도록 하기 위해 inspector_id_last4에는 검사기 ID의 마지막 4자리만 저장됩니다. 데이터베이스의 각 필드는 프라이머리 키 work_id로 식별됩니다. inspector_id_last4 및 unit 필드는 암호화 작업에서 ENCRYPT_AND_SIGN으로 표시됩니다.

다음은 UnitInspection 데이터베이스의 일반 텍스트 항목의 예입니다.

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

필드 조합에서 동등 검색 수행

특정 검사관이 특정 날짜에 실시한 검사에 대해 UnitInspection 데이터베이스를 쿼리하려면 먼저 inspector_id_last4 필드에 대한 표준 비컨을 만듭니다. 이 inspector_id_last4 필드는 암호화 작업에 ENCRYPT_AND_SIGN로 표시됩니다. 암호화된 모든 부분에는 자체 표준 비컨이 필요합니다. inspection_date 필드는 SIGN_ONLY로 표시가 되어 있으며 표준 비컨이 필요하지 않습니다. 다음으로, inspection_date 필드와 inspector_id_last4 표준 비컨에서 복합 비컨을 만듭니다.

복합 비컨을 구성한 후 복합 비컨을 파티션 키로 지정하는 보조 인덱스를 만듭니다. 이 보조 인덱스를 사용하여 데이터베이스에서 특정 검사기 및 검사 날짜와 정확히 일치하는 레코드를 쿼리할 수

있습니다. 예를 들어, ID가 8744로 끝나는 검사기가 특정 날짜에 실시한 모든 검사 목록을 데이터베이스에 쿼리할 수 있습니다.

필드 조합에 대해 복잡한 쿼리 수행

`inspection_date` 범위 내에서 수행된 검사에 대해 데이터베이스를 쿼리하거나 `inspector_id_last4` 또는 `inspector_id_last4.unit`로 제한된 특정 `inspection_date`에서 수행된 검사에 대해 데이터베이스를 쿼리하려면 먼저 `inspector_id_last4` 및 `unit` 필드에 대해 별도의 표준 비컨을 생성합니다. 그런 다음 일반 텍스트 `inspection_date` 필드와 두 개의 표준 비컨을 사용하여 복합 비컨을 만듭니다.

복합 비컨를 구성한 후 복합 비컨을 정렬 키로 지정하는 보조 인덱스를 만듭니다. 이 보조 인덱스를 사용하여 특정 검사기가 특정 날짜에 실시한 검사에 대한 쿼리를 수행할 수 있습니다. 예를 들어 같은 날짜에 검사한 모든 단위의 목록을 데이터베이스에 쿼리할 수 있습니다. 또는 지정된 검사 날짜 범위 사이에 특정 단위에 대해 수행된 모든 검사 목록을 데이터베이스에 쿼리할 수 있습니다.

복합 비컨 구성에 대한 도움말은 복합 비콘 구성을 참조하세요.

비컨 길이 선택

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

검색 가능한 암호화를 위해 구성된 암호화된 필드에 새 값을 작성하면 AWS Database Encryption SDK가 일반 텍스트 값을 통해 HMAC를 계산합니다. 이 HMAC 출력은 해당 필드의 일반 텍스트 값과 일대일(1:1) 일치합니다. 여러 개의 고유한 일반 텍스트 값이 잘린 동일한 HMAC 태그에 매핑되도록 HMAC 출력이 잘립니다. 이러한 충돌 또는 오탐은 일반 텍스트 값에 대한 구별 정보를 식별하는 권한이 없는 사용자의 능력을 제한합니다.

각 비컨에 대해 생성된 평균 오탐 수는 잘린 후 남은 비컨 길이에 따라 결정됩니다. 표준 비컨을 구성할 때 비컨 길이만 정의하면 됩니다. 복합 비컨은 구성된 표준 비컨의 비컨 길이를 사용합니다.

비컨은 필드의 암호화된 상태를 변경하지 않습니다. 그러나 비컨을 사용할 때 쿼리의 효율성과 데이터 분포에 대해 공개되는 정보의 양 사이에는 본질적인 균형이 있습니다.

검색 가능한 암호화의 목표는 비컨을 사용하여 암호화된 데이터에 대한 쿼리를 수행함으로써 클라이언트측 암호화된 데이터베이스와 관련된 성능 비용을 줄이는 것입니다. 비컨은 비컨이 계산되는 암호화된 필드와 함께 저장됩니다. 이는 데이터 세트의 분포에 대한 구별되는 정보를 공개할 수 있음을 의

미합니다. 극단적인 경우 권한이 없는 사용자가 배포에 대해 공개된 정보를 분석하고 이를 사용하여 필드의 일반 텍스트 값을 식별할 수 있습니다. 올바른 비컨 길이를 선택하면 이러한 위험을 완화하고 배포의 기밀성을 유지하는 데 도움이 될 수 있습니다.

위협 모델을 검토하여 필요한 보안 수준을 결정합니다. 예를 들어, 데이터베이스에 액세스할 수 있지만 일반 텍스트 데이터에 액세스할 수 없는 개인이 많을수록 데이터 세트 배포의 기밀성을 더 많이 보호해야 할 수 있습니다. 기밀성을 높이려면 비컨이 더 많은 오탐을 생성해야 합니다. 기밀성이 높아지면 쿼리 성능이 저하됩니다.

보안과 성능 비교

- 비컨 길이가 너무 길면 오탐이 너무 적어 데이터 세트 분포에 대한 구별 정보가 공개될 수 있습니다.
- 비컨 길이가 너무 짧으면 오탐이 너무 많이 발생하고 데이터베이스를 더 광범위하게 검색해야 하므로 쿼리 성능 비용이 증가합니다.

솔루션에 적합한 비컨 길이를 결정할 때 꼭 필요한 것 이상으로 쿼리 성능에 영향을 주지 않고 데이터 보안을 적절하게 유지하는 길이를 찾아야 합니다. 비컨에 의해 유지되는 보안 수준은 데이터 세트의 분포와 비컨이 구성된 필드의 상관 관계에 따라 달라집니다. 다음 주제에서는 비컨이 균일하게 분포되어 있고 상관된 데이터를 포함하지 않는다고 가정합니다.

주제

- 비컨 길이 계산
- 예제

비컨 길이 계산

비컨 길이는 비트 단위로 정의되며 잘린 후에도 유지되는 HMAC 태그의 비트 수를 나타냅니다. 권장되는 비컨 길이는 데이터 세트 분포, 상관 값의 존재, 특정 보안 및 성능 요구 사항에 따라 다릅니다. 데이터 세트가 균일하게 분포된 경우 다음 방정식과 절차를 사용하여 구현에 가장 적합한 비컨 길이를 식별하는 데 도움이 될 수 있습니다. 이러한 방정식은 비컨이 생성할 평균 오탐 수만 추정할 뿐 데이터 세트의 모든 고유 값이 특정 개수의 오탐을 생성한다고 보장하지는 않습니다.

> **Note**
>
> 이러한 방정식의 효율성은 데이터 세트의 분포에 따라 달라집니다. 데이터세트가 균일하게 분포되지 않은 경우 비컨이 내 데이터 세트에 적합한가? 섹션을 참조하세요.
> 일반적으로 데이터 세트가 균일한 분포에서 멀어질수록 비컨 길이를 더 줄여야 합니다.

1.

모집단 추정

모집단은 표준 비컨을 구성하는 필드의 예상 고유 값 수이며, 필드에 저장된 총 예상 값 수가 아닙니다. 예를 들어 직원 회의 위치를 식별하는 암호화된 Room 필드를 고려하세요. Room 필드에는 총 100,000개의 값이 저장될 것으로 예상되지만 직원이 회의를 위해 예약할 수 있는 공간은 50개뿐입니다. 즉, Room 필드에 저장할 수 있는 고유 값은 50개뿐이므로 모집단은 50개입니다.

> **Note**
>
> 표준 비컨이 가상 필드에서 구성된 경우 비컨 길이를 계산하는 데 사용되는 인구는 가상 필드에서 생성된 고유 조합의 수입니다.

모집단을 추정할 때 데이터 세트의 예상 증가를 고려해야 합니다. 비컨으로 새 레코드를 작성한 후에는 비컨 길이를 업데이트할 수 없습니다. 위협 모델과 기존 데이터베이스 솔루션을 검토하여 향후 5년 동안 이 필드에 저장할 것으로 예상되는 고유 값의 수에 대한 추정치를 생성합니다.

모집단은 정확할 필요는 없습니다. 먼저, 현재 데이터베이스의 고유 값 수를 식별하거나 첫 해에 저장할 것으로 예상되는 고유 값 수를 추정합니다. 다음으로, 다음 질문을 사용하여 향후 5년 동안 고유한 가치의 예상 성장을 결정하는 데 도움을 받으세요.

- 고유한 값에 10이 곱해질 것이라고 예상하시나요?
- 고유한 값에 100이 곱해질 것이라고 예상하시나요?
- 고유한 값에 1000이 곱해질 것이라고 예상하시나요?

50,000개와 60,000개의 고유 값 사이의 차이는 중요하지 않으며 둘 다 동일한 권장 비컨 길이가 됩니다. 그러나 50,000과 500,000의 고유 값 사이의 차이는 권장 비컨 길이에 큰 영향을 미칩니다.

우편번호나 성 등 일반적인 데이터 유형의 빈도에 대한 공개 데이터를 검토하는 것이 좋습니다. 예를 들어, 미국에는 41,707개의 우편번호가 있습니다. 사용하는 모집단은 자신의 데이터베이스에 비례해야 합니다. 데이터베이스의 `ZIPCode` 필드에 미국 전역의 데이터가 포함되어 있는 경우 `ZIPCode` 필드에 현재 41,707개의 고유 값이 없더라도 모집단을 41,707로 정의할 수 있습니다. 데이터베이스의 `ZIPCode` 필드에 단일 주의 데이터만 포함되고 향후에도 단일 주의 데이터만을 포함하는 경우 모집단을 41,704가 아닌 해당 주의 총 우편 번호 수로 정의할 수 있습니다.

2. 예상 충돌 횟수에 대한 권장 범위 계산

특정 필드에 대한 적절한 비컨 길이를 결정하려면 먼저 예상되는 충돌 횟수에 대한 적절한 범위를 식별해야 합니다. 예상 충돌 수는 특정 HMAC 태그에 매핑되는 고유 일반 텍스트 값의 평균 예상 수를 나타냅니다. 하나의 고유한 일반 텍스트 값에 대해 예상되는 오탐 수는 예상되는 충돌 수보다 1이 적습니다.

예상되는 충돌 횟수는 2보다 크거나 같고 인구의 제곱근보다 작은 것이 좋습니다. 다음 방정식은 모집단에 16개 이상의 고유 값이 있는 경우에만 작동합니다.

```
2 ≤ number of collisions < √(Population)
```

충돌 횟수가 2회 미만이면 비컨은 너무 적은 양의 오탐을 생성합니다. 평균적으로 필드의 모든 고유 값이 하나의 다른 고유 값에 매핑되어 최소한 하나의 오탐을 생성한다는 의미이므로 예상되는 최소 충돌 수로 2를 권장합니다.

3. 비컨 길이에 대한 권장 범위 계산

   예상되는 충돌의 최소 및 최대 횟수를 식별한 후 다음 방정식을 사용하여 적절한 비컨 길이의 범위를 식별합니다.

   $$\text{number of collisions} = \text{Population} * 2^{-(\text{beacon length})}$$

   먼저, 예상 충돌 횟수가 2(예상 충돌의 최소 권장 횟수)인 비컨 길이를 구합니다.

   $$2 = \text{Population} * 2^{-(\text{beacon length})}$$

   그런 다음 예상되는 충돌 횟수가 모집단의 제곱근(예상되는 최대 권장 충돌 횟수)과 동일한 비컨 길이를 구합니다.

   $$\sqrt{(\text{Population})} = \text{Population} * 2^{-(\text{beacon length})}$$

   이 방정식으로 생성된 출력을 더 짧은 비컨 길이로 반내림하는 것이 좋습니다. 예를 들어 방정식이 15.6의 비컨 길이를 생성하는 경우 해당 값을 16비트로 반올림하는 대신 15비트로 반내림하는 것이 좋습니다.

4. 비컨 길이 선택

   이 방정식은 해당 분야에 권장되는 비컨 길이 범위만 식별합니다. 가능하면 데이터 세트의 보안을 유지하기 위해 더 짧은 비컨 길이를 사용하는 것이 좋습니다. 그러나 실제로 사용하는 비컨 길이는

위협 모델에 따라 결정됩니다. 해당 분야에 가장 적합한 비컨 길이를 결정하기 위해 위협 모델을 검토할 때 성능 요구 사항을 고려하세요.

더 짧은 비컨 길이를 사용하면 쿼리 성능이 저하되고, 더 긴 비컨 길이를 사용하면 보안이 저하됩니다. 일반적으로 데이터 세트가 고르지 않게 분포되어 있거나 상관된 필드에서 별도의 비컨을 구성하는 경우 더 짧은 비컨 길이를 사용하여 데이터 세트 분포에 대해 공개되는 정보의 양을 최소화해야 합니다.

위협 모델을 검토하고 필드 분포에 대해 밝혀진 구별 정보가 전체 보안에 위협이 되지 않는다고 판단하는 경우 계산한 권장 범위보다 긴 비컨 길이를 사용하도록 선택할 수 있습니다. 예를 들어, 필드에 대한 권장 비컨 길이 범위를 9~16비트로 계산한 경우 성능 손실을 방지하기 위해 24비트의 비컨 길이를 사용하도록 선택할 수 있습니다.

비컨 길이를 신중하게 선택하세요. 비컨으로 새 레코드를 작성한 후에는 비컨 길이를 업데이트할 수 없습니다.

예제

`unit` 필드를 암호화 작업의 `ENCRYPT_AND_SIGN`로 표시한 데이터베이스를 고려하세요. `unit` 필드에 대한 표준 비컨을 구성하려면 `unit` 필드에 대한 예상 오탐 수와 비컨 길이를 결정해야 합니다.

1. 모집단 추정

   위협 모델과 현재 데이터베이스 솔루션을 검토한 결과 `unit` 필드는 결국 100,000개의 고유 값을 갖게 될 것으로 예상됩니다.

   이는 모집단 = 100,000을 의미합니다.

2. 예상 충돌 횟수에 대한 권장 범위 계산.

   이 예제에서 예상되는 충돌 횟수는 2~316 사이여야 합니다.

   ```
   2 ≤ number of collisions < √(Population)
   ```

   a.
   ```
   2 ≤ number of collisions < √(100,000)
   ```

   b.
   ```
   2 ≤ number of collisions < 316
   ```

3. 비컨 길이에 대한 권장 범위를 계산합니다.

이 예제에서 비컨 길이는 9~16비트 사이여야 합니다.

```
number of collisions = Population * 2^-(beacon length)
```

a. 예상되는 충돌 횟수가 2단계에서 식별된 최소 횟수와 동일한 비컨 길이를 계산합니다.

```
2 = 100,000 * 2^-(beacon length)
```

비컨 길이 = 15.6 또는 15비트

b. 예상되는 충돌 횟수가 2단계에서 식별된 최대 횟수와 동일한 비컨 길이를 계산합니다.

```
316 = 100,000 * 2^-(beacon length)
```

비컨 길이 = 8.3 또는 8비트

4. 보안 및 성능 요구 사항에 적합한 비컨 길이를 결정합니다.

15개 미만의 비트마다 성능 비용과 보안이 두 배로 늘어납니다.

- 16비트
  - 평균적으로 각 고유 값은 1.5개의 다른 단위에 매핑됩니다.
  - 보안: 잘린 HMAC 태그가 동일한 두 레코드는 동일한 일반 텍스트 값을 가질 가능성이 66%입니다.
  - 성능: 쿼리는 실제로 요청한 레코드 10개마다 레코드 15개를 검색합니다.
- 14비트
  - 평균적으로 각 고유 값은 6.1개의 다른 단위에 매핑됩니다.
  - 보안: 잘린 HMAC 태그가 동일한 두 레코드는 동일한 일반 텍스트 값을 가질 가능성이 33%입니다.
  - 성능: 쿼리는 실제로 요청한 레코드 30개마다 레코드 10개를 검색합니다.

비컨 이름 선택

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

모든 비컨은 고유한 비컨 이름으로 식별됩니다. 비컨이 구성되면 암호화된 필드를 쿼리할 때 사용하는 이름이 비컨 이름이 됩니다. 비컨 이름은 암호화된 필드 또는 가상 필드와 같은 이름일 수 있지만 암호화되지 않은 필드와 같은 이름일 수는 없습니다. 서로 다른 두 비컨은 동일한 비컨 이름을 가질 수 없습니다.

비컨의 이름을 지정하고 구성하는 방법을 보여주는 예제는 비컨 구성을 참조하세요.

표준 비컨 이름 지정

표준 비컨의 이름을 지정할 때는 가능하면 비컨 이름을 비컨 소스로 확인하는 것이 좋습니다. 즉, 표준 비컨을 구성하는 데 사용되는 암호화된 필드 또는 가상 필드의 이름과 비컨 이름은 동일합니다. 예를 들어 `LastName`라는 이름의 암호화된 필드에 대한 표준 비컨을 만드는 경우 비컨 이름도 `LastName`이 되어야 합니다.

비컨 이름이 비컨 소스와 동일한 경우 구성에서 비컨 소스를 생략할 수 있으며 AWS Database Encryption SDK는 비컨 이름을 비컨 소스로 자동으로 사용합니다.

비컨 구성

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

검색 가능한 암호화를 지원하는 두 가지 유형의 비컨이 있습니다. 표준 비컨은 동등 검색을 수행합니다. 데이터베이스에서 검색 가능한 암호화를 구현하는 가장 간단한 방법입니다. 복합 비컨은 기본적 일반 텍스트 문자열과 표준 비컨을 결합하여 보다 복잡한 쿼리를 수행합니다.

비컨은 채워지지 않은 새 데이터베이스에 구현되도록 설계되었습니다. 기존 데이터베이스에 구성된 모든 비컨은 데이터베이스에 기록된 새 레코드만 매핑합니다. 비컨은 필드의 일반 텍스트 값에서 계산됩니다. 필드가 암호화되면 비컨이 기존 데이터를 매핑할 방법이 없습니다. 비컨으로 새 레코드를 작성한 후에는 비컨의 구성을 업데이트할 수 없습니다. 하지만 레코드에 추가하는 새 필드에 대해 새 비컨을 추가할 수 있습니다.

액세스 패턴을 파악한 후에는 데이터베이스 구현의 두 번째 단계로 비컨을 구성해야 합니다. 그런 다음 모든 비컨을 구성한 후 AWS KMS 계층적 키링을 생성하고, 비컨 버전을 정의하고, 각 비컨에 대한 보조 인덱스를 구성하고, 암호화 작업을 정의하고, 데이터베이스 및 AWS Database Encryption SDK 클라이언트를 구성해야 합니다. 자세한 내용은 비컨 사용을 참조하세요.

비컨 버전을 더 쉽게 정의하려면 표준 및 복합 비컨에 대한 목록을 만드는 것이 좋습니다. 생성한 각 비컨을 구성할 때 해당 표준 또는 복합 비컨 목록에 추가합니다.

주제

표준 비컨 구성

표준 비컨은 데이터베이스에서 검색 가능한 암호화를 구현하는 가장 간단한 방법입니다. 암호화된 필드 또는 가상 필드 하나에 대해서만 동등 검색을 수행할 수 있습니다.

구성 구문의 예제

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>();
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "beaconName",
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let standard_beacon_list = vec![
```

```
StandardBeacon::builder().name("beacon_name").length(beacon_length_in_bits).build()?,
```

표준 비컨을 구성하기 위해서는 다음 값을 제공해야 합니다.

비컨 이름

암호화된 필드를 쿼리할 때 사용하는 이름.

비컨 이름은 암호화된 필드 또는 가상 필드와 같은 이름일 수 있지만 암호화되지 않은 필드와 같은 이름일 수는 없습니다. 가능하면 표준 비컨을 구성하는 데 사용할 암호화된 필드 또는 가상 필드의 이름을 사용하는 것이 좋습니다. 서로 다른 두 비컨은 동일한 비컨 이름을 가질 수 없습니다. 구현에 가장 적합한 비컨 이름을 결정하는 데 도움이 필요하면 비컨 이름 선택을 참조하세요.

비컨 길이

잘라낸 후에도 유지되는 비컨 해시 값의 비트 수입니다.

비컨 길이에 따라 해당 비컨에서 생성되는 평균 오탐 수가 결정됩니다. 구현에 적합한 비컨 길이를 결정하는 데 도움이 되는 자세한 내용 및 도움말은 비컨 길이 결정을 참조하세요.

비컨 소스(선택 사항)

표준 비컨을 구성하는 데 사용되는 필드입니다.

비컨 소스는 필드 이름이거나 중첩된 필드의 값을 참조하는 인덱스이어야 합니다. 비컨 이름이 비컨 소스와 동일한 경우 구성에서 비컨 소스를 생략할 수 있으며 AWS Database Encryption SDK는 비컨 이름을 비컨 소스로 자동으로 사용합니다.

가상 필드 생성

가상 필드를 만들려면 가상 필드의 이름과 원본 필드 목록을 제공해야 합니다. 가상 부분 목록에 원본 필드를 추가하는 순서에 따라 가상 필드를 작성할 때 원본 필드를 연결하는 순서가 결정됩니다. 다음 예제에서는 원본 필드 두 개를 완전히 연결하여 가상 필드를 만듭니다.

> **Note**
>
> 데이터베이스를 채우기 전에 가상 필드가 예상 결과를 생성하는지 확인하는 것이 좋습니다. 자세한 내용은 비컨 출력 테스트를 참조하세요.

Java

전체 코드 예제 참조: VirtualBeaconSearchableEncryptionExample.java

```
List<VirtualPart> virtualPartList = new ArrayList<>();
    virtualPartList.add(sourceField1);
    virtualPartList.add(sourceField2);

VirtualField virtualFieldName = VirtualField.builder()
    .name("virtualFieldName")
    .parts(virtualPartList)
    .build();

List<VirtualField> virtualFieldList = new ArrayList<>();
    virtualFieldList.add(virtualFieldName);
```

C# / .NET

전체 코드 예제 참조: VirtualBeaconSearchableEncryptionExample.cs

```
var virtualPartList = new List<VirtualPart> { sourceField1, sourceField2 };

var virtualFieldName = new VirtualField
{
    Name = "virtualFieldName",
    Parts = virtualPartList
};

var virtualFieldList = new List<VirtualField> { virtualFieldName };
```

Rust

전체 코드 예제: virtual_beacon_searchable_encryption.rs를 참조하세요. https://github.com/aws/aws-database-encryption-sdk-dynamodb/blob/main/releases/rust/db_esdk/examples/searchableencryption/virtual_beacon_searchable_encryption.rs

```
let virtual_part_list = vec![source_field_one, source_field_two];

let state_and_has_test_result_field = VirtualField::builder()
    .name("virtual_field_name")
    .parts(virtual_part_list)
    .build()?;
```

```
let virtual_field_list = vec![virtual_field_name];
```

원본 필드의 특정 세그먼트를 사용하여 가상 필드를 만들려면 원본 필드를 가상 부분 목록에 추가하기 전에 해당 변환을 정의해야 합니다.

가상 필드에 대한 보안 고려 사항

비컨은 필드의 암호화된 상태를 변경하지 않습니다. 그러나 비컨을 사용할 때 쿼리의 효율성과 데이터 분포에 대해 공개되는 정보의 양 사이에는 본질적인 균형이 있습니다. 비컨을 구성하는 방식에 따라 해당 비컨이 보존하는 보안 수준이 결정됩니다.

소스 필드가 기존 표준 비컨과 겹치는 가상 필드를 만들지 마세요. 표준 비컨을 만드는 데 이미 사용된 소스 필드를 포함하는 가상 필드를 만들면 두 비컨의 보안 수준이 낮아질 수 있습니다. 보안이 약화되는 정도는 추가 소스 필드에서 추가한 엔트로피 수준에 따라 달라집니다. 엔트로피 수준은 추가 소스 필드의 고유 값 분포와 추가 소스 필드가 가상 필드의 전체 크기에 기여하는 비트 수에 따라 결정됩니다.

모집단 및 비컨 길이를 사용하여 가상 필드의 원본 필드가 데이터 세트의 보안을 유지하는지 확인할 수 있습니다. 모집단은 필드의 예상 고유 값 수입니다. 모집단은 정확할 필요는 없습니다. 필드의 모집단을 추정하는 데 도움이 필요하면 모집단 추정을 참조하세요.

가상 필드의 보안을 검토할 때 다음 예제를 고려하세요.

- Beacon1은 `FieldA`로 구성됩니다. `FieldA`의 모집단은 $2^{(\text{Beacon1 길이})}$보다 큽니다.
- Beacon2는 `VirtualField`으로 구성되어 있는데 이는 `FieldA`, `FieldB`, `FieldC`, 및 `FieldD`로 구성되어 있습니다. `FieldB`, `FieldC`, 및 `FieldD`을 합친 모집단은 2^N 이상입니다.

다음 설명이 참인 경우 Beacon2는 Beacon1과 Beacon2의 보안을 모두 유지합니다.

```
N ≥ (Beacon1 length)/2
```

및

```
N ≥ (Beacon2 length)/2
```

비컨 스타일 정의

표준 비컨을 사용하여 암호화된 필드 또는 가상 필드를 동등하게 검색할 수 있습니다. 또는 복합 비컨을 구성하여 더 복잡한 데이터베이스 작업을 수행하는 데 사용할 수 있습니다. 표준 비컨을 구성하고

관리하는 데 도움이 되도록 AWS Database Encryption SDK는 표준 비컨의 용도를 정의하는 다음과 같은 선택적 비컨 스타일을 제공합니다.

> **Note**
>
> 비컨 스타일을 정의하려면 AWS Database Encryption SDK 버전 3.2 이상을 사용해야 합니다. 비컨 구성에 비컨 스타일을 추가하기 전에 모든 리더에 새 버전을 배포합니다.

PartOnly

로 정의된 표준 비컨은 복합 비컨의 암호화된 부분을 정의하는 데만 사용할 `PartOnly` 수 있습니다. `PartOnly` 표준 비컨은 직접 쿼리할 수 없습니다.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .partOnly(PartOnly.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C#/.NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        PartOnly = new PartOnly()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::PartOnly(PartOnly::builder().build()?))
    .build()?
```

Shared

기본적으로 모든 표준 비컨은 비컨 계산을 위해 고유한 HMAC 키를 생성합니다. 따라서 두 개의 개별 표준 비컨에서 암호화된 필드에 대해 동등 검색을 수행할 수 없습니다. 로 정의된 표준 비컨은 계산에 다른 표준 비컨의 HMAC 키를 Shared 사용합니다.

예를 들어 beacon1 필드를 beacon2 필드와 비교해야 하는 경우 계산에의 HMAC 키를 사용하는 Shared 비컨beacon2으로 beacon1를 정의합니다.

> **Note**
>
> 비Shared컨을 구성하기 전에 보안 및 성능 요구 사항을 고려하세요. Shared 비컨은 데이터 세트 배포에 대해 식별할 수 있는 통계 정보의 양을 늘릴 수 있습니다. 예를 들어 동일한 일반 텍스트 값을 포함하는 공유 필드를 공개할 수 있습니다.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .shared(Shared.builder().other("beacon1").build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C#/.NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        Shared = new Shared { Other = "beacon1" }
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::Shared(
       Shared::builder().other("beacon1").build()?,
    ))
    .build()?
```

AsSet

기본적으로 필드 값이 집합인 경우 AWS Database Encryption SDK는 집합에 대한 단일 표준 비컨을 계산합니다. 따라서가 암호화된 필드CONTAINS(*a*, :*value*)인 쿼리*a*는 수행할 수 없습니다. 로 정의된 표준 비컨은 집합의 각 개별 요소에 대한 개별 표준 비컨 값을 AsSet 계산하고 비컨 값을 항목에 집합으로 저장합니다. 이렇게 하면 AWS Database Encryption SDK가 쿼리를 수행할 수 있습니다CONTAINS(*a*, :*value*).

AsSet 표준 비컨을 정의하려면 집합의 요소가 모두 동일한 비컨 길이를 사용할 수 있도록 동일한 집단에 속해야 합니다. 비컨 값을 계산할 때 충돌이 있는 경우 비컨 세트의 요소가 일반 텍스트 세트보다 적을 수 있습니다.

> **Note**
>
> 비AsSet컨을 구성하기 전에 보안 및 성능 요구 사항을 고려하세요. AsSet 비컨은 데이터 세트 배포에 대해 식별할 수 있는 통계 정보의 양을 늘릴 수 있습니다. 예를 들어 일반 텍스트 세트의 크기를 나타낼 수 있습니다.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .asSet(AsSet.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C#/.NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        AsSet = new AsSet()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::AsSet(AsSet::builder().build()?))
    .build()?
```

SharedSet

로 정의된 표준 비컨은 Shared 및 AsSet 함수를 SharedSet 결합하여 집합 및 필드의 암호화된 값에 대해 동등 검색을 수행할 수 있습니다. 이렇게 하면 AWS Database Encryption SDK가 CONTAINS(*a*, *b*)*a*가 암호화된 세트이고 *b*가 암호화된 필드인 쿼리를 수행할 수 있습니다.

> **Note**
>
> 비Shared컨을 구성하기 전에 보안 및 성능 요구 사항을 고려하세요. SharedSet 비컨은 데이터 세트 배포에 대해 식별할 수 있는 통계 정보의 양을 늘릴 수 있습니다. 예를 들어 일반 텍스트 세트의 크기 또는 동일한 일반 텍스트 값을 포함하는 공유 필드를 공개할 수 있습니다.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .sharedSet(SharedSet.builder().other("beacon1").build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C#/.NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        SharedSet = new SharedSet { Other = "beacon1" }
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::SharedSet(
        SharedSet::builder().other("beacon1").build()?,
```

```
    ))
    .build()?
```

복합 비컨 설정

복합 비컨은 기본 일반 텍스트 문자열과 표준 비컨을 결합하여 단일 인덱스에서 서로 다른 두 가지 레코드 유형을 쿼리하거나 정렬 키로 필드 조합을 쿼리하는 등 복잡한 데이터베이스 작업을 수행합니다. 복합 비컨은 `ENCRYPT_AND_SIGN`, `SIGN_ONLY`및 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 필드로 구성할 수 있습니다. 복합 비컨에 포함된 모든 암호화된 필드에 대해 표준 비컨을 만들어야 합니다.

> **Note**
>
> 데이터베이스를 채우기 전에 복합 비컨이 예상 결과를 생성하는지 확인하는 것이 좋습니다. 자세한 내용은 비컨 출력 테스트를 참조하세요.

구성 구문의 예제

Java

복합 비컨 구성

다음 예제에서는 복합 비컨 구성 내에서 로컬로 암호화된 부분 목록과 서명된 부분 목록을 정의합니다.

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
    .name("compoundBeaconName")
    .split(".")
    .encrypted(encryptedPartList)
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

비컨 버전 정의

다음 예제에서는 비컨 버전에서 암호화되고 서명된 부분 목록을 전역적으로 정의합니다. 비컨 버전 정의에 대한 자세한 내용은 비컨 사용을 참조하세요.

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

전체 코드 샘플 보기: BeaconConfig.cs

복합 비컨 구성

다음 예제에서는 복합 비컨 구성 내에서 로컬로 암호화된 부분 목록과 서명된 부분 목록을 정의합니다.

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Encrypted = encryptedPartList,
    Signed = signedPartList,
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

비컨 버전 정의

다음 예제에서는 비컨 버전에서 암호화되고 서명된 부분 목록을 전역적으로 정의합니다. 비컨 버전 정의에 대한 자세한 내용은 비컨 사용을 참조하세요.

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

전체 코드 샘플 참조: beacon_config.rs

복합 비컨 구성

다음 예제에서는 복합 비컨 구성 내에서 로컬로 암호화된 부분 목록과 서명된 부분 목록을 정의합니다.

```
let compound_beacon_list = vec![
    CompoundBeacon::builder()
        .name("compound_beacon_name")
        .split(".")
        .encrypted(encrypted_parts_list)
        .signed(signed_parts_list)
        .constructors(constructor_list)
        .build()?
```

비컨 버전 정의

다음 예제에서는 비컨 버전에서 암호화되고 서명된 부분 목록을 전역적으로 정의합니다. 비컨 버전 정의에 대한 자세한 내용은 비컨 사용을 참조하세요.

```
let beacon_versions = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .encrypted_parts(encrypted_parts_list)
    .signed_parts(signed_parts_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
        SingleKeyStore::builder()
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_versions];
```

로컬 또는 전역적으로 정의된 목록에서 암호화된 부분과 서명된 부분을 정의할 수 있습니다. 가능하면 비컨 버전의 글로벌 목록에 암호화되고 서명된 부분을 정의하는 것이 좋습니다. 암호화되고 서명된 부분을 전역적으로 정의하면 각 부분을 한 번 정의한 다음 여러 복합 비컨 구성에서 해당 부분을 재사용할 수 있습니다. 암호화되거나 서명된 부분을 한 번만 사용하려는 경우 복합 비컨 구성의 로컬 목록에서 정의할 수 있습니다. 생성자 목록에서 로컬 부분과 글로벌 부분을 모두 참조할 수 있습니다.

암호화 및 서명된 부분 목록을 전역적으로 정의하는 경우 복합 비컨이 복합 비컨 구성의 필드를 조합할 수 있는 가능한 모든 방법을 식별하는 생성자 부분 목록을 제공해야 합니다.

> **Note**
>
> 암호화 및 서명된 부분 목록을 전역적으로 정의하려면 AWS Database Encryption SDK 버전 3.2 이상을 사용해야 합니다. 새 부분을 전역적으로 정의하기 전에 모든 리더에 새 버전을 배포합니다.
>
> 암호화되고 서명된 부분 목록을 전역적으로 정의하도록 기존 비컨 구성을 업데이트할 수 없습니다.

복합 비컨을 구성하기 위해서는 다음 값을 제공해야 합니다.

비컨 이름

암호화된 필드를 쿼리할 때 사용하는 이름.

비컨 이름은 암호화된 필드 또는 가상 필드와 같은 이름일 수 있지만 암호화되지 않은 필드와 같은 이름일 수는 없습니다. 두 비컨이 동일한 비컨 이름을 가질 수는 없습니다. 구현에 가장 적합한 비컨 이름을 결정하는 데 도움이 필요하면 비컨 이름 선택을 참조하세요.

분할 캐릭터

복합 비컨을 구성하는 부분을 구분하는 데 사용되는 문자입니다.

복합 비컨을 구성하는 모든 필드의 일반 텍스트 값에는 분할 문자가 나타날 수 없습니다.

암호화된 부분 목록

복합 비컨에 포함된 ENCRYPT_AND_SIGN 필드를 식별합니다.

각 부분에는 이름과 접두사가 포함되어야 합니다. 부분 이름은 암호화된 필드로 구성된 표준 비컨의 이름이어야 합니다. 접두사는 임의의 문자열일 수 있지만 고유해야 합니다. 암호화된 부분은 서명된 부분과 동일한 접두사를 가질 수 없습니다. 부분을 복합 비컨이 제공하는 다른 부분과 구분하는 짧은 값을 사용하는 것이 좋습니다.

가능하면 암호화된 부분을 전역적으로 정의하는 것이 좋습니다. 하나의 복합 비컨에서만 사용하려는 경우 암호화된 부분을 로컬에서 정의하는 것이 좋습니다. 로컬에서 정의된 암호화된 부분은 전역적으로 정의된 암호화된 부분과 동일한 접두사 또는 이름을 가질 수 없습니다.

Java

```
List<EncryptedPart> encryptedPartList = new ArrayList<>);
EncryptedPart encryptedPartExample = EncryptedPart.builder()
    .name("standardBeaconName")
    .prefix("E-")
    .build();
encryptedPartList.add(encryptedPartExample);
```

C# / .NET

```
var encryptedPartList = new List<EncryptedPart>();
var encryptedPartExample = new EncryptedPart
 {
    Name = "compoundBeaconName",
    Prefix = "E-"
 };
encryptedPartList.Add(encryptedPartExample);
```

Rust

```
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("standard_beacon_name")
        .prefix("E-")
        .build()?
];
```

서명된 부분 목록

복합 비컨에 포함된 서명된 필드를 식별합니다.

> **Note**
>
> 서명된 부분은 선택 사항입니다. 서명된 부분을 참조하지 않는 복합 비컨을 구성할 수 있습니다.

각 부분에는 이름, 출처 및 접두사가 포함되어야 합니다. 소스는 파트가 식별하는 `SIGN_ONLY` 또는 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 필드입니다. 소스는 필드 이름이거나 중첩된 필드의 값을 참조하는 인덱스이어야 합니다. 파트 이름이 소스를 식별하는 경우 소스를 생략하면 AWS Database Encryption SDK가 자동으로 이름을 소스로 사용합니다. 가능하면 소스를 부분 이름으로 지정하는 것이 좋습니다. 접두사는 임의의 문자열일 수 있지만 고유해야 합니다. 서명된 부분은 암호화된 부분과 동일한 접두사를 가질 수 없습니다. 부분을 복합 비컨이 제공하는 다른 부분과 구분하는 짧은 값을 사용하는 것이 좋습니다.

가능하면 서명 부분을 전역적으로 정의하는 것이 좋습니다. 하나의 복합 비컨에서만 사용하려는 경우 서명된 부분을 로컬에서 정의하는 것이 좋습니다. 로컬에서 정의된 서명된 부분은 전역적으로 정의된 서명된 부분과 동일한 접두사 또는 이름을 가질 수 없습니다.

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
SignedPart signedPartExample = SignedPart.builder()
    .name("signedFieldName")
    .prefix("S-")
    .build();
signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

Rust

```
let signed_parts_list = vec![
    SignedPart::builder()
        .name("signed_field_name_1")
        .prefix("S-")
        .build()?,
   SignedPart::builder()
        .name("signed_field_name_2")
        .prefix("SF-")
        .build()?,
];
```

생성자 목록

암호화되고 서명된 부분을 복합 비컨으로 조합할 수 있는 다양한 방법을 정의하는 생성자를 식별합니다. 생성자 목록에서 로컬 부분과 글로벌 부분을 모두 참조할 수 있습니다.

전역적으로 정의된 암호화 및 서명된 부분으로 복합 비컨을 구성하는 경우 생성자 목록을 제공해야 합니다.

전역적으로 정의된 암호화되거나 서명된 부분을 사용하여 복합 비컨을 구성하지 않는 경우 생성자 목록은 선택 사항입니다. 생성자 목록을 지정하지 않으면 AWS Database Encryption SDK는 다음 기본 생성자를 사용하여 복합 비컨을 어셈블합니다.

- 서명된 모든 부분은 서명된 부분 목록에 추가된 순서대로
- 암호화된 모든 부분(암호화된 부분 목록에 추가된 순서대로)
- 모든 부분이 필요합니다.

Constructors

각 생성자는 복합 비컨을 조합할 수 있는 한 가지 방법을 정의하는 생성자 부분의 정렬된 목록입니다. 생성자 부분은 목록에 추가된 순서대로 함께 결합되며 각 부분은 지정된 분할 문자로 구분됩니다.

각 생성자 부분은 암호화된 부분이나 서명된 부분의 이름을 지정하고 생성자 내에서 해당 부분이 필수인지 아니면 선택적인지 정의합니다. 예를 들어 `Field1`, `Field1.Field2`, 및 `Field1.Field2.Field3`에 대한 복합 비컨을 조회하고자 한다면, `Field2` 및 `Field3`을 선택 사항으로 표시하고 생성자를 하나 생성합니다.

생성자마다 필수 부분이 하나 이상 있어야 합니다. 쿼리에 `BEGINS_WITH` 연산자를 사용할 수 있도록 각 생성자의 첫 번째 부분을 필수로 설정하는 것이 좋습니다.

생성자의 필수 부분이 모두 레코드에 있으면 생성자는 성공합니다. 새 레코드를 작성할 때 복합 비컨은 생성자 목록을 사용하여 제공된 값에서 비컨을 조합할 수 있는지 여부를 결정합니다. 생성자 목록에 생성자가 추가된 순서대로 비컨을 조합하려고 시도하고 성공한 첫 번째 생성자를 사용합니다. 생성자가 성공하지 못하면 비컨이 레코드에 기록되지 않습니다.

쿼리 결과가 정확한지 확인하려면 모든 리더와 작성자가 동일한 순서의 생성자를 지정해야 합니다.

다음 절차에 따라 생성자 목록을 지정하세요.

1. 암호화된 각 부분과 서명된 부분에 대한 생성자 부분을 만들어 해당 부분이 필요한지 여부를 정의합니다.

   생성자 부분 이름은 생성자가 나타내는 표준 비컨 또는 서명된 필드의 이름이어야 합니다.

   Java

   ```
   ConstructorPart field1ConstructorPart = ConstructorPart.builder()
           .name("Field1")
           .required(true)
           .build();
   ```

   C# / .NET

   ```
   var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
    = true };
   ```

   Rust

   ```
   let field_1_constructor_part = ConstructorPart::builder()
       .name("field_1")
       .required(true)
       .build()?;
   ```

2. 1단계에서 만든 생성자 부분을 사용하여 복합 비컨을 조합할 수 있도록 가능한 모든 방법에 맞는 생성자를 만듭니다.

   예를 들어 `Field1.Field2.Field3` 및 `Field4.Field2.Field3`에 대해 쿼리하려면 두 개의 생성자를 만들어야 합니다. `Field1` 및 `Field4`은 두 개의 별도 생성자에 정의되어 있으므로 둘 다 필요할 수 있습니다.

   Java

   ```
   // Create a list for Field1.Field2.Field3 queries
   List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
   field123ConstructorPartList.add(field1ConstructorPart);
   field123ConstructorPartList.add(field2ConstructorPart);
   field123ConstructorPartList.add(field3ConstructorPart);
   Constructor field123Constructor = Constructor.builder()
           .parts(field123ConstructorPartList)
           .build();
   // Create a list for Field4.Field2.Field1 queries
   List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
   field421ConstructorPartList.add(field4ConstructorPart);
   field421ConstructorPartList.add(field2ConstructorPart);
   field421ConstructorPartList.add(field1ConstructorPart);
   Constructor field421Constructor = Constructor.builder()
           .parts(field421ConstructorPartList)
           .build();
   ```

   C# / .NET

   ```
   // Create a list for Field1.Field2.Field3 queries
    var field123ConstructorPartList = new Constructor
   {
       Parts = new List<ConstructorPart> { field1ConstructorPart,
    field2ConstructorPart, field3ConstructorPart }
   };
   // Create a list for Field4.Field2.Field1 queries
   var field421ConstructorPartList = new Constructor
   {
       Parts = new List<ConstructorPart> { field4ConstructorPart,
    field2ConstructorPart, field1ConstructorPart }
   };
   ```

Rust

```
// Create a list for field1.field2.field3 queries
let field1_field2_field3_constructor = Constructor::builder()
    .parts(vec![
        field1_constructor_part,
        field2_constroctor_part.clone(),
        field3_constructor_part,
    ])
    .build()?;

// Create a list for field4.field2.field1 queries
let field4_field2_field1_constructor = Constructor::builder()
    .parts(vec![
        field4_constructor_part,
        field2_constroctor_part.clone(),
        field1_constructor_part,
    ])
    .build()?;
```

3. 2단계에서 만든 모든 생성자를 포함하는 생성자 목록을 만듭니다.

Java

```
List<Constructor> constructorList = new ArrayList<>();
constructorList.add(field123Constructor)
constructorList.add(field421Constructor)
```

C# / .NET

```
var constructorList = new List<Constructor>
{
    field123Constructor,
    field421Constructor
};
```

Rust

```
let constructor_list = vec![
    field1_field2_field3_constructor,
    field4_field2_field1_constructor,
```

```
];
```

4. 복합 비컨을 생성할 `constructorList` 때를 지정합니다.

구성의 예

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

다음 예제는 표준 및 복합 비컨을 구성하는 방법을 보여 줍니다. 다음 구성은 비컨 길이를 제공하지 않습니다. 구성에 적합한 비컨 길이를 결정하는 데 도움이 필요하면 비컨 길이 선택을 참조하세요.

비컨을 구성하고 사용하는 방법을 보여주는 전체 코드 예제를 보려면 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에서 Java, .NET 및 Rust 검색 가능한 암호화 예제를 참조하세요.

주제

표준 비컨

정확히 일치하는지 `inspector_id_last4` 필드를 쿼리하려면 다음 구성을 사용하여 표준 비컨을 만듭니다.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>>);
```

```
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "inspector_id_last4",
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;

let standard_beacon_list = vec![last4_beacon, unit_beacon];
```

복합 비컨

`inspector_id_last4` 및`inspector_id_last4.unit` 에서 `UnitInspection` 데이터베이스를 쿼리하려면 다음 구성으로 복합 비컨을 만듭니다. 이 복합 비컨에는 암호화된 부분만 필요합니다.

Java

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
List<StandardBeacon> standardBeaconList = new ArrayList<>);
StandardBeacon inspectorBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(inspectorBeacon);

StandardBeacon unitBeacon = StandardBeacon.builder()
    .name("unit")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(unitBeacon);

// 2. Define the encrypted parts.
List<EncryptedPart> encryptedPartList = new ArrayList<>);
```

```
// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
EncryptedPart encryptedPartInspector = EncryptedPart.builder()
    .name("inspector_id_last4")
    .prefix("I-")
    .build();
encryptedPartList.add(encryptedPartInspector);

EncryptedPart encryptedPartUnit = EncryptedPart.builder()
    .name("unit")
    .prefix("U-")
    .build();
encryptedPartList.add(encryptedPartUnit);

// 3. Create the compound beacon.
// This compound beacon only requires a name, split character,
// and list of encrypted parts
CompoundBeacon inspectorUnitBeacon = CompoundBeacon.builder()
    .name("inspectorUnitBeacon")
    .split(".")
    .sensitive(encryptedPartList)
    .build();
```

C# / .NET

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
StandardBeacon inspectorBeacon = new StandardBeacon
 {
   Name = "inspector_id_last4",
   Length = 10
 };
standardBeaconList.Add(inspectorBeacon);
StandardBeacon unitBeacon = new StandardBeacon
 {
    Name = "unit",
    Length = 30
 };
standardBeaconList.Add(unitBeacon);
```

```
// 2. Define the encrypted parts.
var last4EncryptedPart = new EncryptedPart

// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
var last4EncryptedPart = new EncryptedPart
 {
   Name = "inspector_id_last4",
   Prefix = "I-"
 };
encryptedPartList.Add(last4EncryptedPart);

var unitEncryptedPart = new EncryptedPart
 {
   Name = "unit",
   Prefix = "U-"
 };
encryptedPartList.Add(unitEncryptedPart);

// 3. Create the compound beacon.
// This compound beacon only requires a name, split character,
// and list of encrypted parts
var compoundBeaconList = new List<CompoundBeacon>>);
var inspectorCompoundBeacon = new CompoundBeacon
  {
      Name = "inspector_id_last4",
      Split = ".",
      Encrypted = encryptedPartList
  };
compoundBeaconList.Add(inspectorCompoundBeacon);
```

Rust

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;
```

```
let standard_beacon_list = vec![last4_beacon, unit_beacon];

// 2. Define the encrypted parts.
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("inspector_id_last4")
        .prefix("I-")
        .build()?,
    EncryptedPart::builder().name("unit").prefix("U-").build()?,
];

// 3. Create the compound beacon
// This compound beacon only requires a name, split character,
// and list of encrypted parts
let compound_beacon_list = vec![CompoundBeacon::builder()
    .name("last4UnitCompound")
    .split(".")
    .encrypted(encrypted_parts_list)
    .build()?];
```

비컨 사용

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

비컨을 사용하면 쿼리 중인 전체 데이터베이스를 복호화하지 않고도 암호화된 레코드를 검색할 수 있습니다. 비컨은 채워지지 않은 새 데이터베이스에 구현되도록 설계되었습니다. 기존 데이터베이스에 구성된 모든 비컨은 데이터베이스에 기록된 새 레코드만 매핑합니다. 비컨은 필드의 일반 텍스트 값에서 계산됩니다. 필드가 암호화되면 비컨이 기존 데이터를 매핑할 방법이 없습니다. 비컨으로 새 레코드를 작성한 후에는 비컨의 구성을 업데이트할 수 없습니다. 하지만 레코드에 추가하는 새 필드에 새 비컨을 추가할 수 있습니다.

비컨을 구성한 후에는 데이터베이스를 채우고 비컨에 대한 쿼리를 수행하기 전에 다음 단계를 완료해야 합니다.

1. AWS KMS 계층적 키링 생성

   검색 가능한 암호화를 사용하려면 AWS KMS 계층적 키링을 사용하여 레코드 보호에 사용되는 데이터 키를 생성, 암호화 및 복호화해야 합니다.

   비컨을 구성한 후 계층적 키링 사전 요구 사항을 조합하고 계층적 키링을 생성합니다.

   계층적 키링이 필요한 이유에 대한 자세한 내용은 검색 가능한 암호화를 위한 계층적 키링 사용을 참조하세요.

2.

   비컨 버전 정의

   keyStore, keySource, 구성한 모든 표준 비컨 목록, 구성한 모든 복합 비컨 목록, 암호화된 부분 목록, 서명된 부분 목록 및 비컨 버전을 지정합니다. 비컨 버전에 대해 1을 지정해야 합니다. keySource 정의에 대한 지침은 비컨 키 소스 정의하기 섹션을 참조하세요.

   다음 Java 예제는 단일 테넌트 데이터베이스의 비컨 버전을 정의합니다. 멀티테넌트 데이터베이스의 비컨 버전을 정의하는 데 도움이 필요하면 멀티테넌트 데이터베이스의 검색 가능한 암호화를 참조하세요.

   Java

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartsList)
        .signedParts(signedPartsList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
```

```
);
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branch-key-id,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
        SingleKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
```

```
    .build()?;
let beacon_versions = vec![beacon_version];
```

3. 보조 인덱스 구성

   비컨을 구성한 후 암호화된 필드를 검색하려면 먼저 각 비컨을 반영하는 보조 인덱스를 구성해야 합니다. 자세한 내용은 비컨을 사용한 보조 인덱스 구성 단원을 참조하십시오.

4. 암호화 작업 정의

   표준 비컨을 구성하는 데 사용되는 모든 필드를 ENCRYPT_AND_SIGN으로 표시해야 합니다. 비컨을 구성하는 데 사용되는 다른 모든 필드는 SIGN_ONLY 또는 로 표시되어야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

5. AWS Database Encryption SDK 클라이언트 구성

   DynamoDB 테이블의 테이블 항목을 보호하는 AWS Database Encryption SDK 클라이언트를 구성하려면 DynamoDB용 Java 클라이언트 측 암호화 라이브러리를 참조하세요.

비컨 쿼리

구성하는 비컨의 유형에 따라 수행할 수 있는 쿼리 유형이 결정됩니다. 표준 비컨은 필터 표현식을 사용하여 동등 검색을 수행합니다. 복합 비컨은 리터럴 일반 텍스트 문자열과 표준 비컨을 결합하여 복잡한 쿼리를 수행합니다. 암호화된 데이터를 쿼리할 때는 비컨 이름을 검색합니다.

두 표준 비컨에 동일한 기본 일반 텍스트가 포함되어 있더라도 두 표준 비컨의 값을 비교할 수는 없습니다. 두 개의 표준 비컨은 동일한 일반 텍스트 값에 대해 서로 다른 두 개의 HMAC 태그를 생성합니다. 따라서 표준 비컨은 다음 쿼리를 수행할 수 없습니다.

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)
- CONTAINS(*beacon1*, *beacon2*)

복합 비컨은 다음 쿼리를 수행할 수 있습니다.

- BEGINS_WITH(*a*), 여기서 *a*는 조립된 복합 비컨이 시작하는 필드의 전체 값을 반영합니다. BEGINS_WITH 연산자를 사용하여 특정 하위 문자열로 시작하는 값을 식별할 수 없습니다. 하지만 BEGINS_WITH(*S_*)을 사용할 수 있으며, 여기서 *S_*는 조립된 복합 비컨이 시작하는 부분의 접두사를 반영합니다.

- CONTAINS(*a*), 여기서 *a*는 조립된 복합 비컨에 포함된 필드의 전체 값을 반영합니다. CONTAINS 연산자를 사용하여 세트 내의 특정 하위 문자열이나 값이 포함된 레코드를 식별할 수 없습니다.

  예를 들어 *a*이 세트의 값을 반영하는 쿼리 CONTAINS(*path*, "*a*"는 수행할 수 없습니다.
- 복합 비컨의 서명된 부분을 비교할 수 있습니다. 서명된 부분을 비교할 때 선택적으로 하나 이상의 서명된 부분에 암호화된 부분의 접두사를 추가할 수 있지만 암호화된 필드의 값을 쿼리에 포함할 수는 없습니다.

  예를 들어 서명된 부분을 비교하고 *signedField1* = *signedField2* 또는 *value* IN (*signedField1*, *signedField2*, ...)에 대해 쿼리할 수 있습니다.

  *signedField1.A_* = *signedField2.B_*에 대한 쿼리에 의해 서명된 부분과 암호화된 부분의 접두사를 비교할 수도 있습니다.
- *field* BETWEEN *a* AND *b*, 여기서 *a* 및 *b*는 서명된 부분입니다. 암호화된 부분의 접두사를 하나 이상의 서명된 부분에 선택적으로 추가할 수 있지만 암호화된 필드의 값을 쿼리에 포함할 수는 없습니다.

복합 비컨에 대한 쿼리에 포함시키는 각 부분의 접두사를 포함해야 합니다. 예를 들어, 두 개의 필드, encryptedField 및 signedField로부터 복합 비컨 compoundBeacon을 구성한 경우, 비컨을 쿼리할 때 이 두 부분에 대해 구성된 접두사를 포함해야 합니다.

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue
```

멀티테넌트 데이터베이스를 위한 검색 가능한 암호화

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

데이터베이스에서 검색 가능한 암호화를 구현하려면 AWS KMS 계층적 키링을 사용해야 합니다. AWS KMS 계층적 키링은 레코드를 보호하는 데 사용되는 데이터 키를 생성, 암호화 및 해독합니다. 또한 비컨을 생성하는 데 사용되는 비컨 키도 생성합니다. 멀티테넌트 데이터베이스에서 AWS KMS 계층적 키링을 사용하는 경우 각 테넌트에 대해 고유한 브랜치 키와 비컨 키가 있습니다. 멀티테넌트 데이터베이스에서 암호화된 데이터를 쿼리하려면 쿼리하는 비컨을 생성하는 데 사용된 비컨 키 자료를 식별해야 합니다. 자세한 내용은 the section called “검색 가능한 암호화를 위한 계층적 키링 사용” 단원을 참조하십시오.

멀티테넌트 데이터베이스의 비컨 버전을 정의할 때는 구성한 모든 표준 비컨 목록, 구성한 모든 복합 비컨 목록, 비컨 버전 및 keySource을 지정합니다. 비콘 키 소스를 MultiKeyStore로 정의하고 keyFieldName을 포함해야 하며, 로컬 비컨 키 캐시에 대한 캐시 수명과 로컬 비컨 키 캐시에 대한 최대 캐시 크기를 포함해야 합니다.

서명된 비컨을 구성한 경우 해당 비컨이compoundBeaconList 에 포함되어야 합니다. 서명된 비컨은 SIGN_ONLY 및 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT필드에 대해 복잡한 쿼리를 인덱싱하고 수행하는 복합 비컨의 한 유형입니다.

Java

```
List<BeaconVersion> beaconVersions = new ArrayList<>();
    beaconVersions.add(
        BeaconVersion.builder()
                .standardBeacons(standardBeaconList)
                .compoundBeacons(compoundBeaconList)
                .version(1) // MUST be 1
                .keyStore(branchKeyStoreName)
                .keySource(BeaconKeySource.builder()
                        .multi(MultiKeyStore.builder()
                                .keyFieldName(keyField)
                                .cacheTTL(6000)
                                .maxCacheSize(10)
                        .build())
                .build())
        .build()
    );
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
```

```
            Multi = new MultiKeyStore
            {
                KeyId = branch-key-id,
                CacheTTL = 6000,
                MaxCacheSize = 10
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Multi(
        MultiKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .max_cache_size(10)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_version];
```

keyFieldName

keyFieldName는 지정된 테넌트에 대한 비컨을 생성하는 데 사용된 비컨 키와 관련된 branch-key-id를 저장하는 필드의 이름을 정의합니다.

데이터베이스에 새 레코드를 쓰는 경우 해당 레코드에 대한 비컨을 생성하는 데 사용되는 비컨 키를 식별하는 branch-key-id가 이 필드에 저장됩니다.

기본적으로 keyField는 데이터베이스에 명시적으로 저장되지 않는 개념적 필드입니다. AWS Database Encryption SDK는 자료 설명의 암호화된 데이터 키branch-key-id에서를 식별하고 복

합 비컨 및 서명된 비컨에서 참조할 수 keyField 있도록 개념에 값을 저장합니다. 자료 설명은 서명된 것이므로 개념적 keyField은 서명된 부분으로 간주됩니다.

암호화 작업에 keyField를 SIGN_ONLY 또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 필드로 포함하여 데이터베이스에 필드를 명시적으로 저장할 수도 있습니다. 이렇게 하면 데이터베이스에 레코드를 쓸 때마다 keyFielddp 수동으로 branch-key-id를 포함시켜야 합니다.

멀티테넌트 데이터베이스의 비컨 쿼리

비컨을 쿼리하려면 비컨을 재계산하는 데 필요한 적절한 비컨 키 자료를 식별할 수 있도록 쿼리에 keyField을 포함해야 합니다. 레코드의 비컨을 생성하는 데 사용되는 비컨 키와 관련된 branch-key-id을 지정해야 합니다. 브랜치 키 ID 공급자의 테넌트 branch-key-id를 식별하는 친숙한 이름은 지정할 수 없습니다. 다음과 같은 방법으로 쿼리에 keyField을 포함시킬 수 있습니다.

복합 비컨

레코드에 keyField을 명시적으로 저장하든 저장하지 않든, 복합 비컨에 서명된 부분으로keyField을 직접 포함시킬 수 있습니다. keyField 서명된 부분이 필요합니다.

예를 들어, 필드 2개와 encryptedField 및 signedField를 사용하여 복합 비컨 compoundBeacon을 생성하려면 keyField를 서명된 부분으로 포함해야 합니다. 이렇게 하면 compoundBeacon에서 다음 쿼리를 수행할 수 있습니다.

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue.K_branch-key-id
```

서명된 비컨

AWS Database Encryption SDK는 표준 및 복합 비컨을 사용하여 검색 가능한 암호화 솔루션을 제공합니다. 이러한 비컨에는 하나 이상의 암호화된 필드가 포함되어야 합니다. 그러나 AWS Database Encryption SDK는 일반 텍스트 SIGN_ONLY 및 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 필드에서 완전히 구성할 수 있는 서명된 비컨도 지원합니다.

서명된 비컨은 단일 부분으로 구성할 수 있습니다. keyField를 레코드에 명시적으로 저장하든 저장하지 않든, keyField에서 서명된 비컨을 생성하고 이를 사용하여 keyField 서명된 비컨에 대한 쿼리를 다른 비컨 중 하나에 대한 쿼리와 결합하는 복합 쿼리를 만들 수 있습니다. 예를 들어 다음 쿼리를 수행할 수 있습니다.

```
keyField = K_branch-key-id AND compoundBeacon =
 E_encryptedFieldValue.S_signedFieldValue
```

서명된 비컨을 구성하는 데 도움이 필요하면 서명된 비컨 만들기 섹션을 참조하세요

keyField에서 직접 쿼리하기

암호화 keyField 작업에서 keyField를 지정하고 레코드에 필드를 명시적으로 저장한 경우, 비컨의 쿼리와 비컨의 쿼리를 결합하는 복합 쿼리를 만들 수 있습니다. 표준 비컨을 쿼리하려는 경우 keyField에서 직접 쿼리하도록 선택할 수 있습니다. 예를 들어 다음 쿼리를 수행할 수 있습니다.

```
keyField = branch-key-id AND standardBeacon = S_standardBeaconValue
```

AWS DynamoDB용 Database Encryption SDK

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK for DynamoDB는 Amazon DynamoDB 설계에 클라이언트 측 암호화를 포함할 수 있는 소프트웨어 라이브러리입니다. AWS Database Encryption SDK for DynamoDB는 속성 수준 암호화를 제공하며 암호화할 항목과 데이터의 신뢰성을 보장하는 서명에 포함할 항목을 지정할 수 있습니다. 전송 중 및 유휴 상태의 중요 데이터를 암호화하면 일반 텍스트 데이터를 AWS을 포함한 제3자가 사용할 수 없게 하는 데 도움이 됩니다.

> **Note**
>
> AWS Database Encryption SDK는 PartiQL을 지원하지 않습니다.

DynamoDB에서 테이블은 항목 모음입니다. 각 항목은 속성 모음입니다. 각 속성마다 이름과 값이 있습니다. AWS Database Encryption SDK for DynamoDB는 속성 값을 암호화합니다. 그런 다음 속성에 대한 서명을 계산합니다. 암호화할 속성 값과 암호화 작업의 서명에 포함할 속성 값을 지정합니다.

이 장의 주제에서는 암호화된 필드, 클라이언트 설치 및 구성에 대한 지침, 시작하는 데 도움이 되는 Java 예제를 포함하여 DynamoDB용 AWS Database Encryption SDK에 대한 개요를 제공합니다.

주제

- 클라이언트측 및 서버 측 암호화
- 암호화 및 서명되는 필드
- DynamoDB의 검색 가능한 암호화
- 데이터 모델 업데이트
- AWS Database Encryption SDK for DynamoDB 사용 가능한 프로그래밍 언어
- 레거시 DynamoDB Encryption Client

클라이언트측 및 서버 측 암호화

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK for DynamoDB는 데이터베이스로 전송하기 전에 테이블 데이터를 암호화하는 클라이언트 측 암호화를 지원합니다. 하지만 DynamoDB는 테이블을 디스크에 보관할 때 테이블을 사용자 모르게 암호화하고 사용자가 테이블에 액세스할 때 복호화하는 서버 측 저장 데이터 암호화 기능을 제공합니다.

선택하는 도구는 데이터의 민감도와 애플리케이션의 보안 요구 사항에 따라 달라집니다. DynamoDB용 AWS Database Encryption SDK와 저장 데이터 암호화를 모두 사용할 수 있습니다. 암호화 및 서명된 항목을 DynamoDB로 전송하는 경우 DynamoDB는 항목을 보호 중인 상태로 인식하지 못하며, 이진 속성 값을 가진 일반적인 테이블 항목만 탐지합니다.

서버 측 저장 데이터 암호화

DynamoDB는 테이블을 디스크에 보관할 때 DynamoDB에서 테이블을 사용자 모르게 암호화하고 사용자가 테이블 데이터에 액세스할 때 복호화하는 저장 중 암호화, 서버 측 암호화 기능을 지원합니다.

AWS SDK를 사용하여 DynamoDB와 상호 작용하는 경우 기본적으로 데이터는 HTTPS 연결을 통해 전송 중에 암호화되고 DynamoDB 엔드포인트에서 해독된 다음 DynamoDB에 저장되기 전에 다시 암호화됩니다.

- 암호화 기본 제공. DynamoDB는 모든 테이블을 작성할 때 투명하게 암호화하고 복호화합니다. 저장 데이터 암호화를 활성화하거나 비활성화할 수 없습니다.
- DynamoDB는 암호화 키를 만들고 관리합니다. 각 테이블의 고유 키는 AWS Key Management Service(AWS KMS)을 암호화되지 않은 상태로 남기지 않는 AWS KMS key로 보호됩니다. 기본적으로 DynamoDB는 DynamoDB 서비스 계정의 AWS 소유 키를 사용하지만, 계정에서 AWS 관리형 키 또는 고객 관리 키를 선택하여 일부 또는 모든 테이블을 보호할 수 있습니다.
- 디스크에 대한 모든 테이블 데이터는 암호화됩니다. 암호화된 테이블이 디스크에 저장될 때 는 프라이머리 키와 로컬 및 글로벌 보조 인덱스를 포함하여 모든 테이블 데이터를 암호화합니다. 테이블에 정렬 키가 있는 경우 범위 경계를 표시하는 정렬 키 중 일부가 테이블 메타데이터에 일반 텍스트 형태로 저장됩니다.
- 테이블과 관련된 객체도 암호화됩니다. 저장 데이터 암호화는 DynamoDB 스트림, 전역 테이블, 백업이 내구성 있는 미디어에 기록될 때마다 보호합니다.

- 항목에 액세스하면 해당 항목의 암호가 복호화됩니다. 테이블에 액세스할 때 DynamoDB는 대상 항목을 포함하는 테이블의 부분을 복호화하고 일반 텍스트 항목을 사용자에게 반환합니다.

AWS DynamoDB용 Database Encryption SDK

클라이언트측 암호화는 소스에서 DynamoDB의 스토리지에 이르기까지 전송 중 및 저장 중인 데이터에 대한 엔드투엔드 보호를 제공합니다. 일반 텍스트 데이터는를 포함한 제3자에게 절대 노출되지 않습니다 AWS. 새 DynamoDB 테이블과 함께 AWS Database Encryption SDK for DynamoDB를 사용하거나 기존 Amazon DynamoDB 테이블을 최신 버전의 AWS Database Encryption SDK for DynamoDB로 마이그레이션할 수 있습니다.

- 데이터가 전송 및 저장 시 보호되며, 를 포함한 제3자에게 절대 노출되지 않습니다 AWS.
- 테이블 항목에 서명할 수 있습니다. AWS Database Encryption SDK for DynamoDB에 프라이머리 키 속성을 포함하여 테이블 항목 전체 또는 일부에 대한 서명을 계산하도록 지시할 수 있습니다. 이 서명을 사용하면 속성 추가 또는 삭제, 속성 값 바꾸기 등 항목에 대한 무단 변경 내용을 전체적으로 감지할 수 있습니다.
- 키링을 선택하여 데이터 보호 방법을 결정합니다. 키링에 따라 데이터 키, 궁극적으로 데이터를 보호하는 래핑 키가 결정됩니다. 작업에 가장 적합한 가장 안전한 래핑 키를 사용하세요.
- AWS Database Encryption SDK for DynamoDB는 전체 테이블을 암호화하지 않습니다. 항목에서 암호화할 속성을 선택할 수 있습니다. AWS Database Encryption SDK for DynamoDB는 전체 항목을 암호화하지 않습니다. 속성 이름 또는 프라이머리 키(파티션 키 및 정렬 키) 속성의 이름 또는 값은 암호화하지 않습니다.

AWS Encryption SDK

DynamoDB에 저장하는 데이터를 암호화하는 경우 AWS Database Encryption SDK for DynamoDB를 사용하는 것이 좋습니다.

AWS Encryption SDK는 일반 데이터를 암호화 및 복호화할 수 있도록 도와주는 클라이언트측 암호화 라이브러리입니다. 모든 유형의 데이터를 보호할 수 있지만 데이터베이스 레코드와 같은 정형 데이터에는 사용할 수 있도록 설계되지 않았습니다. DynamoDB용 AWS Database Encryption SDK와 달리는 항목 수준 무결성 검사를 제공할 AWS Encryption SDK 수 없으며 속성을 인식하거나 기본 키의 암호화를 방지하는 로직이 없습니다.

AWS Encryption SDK 를 사용하여 테이블의 요소를 암호화하는 경우 DynamoDB용 AWS Database Encryption SDK와 호환되지 않습니다. 한 라이브러리로는 암호화하고 다른 라이브러리로는 복호화할 수 없습니다.

암호화 및 서명되는 필드

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

AWS Database Encryption SDK for DynamoDB는 Amazon DynamoDB 애플리케이션용으로 특별히 설계된 클라이언트 측 암호화 라이브러리입니다. Amazon DynamoDB는 항목 모음인 테이블에 데이터를 저장합니다. 각 항목은 속성 모음입니다. 각 속성마다 이름과 값이 있습니다. AWS Database Encryption SDK for DynamoDB는 속성 값을 암호화합니다. 그런 다음 속성에 대한 서명을 계산합니다. 암호화할 속성 값과 서명에 포함할 속성 값을 지정할 수 있습니다.

암호화는 속성 값의 기밀성을 보호합니다. 서명은 서명된 모든 속성과 속성 간 관계의 무결성을 제공하고 인증을 제공합니다. 속성을 추가 또는 삭제하거나 암호화된 값을 다른 값으로 대체하는 등 항목 전체에 대한 무단 변경을 탐지할 수 있습니다.

암호화된 항목에서 테이블 이름, 모든 속성 이름, 암호화하지 않은 속성 값, 프라이머리 키(파티션 키 및 정렬 키) 속성의 이름과 값, 속성 유형을 포함한 일부 데이터는 일반 텍스트로 유지됩니다. 이 필드에는 민감한 데이터를 저장하지 마세요.

AWS Database Encryption SDK for DynamoDB의 작동 방식에 대한 자세한 내용은 섹션을 참조하세요AWS Database Encryption SDK 작동 방식.

> **Note**
>
> AWS Database Encryption SDK for DynamoDB 주제의 속성 작업에 대한 모든 언급은 암호화 작업을 참조합니다.

주제

- 속성 값 암호화
- 항목에 서명

속성 값 암호화

AWS Database Encryption SDK for DynamoDB는 지정한 속성의 값(속성 이름 또는 유형은 아님)을 암호화합니다. 암호화된 속성 값을 확인하려면 속성 작업을 사용합니다.

예를 들어 이 항목에는 `example` 및 `test` 속성이 포함됩니다.

```
'example': 'data',
'test': 'test-value',
...
```

`example` 속성을 암호화하지만 `test` 속성을 암호화하지 않으면 결과는 다음과 같습니다. 암호화된 `example` 속성 값은 문자열이 아닌 이진 데이터입니다.

```
'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb\x9fY
\x9f\xf3\xc9C\x83\r\xbb\\"),
'test': 'test-value'
...
```

각 항목의 프라이머리 키 속성(파티션 키 및 정렬 키)은 DynamoDB가 테이블에서 항목을 찾는 데 사용하기 때문에 일반 텍스트로 유지되어야 합니다. 서명은 해야 하지만 암호화해서는 안 됩니다.

AWS Database Encryption SDK for DynamoDB는 기본 키 속성을 식별하고 값이 서명되었지만 암호화되지 않았는지 확인합니다. 그리고 프라이머리 키를 식별한 다음 암호화하려고 하면 클라이언트에서 예외가 발생합니다.

클라이언트는 항목에 추가하는 새 속성(aws_dbe_head)에 자료 설명을 저장합니다. 자료 설명은 항목이 암호화되고 서명된 방법을 설명합니다. 클라이언트는 이 정보를 사용하여 항목을 확인하고 암호를 복호화합니다. 자료 설명을 저장하는 필드는 암호화되지 않습니다.

항목에 서명

지정된 속성 값을 암호화한 후 AWS Database Encryption SDK for DynamoDB는 자료 설명, 암호화 컨텍스트 및 속성 작업`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`에서 `ENCRYPT_AND_SIGN`, `SIGN_ONLY`또는 로 표시된 각 필드의 표준화를 통해 해시 기반 메시지 인증 코드(HMACs)와 디지털 서명을 계산합니다. ECDSA 서명은 기본적으로 활성화되어 있지만 필수는 아닙니다. 클라이언트는 항목에 추가하는 새 속성(`aws_dbe_foot`)에 HMAC와 서명을 저장합니다.

DynamoDB의 검색 가능한 암호화

검색 가능한 암호화를 위해 Amazon DynamoDB 테이블을 구성하려면 AWS KMS 계층 키링을 사용하여 항목을 보호하는 데 사용되는 데이터 키를 생성, 암호화 및 복호화해야 합니다. 또한 테이블 암호화 구성에 `SearchConfig`를 포함해야 합니다.

> **Note**
>
> DynamoDB용 Java 클라이언트 측 암호화 라이브러리를 사용하는 경우 DynamoDB API용 하위 수준 AWS Database Encryption SDK를 사용하여 테이블 항목을 암호화, 서명, 확인 및 해독해야 합니다. DynamoDB Enhanced Client와 하위 수준 `DynamoDBItemEncryptor`은 검색 가능한 암호화를 지원하지 않습니다.

주제

비컨을 사용한 보조 인덱스 구성

비컨를 구성한 후 암호화된 속성을 검색하려면 먼저 각 비컨을 반영하는 보조 인덱스를 구성해야 합니다.

표준 또는 복합 비컨을 구성하면 AWS Database Encryption SDK가 비컨 이름에 `aws_dbe_b_` 접두사를 추가하여 서버가 비컨을 쉽게 식별할 수 있도록 합니다. 예를 들어 복합 비컨 `compoundBeacon`의 이름을 지정하면 실제로는 전체 비컨 이름이 `aws_dbe_b_compoundBeacon`가 됩니다. 표준 또는 복합 비컨을 포함하는 보조 인덱스를 구성하려면 비컨 이름을 식별할 때 `aws_dbe_b_` 접두사를 포함해야 합니다.

파티션 키와 정렬 키

프라이머리 키 값은 암호화할 수 없습니다. 파티션 및 정렬 키에 서명해야 합니다. 프라이머리 키 값은 표준 또는 복합 비컨이 될 수 없습니다.

`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 속성을 지정`SIGN_ONLY`하지 않는 한 기본 키 값은 이어야 하며, 파티션 및 정렬 속성도 여야 합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

프라이머리 키 값은 서명된 비컨일 수 있습니다. 각 프라이머리 키 값에 대해 고유한 서명된 비컨을 구성한 경우 프라이머리 키 값을 서명된 비컨 이름으로 식별하는 속성 이름을 지정해야 합니다. 그러나 AWS Database Encryption SDK는 서명된 비컨에 `aws_dbe_b_` 접두사를 추가하지 않습니다. 프라이머리 키 값에 대해 고유한 서명된 비컨를 구성한 경우에도 보조 인덱스를 구성할 때 프라이머리 키 값의 속성 이름만 지정하면 됩니다.

로컬 보조 인덱스

로컬 보조 인덱스의 정렬 키는 비컨일 수 있습니다.

정렬 키에 비컨를 지정하는 경우 유형은 문자열이어야 합니다. 정렬 키에 표준 또는 복합 비컨를 지정하는 경우 비컨 이름을 지정할 때 aws_dbe_b_ 접두사를 포함해야 합니다. 서명된 비컨를 지정하는 경우 접두사 없이 비컨 이름을 지정합니다.

글로벌 보조 인덱스

글로벌 보조 인덱스의 파티션 키와 정렬 키는 모두 비컨일 수 있습니다.

파티션이나 정렬 키에 비컨을 지정하는 경우 유형은 문자열이어야 합니다. 정렬 키에 표준 또는 복합 비컨를 지정하는 경우 비컨 이름을 지정할 때 aws_dbe_b_ 접두사를 포함해야 합니다. 서명된 비컨를 지정하는 경우 접두사 없이 비컨 이름을 지정합니다.

속성 프로젝션

프로젝션은 테이블에서 보조 인덱스로 복사되는 속성 집합입니다. 테이블의 파티션 키와 정렬 키는 항상 인덱스로 프로젝션되지만, 다른 속성을 프로젝션하여 애플리케이션의 쿼리 요건을 지원하는 것도 가능합니다. DynamoDB는 속성 프로젝션을 위한 세 가지 옵션(KEYS_ONLY, INCLUDE, 및 ALL)을 제공합니다.

INCLUDE 속성 프로젝션을 사용하여 비컨을 검색하는 경우 비컨을 구성하는 모든 속성의 이름과 aws_dbe_b_ 접두사를 포함한 비컨 이름을 지정해야 합니다. 예를 들어 field1, field2, 및 field3에서 복합 비콘 compoundBeacon을 구성한 경우 프로젝션에서 aws_dbe_b_compoundBeacon, field1, field2, 및 field3를 지정해야 합니다.

글로벌 보조 인덱스는 프로젝션에 명시적으로 지정된 속성만 사용할 수 있지만 로컬 보조 인덱스는 모든 속성을 사용할 수 있습니다.

비컨 출력 테스트

복합 비컨을 구성하거나 가상 필드를 사용하여 비컨을 구성한 경우 DynamoDB 테이블을 채우기 전에 이러한 비컨이 예상 출력을 생성하는지 확인하는 것이 좋습니다.

AWS Database Encryption SDK는 가상 필드 및 복합 비컨 출력 문제를 해결하는 데 도움이 되는 DynamoDbEncryptionTransforms 서비스를 제공합니다.

가상 필드 테스트

다음 코드 조각은 테스트 항목을 생성하고, DynamoDB 테이블 암호화 구성으로 `DynamoDbEncryptionTransforms` 서비스를 정의하고,를 사용하여 가상 필드가 예상 출력을 생성하는지 `ResolveAttributes` 확인하는 방법을 보여줍니다.

Java

전체 코드 샘플: VirtualBeaconSearchableEncryptionExample.java 참조

```
// Create test items
final PutItemRequest itemWithHasTestResultPutRequest = PutItemRequest.builder()
    .tableName(ddbTableName)
    .item(itemWithHasTestResult)
    .build();

final PutItemResponse itemWithHasTestResultPutResponse =
 ddb.putItem(itemWithHasTestResultPutRequest);

final PutItemRequest itemWithNoHasTestResultPutRequest = PutItemRequest.builder()
    .tableName(ddbTableName)
    .item(itemWithNoHasTestResult)
    .build();

final PutItemResponse itemWithNoHasTestResultPutResponse =
 ddb.putItem(itemWithNoHasTestResultPutRequest);

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
    .Item(itemWithHasTestResult)
    .Version(1)
    .build();
final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that VirtualFields has the expected value
Map<String, String> vf = new HashMap<>();
vf.put("stateAndHasTestResult", "CAt");
assert resolveOutput.VirtualFields().equals(vf);
```

C# / .NET

전체 코드 샘플: VirtualBeaconSearchableEncryptionExample.cs를 참조하세요.
VirtualBeaconSearchableEncryptionExample.cs

```
 // Create item with hasTestResult=true
var itemWithHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("ABC-123"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = true }
};

// Create item with hasTestResult=false
var itemWithNoHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("DEF-456"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = false }
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);

// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
    Item = itemWithHasTestResult,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that VirtualFields has the expected value
Debug.Assert(resolveOutput.VirtualFields.Count == 1);
Debug.Assert(resolveOutput.VirtualFields["stateAndHasTestResult"] == "CAt");
```

Rust

전체 코드 샘플: virtual_beacon_searchable_encryption.rs를 참조하세요. https://github.com/aws/aws-database-encryption-sdk-dynamodb/blob/main/releases/rust/db_esdk/examples/searchableencryption/virtual_beacon_searchable_encryption.rs

```
// Create item with hasTestResult=true
let item_with_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("ABC-123".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(true)),
]);

// Create item with hasTestResult=false
let item_with_no_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("DEF-456".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(false)),
]);

// Define the transform service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;

// Verify the configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item_with_has_test_result.clone())
    .version(1)
```

```
    .send()
    .await?;

// Verify that VirtualFields has the expected value
let virtual_fields = resolve_output.virtual_fields.unwrap();
assert_eq!(virtual_fields.len(), 1);
assert_eq!(virtual_fields["stateAndHasTestResult"], "CAt");
```

복합 비컨 테스트

다음 코드 조각은 테스트 항목을 생성하고, DynamoDB 테이블 암호화 구성으로 `DynamoDbEncryptionTransforms` 서비스를 정의하고,를 `ResolveAttributes` 사용하여 복합 비컨이 예상 출력을 생성하는지 확인하는 방법을 보여줍니다.

Java

전체 코드 샘플: CompoundBeaconSearchableEncryptionExample.java 참조

```
// Create an item with both attributes used in the compound beacon.
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("work_id", AttributeValue.builder().s("9ce39272-8068-4efd-a211-
cd162ad65d4c").build());
item.put("inspection_date", AttributeValue.builder().s("2023-06-13").build());
item.put("inspector_id_last4", AttributeValue.builder().s("5678").build());
item.put("unit", AttributeValue.builder().s("011899988199").build());

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
    .Item(item)
    .Version(1)
    .build();

final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that CompoundBeacons has the expected value
Map<String, String> cbs = new HashMap<>();
cbs.put("last4UnitCompound", "L-5678.U-011899988199");
```

```
assert resolveOutput.CompoundBeacons().equals(cbs);
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

C# / .NET

전체 코드 샘플: CompoundBeaconSearchableEncryptionExample.cs 참조

```
// Create an item with both attributes used in the compound beacon
var item = new Dictionary<String, AttributeValue>
{
    ["work_id"] = new AttributeValue("9ce39272-8068-4efd-a211-cd162ad65d4c"),
    ["inspection_date"] = new AttributeValue("2023-06-13"),
    ["inspector_id_last4"] = new AttributeValue("5678"),
    ["unit"] = new AttributeValue("011899988199")
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);

// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
    Item = item,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);


// Verify that CompoundBeacons has the expected value
Debug.Assert(resolveOutput.CompoundBeacons.Count == 1);
Debug.Assert(resolveOutput.CompoundBeacons["last4UnitCompound"] ==
 "L-5678.U-011899988199");
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

Rust

전체 코드 샘플: compound_beacon_searchable_encryption.rs를 참조하세요. https://github.com/aws/aws-database-encryption-sdk-dynamodb/blob/main/releases/rust/db_esdk/examples/searchableencryption/compound_beacon_searchable_encryption.rs

```
// Create an item with both attributes used in the compound beacon
let item = HashMap::from([
    (
        "work_id".to_string(),
        AttributeValue::S("9ce39272-8068-4efd-a211-cd162ad65d4c".to_string()),
    ),
    (
        "inspection_date".to_string(),
        AttributeValue::S("2023-06-13".to_string()),
    ),
    (
        "inspector_id_last4".to_string(),
        AttributeValue::S("5678".to_string()),
    ),
    (
        "unit".to_string(),
        AttributeValue::S("011899988199".to_string()),
    ),
]);

// Define the transforms service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;

// Verify configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item.clone())
    .version(1)
    .send()
    .await?;

// Verify that CompoundBeacons has the expected value
Dlet compound_beacons = resolve_output.compound_beacons.unwrap();
assert_eq!(compound_beacons.len(), 1);
assert_eq!(
    compound_beacons["last4UnitCompound"],
```

```
    "L-5678.U-011899988199"
);
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

데이터 모델 업데이트

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

DynamoDB용 AWS Database Encryption SDK를 구성할 때 속성 작업을 제공합니다. 암호화 시 AWS 데이터베이스 암호화 SDK는 속성 작업을 사용하여 암호화 및 서명할 속성, 서명할 속성(암호화하지 않음), 무시할 속성을 식별합니다. 또한 서명에서 제외되는 속성을 클라이언트에게 명시적으로 알리도록 허용된 무서명 속성을 정의합니다. 복호화 시 AWS Database Encryption SDK는 사용자가 정의한 허용된 서명되지 않은 속성을 사용하여 서명에 포함되지 않은 속성을 식별합니다. 속성 작업은 암호화된 항목에 저장되지 않으며 AWS Database Encryption SDK는 속성 작업을 자동으로 업데이트하지 않습니다.

속성 작업을 신중하게 선택합니다. 확실하지 않은 경우 Encrypt and sign(암호화 및 서명)을 사용합니다. AWS Database Encryption SDK를 사용하여 항목을 보호한 후에는 기존 , ENCRYPT_AND_SIGN SIGN_ONLY또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 로 변경할 수 없습니다DO_NOTHING. 그러나 다음과 같이 안전하게 변경할 수 있습니다.

- 새 ENCRYPT_AND_SIGN, SIGN_ONLY및 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성 추가
- 기존 속성 제거
- 기존 ENCRYPT_AND_SIGN 속성을 SIGN_ONLY 또는 로 변경 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT
- 기존 SIGN_ONLY 또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 로 변경 ENCRYPT_AND_SIGN
- 새 DO_NOTHING 속성 추가
- 기존 SIGN_ONLY 속성을 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT로 변경합니다.
- 기존 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 SIGN_ONLY로 변경합니다.

검색 가능한 암호화에 대한 고려 사항

데이터 모델을 업데이트하기 전에 업데이트가 속성으로 구성한 모든 비컨에 어떤 영향을 미칠 수 있는지 신중하게 고려해야 합니다. 비컨으로 새 레코드를 작성한 후에는 비컨의 구성을 업데이트할 수 없습니다. 비컨을 구성하는 데 사용한 속성과 관련된 속성 작업은 업데이트할 수 없습니다. 기존 속성 및 관련 비컨을 제거하면 해당 비컨을 사용하여 기존 레코드를 쿼리할 수 없습니다. 레코드에 추가하는 새 필드에 대해 새 비컨을 만들 수 있지만 새 필드를 포함하도록 기존 비컨을 업데이트할 수는 없습니다.

SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성에 대한 고려 사항

기본적으로 파티션 및 정렬 키는 암호화 컨텍스트에 포함된 유일한 속성입니다. AWS KMS 계층적 키링의 브랜치 키 ID 공급자가 암호화 컨텍스트에서 복호화하는 데 필요한 브랜치 키를 식별할 수 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 있도록 추가 필드를 로 정의하는 것이 좋습니다. 자세한 내용은 브랜치 키 ID 공급자를 참조하세요. SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 암호화 작업을 사용하려면 AWS Database Encryption SDK 버전 3.3 이상을 사용해야 합니다. 를 포함하도록 데이터 모델을 업데이트하기 전에 모든 리더에 새 버전을 배포합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

새 **ENCRYPT_AND_SIGN**, **SIGN_ONLY**및 **SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT** 속성 추가

새 ENCRYPT_AND_SIGN, SIGN_ONLY또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 추가하려면 속성 작업에서 새 속성을 정의합니다.

기존 DO_NOTHING 속성을 제거하고 ENCRYPT_AND_SIGN, SIGN_ONLY또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성으로 다시 추가할 수 없습니다.

주석이 달린 데이터 클래스 사용

TableSchema를 사용하여 속성 작업을 정의한 경우 주석이 달린 데이터 클래스에 새 속성을 추가합니다. 새 속성에 속성 작업 주석을 지정하지 않으면 클라이언트는 기본적으로 새 속성을 암호화하고 서명합니다(속성이 프라이머리 키의 일부

가 아닌 경우). 새 속성에만 서명하려면 @DynamoDBEncryptionSignOnly 또는 @DynamoDBEncryptionSignAndIncludeInEncryptionContext 주석을 사용하여 새 속성을 추가해야 합니다.

객체 모델 사용

속성 작업을 수동으로 정의한 경우 객체 모델의 속성 작업에 새 속성을 추가하고 ENCRYPT_AND_SIGN, SIGN_ONLY또는를 속성 작업SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT으로 지정합니다.

기존 속성 제거

속성이 더 이상 필요하지 않다고 판단되면 해당 속성에 데이터 쓰기를 중단하거나 속성 작업에서 해당 속성을 공식적으로 제거할 수 있습니다. 속성에 새 데이터 쓰기를 중지해도 해당 속성은 여전히 속성 작업에 표시됩니다. 이는 나중에 속성 사용을 다시 시작해야 하는 경우에 유용할 수 있습니다. 속성 작업에서 속성을 정식으로 제거해도 데이터 세트에서 제거되지는 않습니다. 데이터 세트에는 해당 속성이 포함된 항목이 계속 포함됩니다.

기존 ENCRYPT_AND_SIGN, SIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, 또는 DO_NOTHING 속성을 공식적으로 제거하려면 속성 작업을 업데이트합니다.

DO_NOTHING 속성을 제거하는 경우 허용된 무서명 속성에서 해당 속성을 제거해서는 안 됩니다. 더 이상 해당 속성에 새 값을 쓰지 않아도 클라이언트가 속성이 포함된 기존 항목을 읽으려면 속성이 서명되지 않았음을 알아야 합니다.

주석이 달린 데이터 클래스 사용

TableSchema를 사용하여 속성 작업을 정의한 경우 주석이 달린 데이터 클래스에서 해당 속성을 제거합니다.

객체 모델 사용

속성 동작을 수동으로 정의한 경우 객체 모델의 속성 작업에서 속성을 제거합니다.

기존 **ENCRYPT_AND_SIGN** 속성을 **SIGN_ONLY** 또는 로 변경 **SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT**

기존 ENCRYPT_AND_SIGN 속성을 SIGN_ONLY 또는 로 변경하려면 속성 작업을 업데이트SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT해야 합니다. 업데이트를 배포한 후 클라이언트는 속성에 레코드된 기존 값을 확인하고 복호화할 수 있지만 속성에 기록된 새 값에만 서명합니다.

> **Note**
>
> 기존 ENCRYPT_AND_SIGN 속성을 SIGN_ONLY 또는 로 변경하기 전에 보안 요구 사항을 신중하게 고려하세요SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT. 민감한 데이터를 저장할 수 있는 모든 속성은 암호화되어야 합니다.

주석이 달린 데이터 클래스 사용

를 사용하여 속성 작업을 정의한 경우 주석이 달린 데이터 클래스에 @DynamoDBEncryptionSignOnly 또는 @DynamoDBEncryptionSignAndIncludeInEncryptionContext 주석을 포함하도록 기존 속성을 TableSchema업데이트합니다.

객체 모델 사용

속성 작업을 수동으로 정의한 경우 객체 모델에서 기존 속성과 연결된 속성 작업을에서 SIGN_ONLY 또는 ENCRYPT_AND_SIGN 로 업데이트SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT합니다.

기존 SIGN_ONLY 또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 로 변경 ENCRYPT_AND_SIGN

기존 SIGN_ONLY 또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 로 변경하려면 속성 작업을 업데이트ENCRYPT_AND_SIGN해야 합니다. 업데이트를 배포한 후 클라이언트는 속성에 레코드된 기존 값을 확인하고 속성에 기록된 새 값을 암호화하고 서명할 수 있습니다.

주석이 달린 데이터 클래스 사용

를 사용하여 속성 작업을 정의한 경우 기존 속성에서 @DynamoDBEncryptionSignOnly 또는 @DynamoDBEncryptionSignAndIncludeInEncryptionContext 주석을 TableSchema제거합니다.

객체 모델 사용

속성 작업을 수동으로 정의한 경우 객체 모델의 SIGN_ONLY 또는 ENCRYPT_AND_SIGN에서 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT로 속성과 연결된 속성 작업을 업데이트합니다.

새 DO_NOTHING 속성 추가

새 DO_NOTHING 속성을 추가할 때 오류가 발생할 위험을 줄이려면 DO_NOTHING 속성의 이름을 지정할 때 고유한 접두사를 지정한 다음 해당 접두사를 사용하여 허용되는 무서명 속성을 정의하는 것이 좋습니다.

주석이 달린 데이터 클래스에서 기존 ENCRYPT_AND_SIGNSIGN_ONLY, 또는 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 제거한 다음 속성을 속성으로 다시 추가할 수 없습니다DO_NOTHING. 완전히 새로운 DO_NOTHING 속성만 추가할 수 있습니다.

새 DO_NOTHING 속성을 추가하는 단계는 허용된 서명되지 않은 속성을 목록에 명시적으로 정의했는지 아니면 접두사를 사용하여 정의했는지에 따라 달라집니다.

허용된 무서명 속성 접두사 사용

TableSchema을 사용하여 속성 작업을 정의한 경우 @DynamoDBEncryptionDoNothing 주석을 사용하여 주석이 달린 데이터 클래스에 새 DO_NOTHING 속성을 추가합니다. 속성 작업을 수동으로 정의한 경우 새 속성을 포함하도록 속성 작업을 업데이트합니다. DO_NOTHING 속성 작업을 사용하여 새 속성을 명시적으로 구성해야 합니다. 새 속성 이름에 동일한 고유 접두사를 포함해야 합니다.

허용된 무서명 속성 목록 사용

1. 허용된 무서명 속성 목록에 새 DO_NOTHING 속성을 추가하고 업데이트된 목록을 배포합니다.
2. 1단계에서 변경한 내용을 배포합니다.

   이 데이터를 읽어야 하는 모든 호스트에 변경 내용이 전파되기 전까지는 3단계로 넘어갈 수 없습니다.
3. 새 DO_NOTHING 속성을 속성 작업에 추가합니다.

   a. TableSchema을 사용하여 속성 작업을 정의한 경우 @DynamoDBEncryptionDoNothing 주석을 사용하여 주석이 달린 데이터 클래스에 새 DO_NOTHING 속성을 추가합니다.

   b. 속성 작업을 수동으로 정의한 경우 새 속성을 포함하도록 속성 작업을 업데이트합니다. DO_NOTHING 속성 작업을 사용하여 새 속성을 명시적으로 구성해야 합니다.
4. 3단계에서 변경한 내용을 배포합니다.

기존 SIGN_ONLY 속성을 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT로 변경합니다.

기존 SIGN_ONLY 속성을 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT로 변경하려면 속성 작업을 업데이트해야 합니다. 업데이트를 배포한 후 클라이언트는 속성에 기록된 기존 값을 확인할 수 있으며 속성에 기록된 새 값에 계속 서명합니다. 속성에 기록된 새 값은 암호화 컨텍스트에 포함됩니다.

SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

주석이 달린 데이터 클래스 사용

를 사용하여 속성 작업을 정의한 경우 속성과 연결된 속성 작업을에서 로 TableSchema업데이트@DynamoDBEncryptionSignOnly합니다@DynamoDBEncryptionSignAndIncludeInEncryptionContext.

객체 모델 사용

속성 작업을 수동으로 정의한 경우, 객체 모델에서 속성과 연관된 속성 작업을 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 에서 SIGN_ONLY 로 업데이트합니다.

기존 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 SIGN_ONLY로 변경합니다.

기존 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 SIGN_ONLY로 변경하려면 속성 작업을 업데이트해야 합니다. 업데이트를 배포한 후 클라이언트는 속성에 기록된 기존 값을 확인할 수 있으며 속성에 기록된 새 값에 계속 서명합니다. 속성에 기록된 새 값은 암호화 컨텍스트에 포함되지 않습니다.

기존 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 로 변경하기 전에 업데이트가 브랜치 키 ID 공급자의 기능에 어떤 영향을 미칠 수 있는지 SIGN_ONLY신중하게 고려하세요.

주석이 달린 데이터 클래스 사용

를 사용하여 속성 작업을 정의한 경우 속성과 연결된 속성 작업을에서 로 TableSchema업데이트@DynamoDBEncryptionSignAndIncludeInEncryptionContext합니다@DynamoDBEncryptionSignOnly.

객체 모델 사용

속성 작업을 수동으로 정의한 경우, 객체 모델에서 속성과 연관된 속성 작업을 `SIGN_ONLY` 에서 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 로 업데이트합니다.

AWS Database Encryption SDK for DynamoDB 사용 가능한 프로그래밍 언어

DynamoDB용 AWS Database Encryption SDK는 다음 프로그래밍 언어에 사용할 수 있습니다. 언어별 라이브러리는 다양하지만 결과 구현은 상호 운용이 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.

주제

- Java
- .NET
- Rust

Java

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

이 주제에서는 DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x를 설치하고 사용하는 방법을 설명합니다. DynamoDB용 AWS Database Encryption SDK를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에서 Java 예제를 참조하세요.

> **Note**
>
> 다음 주제에서는 DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x에 중점을 둡니다.
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. AWS Database Encryption SDK는 레거시 DynamoDB Encryption Client 버전을 계속 지원합니다.

주제

사전 조건

버전 3.x을 설치하기 전에 DynamoDB용 Java 클라이언트측 암호화 라이브러리의 경우 다음과 같은 사전 요구 사항이 있는지 확인합니다.

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.

AWS SDK for Java 2.x

AWS Database Encryption SDK for DynamoDB에는의 DynamoDB 향상된 클라이언트 모듈이 필요합니다 AWS SDK for Java 2.x. 전체 SDK를 설치하거나 이 모듈만 설치할 수 있습니다.

버전 업데이트에 대한 자세한 내용은의 버전 1.x에서 2.x로 마이그레이션을 AWS SDK for Java참조하세요. AWS SDK for Java

AWS SDK for Java 는 Apache Maven을 통해 사용할 수 있습니다. 전체 AWS SDK for Java또는 `dynamodb-enhanced` 모듈에 대한 종속성을 선언할 수 있습니다.

Apache Maven을 AWS SDK for Java 사용하여 설치

- 전체 AWS SDK for Java를 종속성으로 가져오려면 `pom.xml` 파일에 선언하세요.
- AWS SDK for Java에서 Amazon DynamoDB 모듈에 대해서만 종속성을 생성하려면 특정 모듈을 지정하는 지침을 따릅니다. `groupId`를 `software.amazon.awssdk`로, `artifactID`를 `dynamodb-enhanced`로 설정합니다.

> **Note**
>
> AWS KMS 키링 또는 AWS KMS 계층적 키링을 사용하는 경우 AWS KMS 모듈에 대한 종속성도 생성해야 합니다. `groupId`를 `software.amazon.awssdk`로, `artifactID`를 `kms`로 설정합니다.

설치

다음 방법으로 DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x를 설치할 수 있습니다.

Apache Maven 사용

Amazon DynamoDB Encryption Client for Java는 다음 종속성 정의와 함께 Apache Maven을 통해 사용할 수 있습니다.

```
<dependency>
  <groupId>software.amazon.cryptography</groupId>
  <artifactId>aws-database-encryption-sdk-dynamodb</artifactId>
  <version>version-number</version>
</dependency>
```

Gradle Kotlin 사용

Gradle을 사용하면 Gradle 프로젝트의 종속성 섹션에 다음을 추가하여 Java용 Amazon DynamoDB Encryption Client에 대한 종속성을 선언할 수 있습니다.

```
implementation("software.amazon.cryptography:aws-database-encryption-sdk-
dynamodb:version-number")
```

직접

DynamoDB용 Java 클라이언트 측 암호화 라이브러리를 설치하려면 aws-database-encryption-sdk-dynamodb GitHub 리포지토리를 복제하거나 다운로드합니다.

SDK를 설치한 후이 가이드의 예제 코드와 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에 있는 Java 예제를 살펴보면서 시작합니다.

DynamoDB용 Java 클라이언트측 암호화 라이브러리 사용

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

이 주제에서는 DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x의 일부 기능과 헬퍼 클래스에 대해 설명합니다.

DynamoDB용 Java 클라이언트 측 암호화 라이브러리를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에 있는 Java 예제인 Java 예제를 참조하세요.

주제

- 항목 암호화 도구
- AWS Database Encryption SDK for DynamoDB의 속성 작업
- AWS Database Encryption SDK for DynamoDB의 암호화 구성
- AWS Database Encryption SDK로 항목 업데이트
- 서명된 집합 암호 해독

항목 암호화 도구

코어에서 AWS Database Encryption SDK for DynamoDB는 항목 암호화 도구입니다. DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x 를 사용하여 다음과 같은 방법으로 DynamoDB 테이블 항목을 암호화, 서명, 확인 및 복호화할 수 있습니다.

DynamoDB Enhanced Client

`DynamoDbEncryptionInterceptor`를 사용하여 DynamoDB `PutItem` 요청에 따라 클라이언트 측에서 항목을 자동으로 암호화하고 서명하도록 DynamoDB Enhanced Client를 구성할 수 있습니다. DynamoDB Enhanced Client에서는 주석이 달린 데이터 클래스를 사용하여 속성 작업을 정의할 수 있습니다. 가능하면 DynamoDB Enhanced Client를 사용하는 것이 좋습니다.

DynamoDB Enhanced Client는 검색 가능한 암호화를 지원하지 않습니다.

> **Note**
>
> AWS Database Encryption SDK는 중첩 속성에 대한 주석을 지원하지 않습니다.

하위 수준 DynamoDB API

`DynamoDbEncryptionInterceptor`를 사용하여 DynamoDB `PutItem` 요청에 따라 클라이언트 측에서 항목을 자동으로 암호화하고 서명하도록 하위 수준 DynamoDB API를 구성할 수 있습니다.

검색 가능한 암호화를 사용하려면 하위 수준 DynamoDB API를 사용해야 합니다.

하위 수준 **`DynamoDbItemEncryptor`**

하위 수준 `DynamoDbItemEncryptor`에서는 DynamoDB를 호출하지 않고도 테이블 항목을 직접 암호화하고 서명 또는 복호화하고 확인합니다. DynamoDB `PutItem` 또는 `GetItem` 요청을 하지 않습니다. 예를 들어 하위 수준 `DynamoDbItemEncryptor`을 사용하여 이미 검색한 DynamoDB 항목을 직접 복호화하고 확인할 수 있습니다.

하위 수준 `DynamoDbItemEncryptor`은 검색 가능한 암호화를 지원하지 않습니다.

AWS Database Encryption SDK for DynamoDB의 속성 작업

속성 작업은 암호화 및 서명되는 속성 값, 서명만 되는 속성 값, 암호화 컨텍스트에 서명 및 포함되는 속성 값, 무시되는 속성 값을 결정합니다.

> **Note**
>
> `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 암호화 작업을 사용하려면 AWS Database Encryption SDK 버전 3.3 이상을 사용해야 합니다. 를 포함하도록 데이터 모델을 업데이트하기 전에 모든 리더에 새 버전을 배포합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

하위 수준 DynamoDB API 또는 하위 수준 `DynamoDbItemEncryptor`을 사용하는 경우 속성 작업을 수동으로 정의해야 합니다. DynamoDB Enhanced Client를 사용하는 경우 속성 작업을 수동으로 정의하거나 주석이 달린 데이터 클래스를 사용하여 `TableSchema`을 생성할 수 있습니다. 구성 프로세스를 단순화하려면 주석이 달린 데이터 클래스를 사용하는 것이 좋습니다. 주석이 달린 데이터 클래스를 사용하는 경우 객체를 한 번만 모델링하면 됩니다.

> **Note**
>
> 속성 작업을 정의한 후에는 서명에서 제외할 속성을 정의해야 합니다. 나중에 서명되지 않은 새 속성을 더 쉽게 추가할 수 있도록 서명되지 않은 속성을 식별할 고유한 접두사(예: ":“)를 선택하는 것이 좋습니다. DynamoDB 스키마와 속성 작업을 정의할 때 DO_NOTHING로 표시된 모든 속성의 속성 이름에 이 접두사를 포함합니다.

주석이 달린 데이터 클래스 사용

주석이 달린 데이터 클래스를 사용하여 DynamoDB Enhanced Client 및 DynamoDbEncryptionInterceptor에서 속성 작업을 지정합니다. AWS Database Encryption SDK for DynamoDB는 속성 유형을 정의하는 표준 DynamoDB 속성 주석을 사용하여 속성을 보호하는 방법을 결정합니다. 기본적으로 기본 키(서명되지만 암호화되지 않음)를 제외하고는 모든 속성이 암호화 및 서명됩니다.

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 암호화 작업을 사용하려면 AWS Database Encryption SDK 버전 3.3 이상을 사용해야 합니다. 를 포함하도록 데이터 모델을 업데이트하기 전에 모든 리더에 새 버전을 배포합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

DynamoDB 향상된 클라이언트 주석에 대한 자세한 지침은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에서 SimpleClass.java를 참조하세요.

기본적으로 프라이머리 키 속성은 서명되지만 암호화되지는 않으며(SIGN_ONLY) 다른 모든 속성은 암호화되고 서명됩니다(ENCRYPT_AND_SIGN). 속성을 로 정의하는 경우 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT. 예외를 지정하려면 DynamoDB용 Java 클라이언트측 암호화 라이브러리에 정의된 암호화 주석을 사용합니다. 예를 들어, 특정 속성에 서명만 적용하려면 @DynamoDbEncryptionSignOnly 주석을 사용합니다. 특정 속성을 서명하여 암호화 컨텍스트에 포함하려면를 사용합니다@DynamoDbEncryptionSignAndIncludeInEncryptionContext. 특정 속성에 서명되거나 암호화되지 않도록 하려면(DO_NOTHING) @DynamoDbEncryptionDoNothing 주석을 사용합니다.

> **Note**
>
> AWS Database Encryption SDK는 중첩 속성에 대한 주석을 지원하지 않습니다.

다음 예제에서는 ENCRYPT_AND_SIGN, SIGN_ONLY및 DO_NOTHING속성 작업을 정의하는 데 사용되는 주석을 보여줍니다. 를 정의하는 데 사용되는 주석을 보여주는 예제는 SimpleClass4.java를 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT참조하세요.

```
@DynamoDbBean
public class SimpleClass {

    private String partitionKey;
    private int sortKey;
    private String attribute1;
    private String attribute2;
    private String attribute3;

    @DynamoDbPartitionKey
    @DynamoDbAttribute(value = "partition_key")
    public String getPartitionKey() {
        return this.partitionKey;
    }

    public void setPartitionKey(String partitionKey) {
        this.partitionKey = partitionKey;
    }

    @DynamoDbSortKey
    @DynamoDbAttribute(value = "sort_key")
    public int getSortKey() {
        return this.sortKey;
    }

    public void setSortKey(int sortKey) {
        this.sortKey = sortKey;
    }

    public String getAttribute1() {
        return this.attribute1;
    }
```

```
    public void setAttribute1(String attribute1) {
        this.attribute1 = attribute1;
    }

    @DynamoDbEncryptionSignOnly
    public String getAttribute2() {
        return this.attribute2;
    }

    public void setAttribute2(String attribute2) {
        this.attribute2 = attribute2;
    }

    @DynamoDbEncryptionDoNothing
    public String getAttribute3() {
        return this.attribute3;
    }

    @DynamoDbAttribute(value = ":attribute3")
    public void setAttribute3(String attribute3) {
        this.attribute3 = attribute3;
    }

}
```

주석이 달린 데이터 클래스를 사용하여 다음 코드 조각에 표시된 것처럼 TableSchema을 생성합니다.

```
final TableSchema<SimpleClass> tableSchema = TableSchema.fromBean(SimpleClass.class);
```

속성 작업 수동 정의

속성 작업을 수동으로 지정하려면 이름-값 페어가 속성 이름과 지정된 작업을 나타내는 Map 객체를 만듭니다.

속성을 암호화하고 서명하도록 ENCRYPT_AND_SIGN을 지정합니다. 속성을 서명하되 암호화하지 않도록 SIGN_ONLY을 지정합니다. 를 지정SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT하여 속성에 서명하고 암호화 컨텍스트에 포함합니다. 서명하지 않으면 속성을 암호화할 수 없습니다. 속성을 무시하도록 DO_NOTHING을 지정합니다.

파티션 및 정렬 속성은 SIGN_ONLY 또는 여야 합니
다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT. 속성을 로 정의하는 경우

SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 암호화 작업을 사용하려면 AWS Database Encryption SDK 버전 3.3 이상을 사용해야 합니다. 를 포함하도록 데이터 모델을 업데이트하기 전에 모든 리더에 새 버전을 배포합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be signed
attributeActionsOnEncrypt.put("partition_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
// The sort attribute must be signed
attributeActionsOnEncrypt.put("sort_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute3",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put(":attribute4", CryptoAction.DO_NOTHING);
```

AWS Database Encryption SDK for DynamoDB의 암호화 구성

AWS Database Encryption SDK를 사용하는 경우 DynamoDB 테이블에 대한 암호화 구성을 명시적으로 정의해야 합니다. 암호화 구성에 필요한 값은 속성 작업을 수동으로 정의했는지 아니면 주석이 달린 데이터 클래스를 사용하여 정의했는지에 따라 달라집니다.

다음 스니펫은 DynamoDB Enhanced Client `TableSchema`를 사용하는 DynamoDB 테이블 암호화 구성을 정의하고 고유한 접두사로 정의된 서명되지 않은 속성을 허용합니다.

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
        DynamoDbEnhancedTableEncryptionConfig.builder()
            .logicalTableName(ddbTableName)
            .keyring(kmsKeyring)
            .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
            .schemaOnEncrypt(tableSchema)
```

```
            // Optional: only required if you use beacons
            .search(SearchConfig.builder()
                    .writeVersion(1) // MUST be 1
                    .versions(beaconVersions)
                    .build())
            .build());
```

논리적 테이블 이름

DynamoDB 테이블의 논리적 테이블 이름.

논리적 테이블 이름은 테이블에 저장된 모든 데이터에 암호로 바인딩되어 DynamoDB 복원 작업을 간소화합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 항상 같은 논리적 테이블 이름을 지정해야 합니다. 복호화이 성공하려면 논리적 테이블 이름이 암호화에 지정된 이름과 일치해야 합니다. 백업에서 DynamoDB 테이블을 복원한 후 DynamoDB 테이블 이름이 변경되더라도 논리적 테이블 이름을 사용하면 복호화 작업에서 테이블을 계속 인식할 수 있습니다.

허용된 서명되지 않은 속성

속성 작업에 `DO_NOTHING`로 표시된 속성.

허용된 무서명 서명에서 제외되는 속성을 클라이언트에게 알려줍니다. 클라이언트는 다른 모든 속성이 서명에 포함되어 있다고 가정합니다. 그런 다음 레코드를 복호화할 때 클라이언트는 확인해야 할 속성과 지정한 허용된 무서명 속성 중에서 무시할 속성을 결정합니다. 허용된 무서명 속성에서는 속성을 제거할 수 없습니다.

모든 `DO_NOTHING` 속성을 나열하는 배열을 만들어 무서명 허용 속성을 명시적으로 정의할 수 있습니다. `DO_NOTHING` 속성의 이름을 지정할 때 고유한 접두사를 지정하고 이 접두사를 사용하여 무서명 속성을 클라이언트에게 알릴 수도 있습니다. 고유한 접두사를 지정하는 것이 좋습니다. 이렇게 하면 나중에 새 `DO_NOTHING` 속성을 추가하는 프로세스가 단순해지기 때문입니다. 자세한 내용은 데이터 모델 업데이트 단원을 참조하십시오.

모든 `DO_NOTHING` 속성에 접두사를 지정하지 않는 경우 클라이언트가 복호화 시 서명되지 않을 것으로 예상되는 모든 속성을 명시적으로 나열하는 `allowedUnsignedAttributes` 배열을 구성할 수 있습니다. 반드시 필요한 경우에만 허용된 서명되지 않은 속성을 명시적으로 정의해야 합니다.

검색 구성(선택 사항)

`SearchConfig`는 비컨 버전을 정의합니다.

검색 가능한 암호화 또는 서명된 비컨을 사용하려면 `SearchConfig`를 지정해야 합니다.

알고리즘 제품군(선택 사항)

`algorithmSuiteId`은 AWS Database Encryption SDK가 사용하는 알고리즘 제품군을 정의합니다.

대체 알고리즘 제품군을 명시적으로 지정하지 않는 한 AWS Database Encryption SDK는 기본 알고리즘 제품군을 사용합니다. 기본 알고리즘 제품군은 키 도출, 디지털 서명 및 키 커밋과 함께 AES-GCM 알고리즘을 사용합니다. 기본 알고리즘 제품군이 대부분의 애플리케이션에 적합할 가능성이 높지만 대체 알고리즘 제품군을 선택할 수도 있습니다. 예를 들어, 일부 신뢰 모델은 디지털 서명이 없는 알고리즘 제품군으로 충분할 수 있습니다. AWS Database Encryption SDK가 지원하는 알고리즘 제품군에 대한 자세한 내용은 섹션을 참조하세요AWS Database Encryption SDK에서 지원되는 알고리즘 제품군.

ECDSA 디지털 서명이 없는 AES-GCM 알고리즘 제품군을 선택하려면 테이블 암호화 구성에 다음 코드 조각을 포함합니다.

```
.algorithmSuiteId(
    DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

AWS Database Encryption SDK로 항목 업데이트

AWS Database Encryption SDK는 암호화되거나 서명된 항목에 대해 ddb:UpdateItem을 지원하지 않습니다. 암호화되거나 서명된 항목을 업데이트하려면 ddb:PutItem을 사용해야 합니다. `PutItem` 요청에 기존 항목과 동일한 프라이머리 키를 지정하면 새 항목이 기존 항목을 완전히 대체합니다. 항목을 업데이트한 후 CLOBBER를 사용하여 저장 시 모든 속성을 지우고 바꿀 수도 있습니다.

서명된 집합 암호 해독

AWS Database Encryption SDK 버전 3.0.0 및 3.1.0에서는 세트 유형 속성을 로 정의하면 세트의 `SIGN_ONLY`값이 제공된 순서대로 정식화됩니다. DynamoDB는 집합의 순서를 보존하지 않습니다. 따라서 집합을 포함하는 항목의 서명 검증이 실패할 수 있습니다. 집합 속성에 동일한 값이 포함되어 있더라도 집합의 값이 AWS Database Encryption SDK에 제공된 것과 다른 순서로 반환되면 서명 검증이 실패합니다.

> **Note**
>
> AWS Database Encryption SDK 버전 3.1.1 이상에서는 모든 세트 유형 속성의 값을 정규화하므로 DynamoDB에 기록된 것과 동일한 순서로 값을 읽을 수 있습니다.

서명 검증이 실패하는 경우 암호 해독 작업이 실패하고 다음 오류 메시지가 반환됩니다.

```
software.amazon.cryptography.dbencryptionsdk.structuredencryption.model.StructuredEncryptionException: 일치하는 수신자 태그가 없습니다.
```

위의 오류 메시지가 표시되고 암호 해독하려는 항목에 버전 3.0.0 또는 3.1.0을 사용하여 서명된 세트가 포함되어 있다고 생각되는 경우, GitHub에 있는 aws-database-encryption-sdk-dynamodb-java 리포지토리의 DecryptWithPermute 디렉터리에서 집합을 성공적으로 검증하는 방법에 대한 자세한 내용을 참조하세요.

Java 예제

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

다음 예제에서는 DynamoDB용 Java 클라이언트측 암호화 라이브러리를 사용하여 애플리케이션의 테이블 항목을 보호하는 방법을 보여줍니다. GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에 있는 Java 예제에서 더 많은 예제를 찾을 수 있습니다(자체 예제를 제공할 수 있습니다).

다음 예제는 채워지지 않은 새 Amazon DynamoDB 테이블에서 DynamoDB용 Java 클라이언트 측 암호화 라이브러리를 구성하는 방법을 보여줍니다. 클라이언트측 암호화를 위해 기존 Amazon DynamoDB 테이블을 구성하려면 기존 테이블에 버전 3.x 추가 섹션을 참조하세요.

주제

- DynamoDB Enhanced Client 사용
- 하위 수준 DynamoDB API 사용
- 하위 수준 DynamoDbItemEncryptor 사용

DynamoDB Enhanced Client 사용

다음 예제는 DynamoDB API 호출의 일부로 DynamoDB Enhanced Client와 AWS KMS 키링을 포함한 `DynamoDbEncryptionInterceptor`을 사용하여 DynamoDB 테이블 항목을 암호화하는 방법을 보여줍니다.

DynamoDB Enhanced Client에서 지원되는 모든 키링을 사용할 수 있지만 가능하면 AWS KMS 키링 중 하나를 사용하는 것이 좋습니다.

> **Note**
>
> DynamoDB Enhanced Client는 검색 가능한 암호화를 지원하지 않습니다. 하위 수준 DynamoDB API와 함께 `DynamoDbEncryptionInterceptor`을 사용하여 검색 가능한 암호화를 사용할 수 있습니다.

전체 코드 샘플 보기: EnhancedPutGetExample.java

1단계: AWS KMS 키링 생성

다음 예제에서는 `CreateAwsKmsMrkMultiKeyring`를 사용하여 대칭 암호화 KMS AWS KMS 키로 키링을 생성합니다. 이 `CreateAwsKmsMrkMultiKeyring` 방법을 사용하면 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리할 수 있습니다.

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

2단계: 주석이 달린 데이터 클래스에서 테이블 스키마 생성

다음 예제에서는 주석이 달린 데이터 클래스를 사용하여 `TableSchema`를 만듭니다.

이 예제에서는 주석이 달린 데이터 클래스와 속성 작업을 SimpleClass.java를 사용하여 정의했다고 가정합니다. 속성 작업에 주석을 다는 방법에 대한 자세한 지침은 주석이 달린 데이터 클래스 사용을 참조하세요.

> **Note**
>
> AWS Database Encryption SDK는 중첩 속성에 대한 주석을 지원하지 않습니다.

```
final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);
```

3단계: 시그니처에서 제외할 속성을 정의합니다.

다음 예제에서는 모든 DO_NOTHING 속성이 고유한 접두사 ":“를 공유한다고 가정하고 이 접두사를 사용하여 허용된 서명되지 않은 속성을 정의합니다. 클라이언트는 접두사가 ":"인 모든 속성 이름이 서명에서 제외된다고 가정합니다. 자세한 내용은 Allowed unsigned attributes 단원을 참조하십시오.

```
final String unsignedAttrPrefix = ":";
```

4단계: 암호화 구성 생성

다음 예제는 DynamoDB 테이블의 암호화 구성을 나타내는 tableConfigs 맵을 정의합니다.

이 예제에서는 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB의 암호화 구성 단원을 참조하십시오.

> **Note**
>
> 검색 가능한 암호화 또는 서명된 비컨을 사용하려면 암호화 구성에도 SearchConfig을 포함해야 합니다.

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
    DynamoDbEnhancedTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .schemaOnEncrypt(tableSchema)
        .build());
```

5단계: **DynamoDbEncryptionInterceptor** 생성

다음 예제에서는 4단계의 tableConfigs를 사용하여 새 DynamoDbEncryptionInterceptor을 만듭니다.

```
final DynamoDbEncryptionInterceptor interceptor =
```

```
    DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
        CreateDynamoDbEncryptionInterceptorInput.builder()
            .tableEncryptionConfigs(tableConfigs)
            .build()
    );
```

6단계: 새 AWS SDK DynamoDB 클라이언트 생성

다음 예제에서는 5단계`interceptor`의를 사용하여 새 AWS SDK DynamoDB 클라이언트를 생성합니다.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

7단계: DynamoDB Enhanced Client 생성 및 테이블 생성

다음 예제는 6단계에서 생성한 AWS SDK DynamoDB client를 사용하여 DynamoDB Enhanced Client를 생성하고 주석이 달린 데이터 클래스를 사용하여 테이블을 생성합니다.

```
final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
        .build();
final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
```

8단계: 테이블 항목 암호화 및 서명

다음 예제는 DynamoDB Enhanced Client를 사용하여 DynamoDB 테이블에 항목을 추가합니다. 항목은 DynamoDB로 전송되기 전에 클라이언트측에서 암호화되고 서명됩니다.

```
final SimpleClass item = new SimpleClass();
item.setPartitionKey("EnhancedPutGetExample");
item.setSortKey(0);
item.setAttribute1("encrypt and sign me!");
item.setAttribute2("sign me!");
item.setAttribute3("ignore me!");
```

```
table.putItem(item);
```

하위 수준 DynamoDB API 사용

다음 예제는 AWS KMS 키링이 있는 하위 수준 DynamoDB API를 사용하여 DynamoDB `PutItem` 요청으로 클라이언트측에서 항목을 자동으로 암호화하고 서명하는 방법을 보여줍니다.

지원되는 모든 키링을 사용할 수 있지만 가능하면 AWS KMS 키링 중 하나를 사용하는 것이 좋습니다.

전체 코드 샘플 보기: BasicPutGetExample.java

1단계: AWS KMS 키링 생성

다음 예제에서는 `CreateAwsKmsMrkMultiKeyring`를 사용하여 대칭 암호화 KMS AWS KMS 키로 키링을 생성합니다. 이 `CreateAwsKmsMrkMultiKeyring` 방법을 사용하면 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리할 수 있습니다.

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

2단계: 속성 작업 구성

다음 예제에서는 테이블 항목에 대한 샘플 속성 작업을 나타내는 `attributeActionsOnEncrypt` 맵을 정의합니다.

> **Note**
>
> 다음 예제에서는 속성을 로 정의하지 않습니
> 다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.
> `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
```

```
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

3단계: 시그니처에서 제외할 속성을 정의합니다.

다음 예제에서는 모든 DO_NOTHING 속성이 고유한 접두사 ":“를 공유한다고 가정하고 이 접두사를 사용하여 허용된 서명되지 않은 속성을 정의합니다. 클라이언트는 접두사가 ":"인 모든 속성 이름이 서명에서 제외된다고 가정합니다. 자세한 내용은 Allowed unsigned attributes 단원을 참조하십시오.

```
final String unsignedAttrPrefix = ":";
```

4단계: DynamoDB 테이블 암호화 구성 정의

다음 예제는 이 DynamoDB 테이블의 암호화 구성을 나타내는 tableConfigs 맵을 정의합니다.

이 예제에서는 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB의 암호화 구성 단원을 참조하십시오.

> **Note**
>
> 검색 가능한 암호화 또는 서명된 비컨을 사용하려면 암호화 구성에도 SearchConfig을 포함해야 합니다.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

```
tableConfigs.put(ddbTableName, config);
```

5단계: **DynamoDbEncryptionInterceptor** 생성

다음 예제는 4단계의 tableConfigs를 사용하여 DynamoDbEncryptionInterceptor을 생성합니다.

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

6단계: 새 AWS SDK DynamoDB 클라이언트 생성

다음 예제에서는 5단계interceptor의를 사용하여 새 AWS SDK DynamoDB 클라이언트를 생성합니다.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                       .addExecutionInterceptor(interceptor)
                       .build())
        .build();
```

7단계: DynamoDB 테이블 항목 암호화 및 서명

다음 예제는 샘플 테이블 항목을 나타내는 item 맵을 정의하고 해당 항목을 DynamoDB 테이블에 배치합니다. 항목은 DynamoDB로 전송되기 전에 클라이언트측에서 암호화되고 서명됩니다.

```
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("partition_key", AttributeValue.builder().s("BasicPutGetExample").build());
item.put("sort_key", AttributeValue.builder().n("0").build());
item.put("attribute1", AttributeValue.builder().s("encrypt and sign me!").build());
item.put("attribute2", AttributeValue.builder().s("sign me!").build());
item.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final PutItemRequest putRequest = PutItemRequest.builder()
        .tableName(ddbTableName)
        .item(item)
        .build();
```

```
final PutItemResponse putResponse = ddb.putItem(putRequest);
```

하위 수준 DynamoDbItemEncryptor 사용

다음 예제는 AWS KMS 키링이 있는 하위 수준 DynamoDbItemEncryptor을 사용하여 테이블 항목을 직접 암호화하고 서명하는 방법을 보여줍니다. DynamoDbItemEncryptor는 DynamoDB 테이블에 항목을 배치하지 않습니다.

DynamoDB Enhanced Client에서 지원되는 모든 키링을 사용할 수 있지만 가능하면 AWS KMS 키링 중 하나를 사용하는 것이 좋습니다.

> **Note**
>
> 하위 수준 DynamoDbItemEncryptor은 검색 가능한 암호화를 지원하지 않습니다. 하위 수준 DynamoDB API와 함께 DynamoDbEncryptionInterceptor을 사용하여 검색 가능한 암호화를 사용할 수 있습니다.

전체 코드 샘플 보기: ItemEncryptDecryptExample.java

1단계: AWS KMS 키링 생성

다음 예제에서는 CreateAwsKmsMrkMultiKeyring를 사용하여 대칭 암호화 KMS AWS KMS 키로 키링을 생성합니다. 이 CreateAwsKmsMrkMultiKeyring 방법을 사용하면 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리할 수 있습니다.

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

2단계: 속성 작업 구성

다음 예제에서는 테이블 항목에 대한 샘플 속성 작업을 나타내는 attributeActionsOnEncrypt 맵을 정의합니다.

> **Note**
>
> 다음 예제에서는 속성을 로 정의하지 않습니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

3단계: 시그니처에서 제외할 속성을 정의합니다.

다음 예제에서는 모든 DO_NOTHING 속성이 고유한 접두사 ":“를 공유한다고 가정하고 이 접두사를 사용하여 허용된 서명되지 않은 속성을 정의합니다. 클라이언트는 접두사가 ":"인 모든 속성 이름이 서명에서 제외된다고 가정합니다. 자세한 내용은 Allowed unsigned attributes 단원을 참조하십시오.

```
final String unsignedAttrPrefix = ":";
```

4단계: **DynamoDbItemEncryptor** 구성 정의

다음 예제에서는 DynamoDbItemEncryptor의 구성을 정의합니다.

이 예제에서는 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB의 암호화 구성 단원을 참조하십시오.

```
final DynamoDbItemEncryptorConfig config = DynamoDbItemEncryptorConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
```

```
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

5단계: **`DynamoDbItemEncryptor`** 생성

다음 예제에서는 4단계의 `config`를 사용하여 새 `DynamoDbItemEncryptor`을 만듭니다.

```
final DynamoDbItemEncryptor itemEncryptor = DynamoDbItemEncryptor.builder()
        .DynamoDbItemEncryptorConfig(config)
        .build();
```

6단계: 테이블 항목을 직접 암호화하고 서명합니다.

다음 예제에서는 `DynamoDbItemEncryptor`를 사용하여 항목을 직접 암호화하고 서명합니다. `DynamoDbItemEncryptor`는 DynamoDB 테이블에 항목을 배치하지 않습니다.

```
final Map<String, AttributeValue> originalItem = new HashMap<>();
originalItem.put("partition_key",
 AttributeValue.builder().s("ItemEncryptDecryptExample").build());
originalItem.put("sort_key", AttributeValue.builder().n("0").build());
originalItem.put("attribute1", AttributeValue.builder().s("encrypt and sign
 me!").build());
originalItem.put("attribute2", AttributeValue.builder().s("sign me!").build());
originalItem.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final Map<String, AttributeValue> encryptedItem = itemEncryptor.EncryptItem(
        EncryptItemInput.builder()
                .plaintextItem(originalItem)
                .build()
).encryptedItem();
```

DynamoDB용 AWS Database Encryption SDK를 사용하도록 기존 DynamoDB 테이블 구성

> 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

버전 3.x를 포함하여 DynamoDB용 Java 클라이언트측 암호화 라이브러리 중 하나를 사용하면 기존 Amazon DynamoDB 테이블을 클라이언트측 암호화를구성할 수 있습니다. 이 주제에서는 채워진 기존

DynamoDB 테이블에 버전 3.x를 추가하기 위해 수행해야 하는 세 가지 단계에 대한 지침을 제공합니다.

사전 조건

DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x에는 AWS SDK for Java 2.x 에서 제공하는 DynamoDB Enhanced Client가 필요합니다. DynamoDBMapper를 계속 사용하는 경우 DynamoDB Enhanced Client를 사용하려면 AWS SDK for Java 2.x 로 마이그레이션해야 합니다.

AWS SDK for Java의 버전 1.x에서 2.x로 마이그레이션하기 지침을 따르세요.

그런 다음 DynamoDB Enhanced Client API를 사용하여 시작하기 지침을 따르세요.

DynamoDB용 Java 클라이언트측 암호화 라이브러리를 사용하도록 테이블을 구성하기 전에 `TableSchema` 주석이 달린 데이터 클래스를 사용을 생성하고 고급 클라이언트를 생성해야 합니다.

1단계: 암호화된 항목 읽기 및 쓰기 준비

다음 단계를 완료하여 AWS Database Encryption SDK 클라이언트가 암호화된 항목을 읽고 쓸 수 있도록 준비합니다. 다음 변경사항을 배포한 후에도 클라이언트는 계속해서 일반 텍스트 항목을 읽고 씁니다. 테이블에 기록된 새 항목을 암호화하거나 서명하지는 않지만 암호화된 항목이 나타나는 즉시 복호화할 수 있습니다. 이러한 변경으로 인해 클라이언트는 새 항목의 암호화를 시작할 수 있습니다. 다음 단계로 진행하기 전에 각 리더에 다음 변경 내용을 배포해야 합니다.

1. 속성 작업 정의

   암호화 및 서명할 속성 값, 서명만 가능한 속성 값, 무시할 속성 값을 정의하는 속성 작업을 포함하도록 주석이 달린 데이터 클래스를 업데이트합니다.

   DynamoDB 향상된 클라이언트 주석에 대한 자세한 지침은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에서 SimpleClass.java를 참조하세요.

   기본적으로 프라이머리 키 속성은 서명되지만 암호화되지는 않으며(`SIGN_ONLY`) 다른 모든 속성은 암호화되고 서명됩니다(`ENCRYPT_AND_SIGN`). 예외를 지정하려면 DynamoDB용 Java 클라이언트측 암호화 라이브러리에 정의된 암호화 주석을 사용합니다. 예를 들어, 특정 속성에 서명되도록 하려면 `@DynamoDbEncryptionSignOnly` 주석만 사용합니다. 특정 속성에 서명하고 암호화 컨텍스트에 포함하려면 `@DynamoDbEncryptionSignAndIncludeInEncryptionContext` 주석을 사용합니다. 특정 속성에 서명되거나 암호화되지 않도록 하려면(`DO_NOTHING`) `@DynamoDbEncryptionDoNothing` 주석을 사용합니다.

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT. 를 정의하는 데 사용되는 주석을 보여주는 예제는 SimpleClass4.java를 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT참조하세요.

주석의 예제는 주석이 달린 데이터 클래스 사용 섹션을 참조하세요.

2. 서명에서 제외할 속성을 정의합니다.

   다음 예제에서는 모든 DO_NOTHING 속성이 고유한 접두사 ":“를 공유한다고 가정하고 이 접두사를 사용하여 허용된 서명되지 않은 속성을 정의합니다. 클라이언트는 접두사가 ":"인 모든 속성 이름은 서명에서 제외된 것으로 간주합니다. 자세한 내용은 Allowed unsigned attributes 단원을 참조하십시오.

```
final String unsignedAttrPrefix = ":";
```

3. 키링 생성

   다음 예제에서는 AWS KMS 키링을 생성합니다. AWS KMS 키링은 대칭 암호화 또는 비대칭 RSA AWS KMS keys 를 사용하여 데이터 키를 생성, 암호화 및 복호화합니다.

   이 예제에서는 CreateMrkMultiKeyring를 사용하여 대칭 암호화 KSM 키를 포함한 AWS KMS 키링을 생성합니다. 이 CreateAwsKmsMrkMultiKeyring 방법을 사용하면 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리할 수 있습니다.

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. DynamoDB 테이블 암호화 구성 정의

   다음 예제는 이 DynamoDB 테이블의 암호화 구성을 나타내는 tableConfigs 맵을 정의합니다.

이 예제에서는 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB의 암호화 구성 단원을 참조하십시오.

`FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`을 일반 텍스트 오버라이드로 지정해야 합니다. 이 정책은 계속해서 일반 텍스트 항목을 읽고 쓰고, 암호화된 항목을 읽고, 클라이언트가 암호화된 항목을 쓸 수 있도록 준비시킵니다.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

5. **`DynamoDbEncryptionInterceptor` 생성**

   다음 예제는 3단계의 `tableConfigs`를 사용하여 `DynamoDbEncryptionInterceptor`을 생성합니다.

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

2단계: 암호화되고 서명된 항목 쓰기

`DynamoDbEncryptionInterceptor` 구성의 일반 텍스트 정책을 업데이트하여 클라이언트가 암호화되고 서명된 항목을 쓸 수 있도록 허용합니다. 다음 변경 사항을 배포하면 클라이언트는 1단계에서 구성한 속성 작업을 기반으로 새 항목을 암호화하고 서명합니다. 클라이언트는 일반 텍스트 항목과 암호화되고 서명된 항목을 읽을 수 있습니다.

3단계로 진행하기 전에 테이블의 기존 일반 텍스트 항목을 모두 암호화하고 서명해야 합니다. 기존 일반 텍스트 항목을 빠르게 암호화하기 위해 실행할 수 있는 단일 지표나 쿼리는 없습니다. 시스템에

가장 적합한 프로세스를 사용하세요. 예를 들어, 테이블을 천천히 스캔한 다음 정의한 속성 작업 및 암호화 구성을 사용하여 항목을 다시 쓰는 비동기 프로세스를 사용할 수 있습니다. 테이블의 일반 텍스트 항목을 식별하려면 암호화되고 서명될 때 AWS Database Encryption SDK가 항목에 추가하는 aws_dbe_head 및 aws_dbe_foot 속성이 포함되지 않은 모든 항목을 스캔하는 것이 좋습니다.

다음 예시에서는 1단계의 테이블 암호화 구성을 업데이트합니다. 일반 텍스트 오버라이드를 FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT로 업데이트해야 합니다. 이 정책은 일반 텍스트 항목을 계속 읽지만 암호화된 항목을 읽고 쓸 수도 있습니다. 업데이트된를 DynamoDbEncryptionInterceptor 사용하여 새를 생성합니다tableConfigs.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

3단계: 암호화되고 서명된 항목만 읽기

모든 항목을 암호화하고 서명한 후에는 DynamoDbEncryptionInterceptor 구성의 일반 텍스트 재정의를 업데이트하여 클라이언트가 암호화되고 서명된 항목만 읽고 쓸 수 있도록 합니다. 다음 변경 사항을 배포하면 클라이언트는 1단계에서 구성한 속성 작업을 기반으로 새 항목을 암호화하고 서명합니다. 클라이언트는 암호화되고 서명된 항목만 읽을 수 있습니다.

다음 예시에서는 2단계의 테이블 암호화 구성을 업데이트합니다.
FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT으로 일반 텍스트 재정의를 업데이트하거나 구성에서 일반 텍스트 정책을 제거할 수 있습니다. 클라이언트는 기본적으로 암호화되고 서명된 항목만 읽고 씁니다. 업데이트된를 DynamoDbEncryptionInterceptor 사용하여 새를 생성합니다tableConfigs.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
```

```
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        // Optional: you can also remove the plaintext policy from your configuration

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x로 마이그레이션

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x는 2.x 코드 베이스를 대대적으로 재작성한 것입니다. 여기에는 새로운 구조화된 데이터 형식, 향상된 멀티테넌시 지원, 원활한 스키마 변경, 검색 가능한 암호화 지원 등 많은 업데이트가 포함되어 있습니다. 이 항목에서는 코드를 버전 3.x으로 마이그레이션하는 방법에 대한 지침을 제공합니다.

버전 1.x에서 2.x로 마이그레이션

버전 3.x로 마이그레이션하기 전에 버전 2.x로 마이그레이션합니다. 버전 2.x에서는 Most Recent Provider의 기호가 `MostRecentProvider`에서 `CachingMostRecentProvider`로 변경되었습니다. 현재 `MostRecentProvider` 기호와 함께 DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 1.x 를 사용하는 경우 코드의 기호 이름을 `CachingMostRecentProvider`으로 업데이트해야 합니다. 자세한 내용은 Most Recent Provider 업데이트를 참조하세요.

버전 2.x에서 3.x로 마이그레이션

다음 절차에서는 코드를 DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 2.x에서 버전 3.x로 마이그레이션하는 방법을 설명합니다.

1단계. 새 형식의 항목을 읽을 준비하기

다음 단계를 완료하여 AWS Database Encryption SDK 클라이언트가 새 형식으로 항목을 읽을 수 있도록 준비합니다. 다음 변경 사항을 배포한 후에도 클라이언트는 버전 2.x에서와 동일한 방식으로 계속 동작합니다. 클라이언트는 계속해서 버전 2.x 형식의 항목을 읽고 쓰지만 이러한 변경 사항을 통해 클라이언트는 새 형식의 항목을 읽을 수 있도록 준비합니다.

를 버전 2.x AWS SDK for Java 로 업데이트

DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x에는 DynamoDB Enhanced Client가 필요합니다. DynamoDB Enhanced Client는 이전 버전에서 사용된 DynamoDBMapper를 대체합니다. 향상된 클라이언트를 사용하려면 AWS SDK for Java 2.x를 사용해야 합니다.

AWS SDK for Java의 버전 1.x에서 2.x로 마이그레이션하기 지침을 따르세요.

필요한 AWS SDK for Java 2.x 모듈에 대한 자세한 내용은 섹션을 참조하세요사전 조건.

레거시 버전으로 암호화된 항목을 읽도록 클라이언트 구성

다음 절차는 아래 코드 예제에 나와 있는 단계의 개요를 제공합니다.

1. 키링을 생성합니다.

   키링 및 암호화 자료 관리자는 이전 버전의 DynamoDB용 Java 클라이언트측 암호화 라이브러리에서 사용하던 암호화 자료 공급자를 대체합니다.

   > **Important**
   >
   > 키링을 생성할 때 지정하는 래핑 키는 버전 2.x에서 암호화 자료 공급자에 사용한 래핑 키와 동일해야 합니다.

2. 주석이 달린 클래스 위에 테이블 스키마를 만듭니다.

   이 단계에서는 새 형식으로 항목을 작성하기 시작할 때 사용할 속성 작업을 정의합니다.

   새로운 DynamoDB Enhanced Client 사용에 대한 지침은 AWS SDK for Java 개발자 안내서의 `TableSchema` 생성을 참조하세요.

   다음 예제에서는 새 속성 작업 주석을 사용하여 버전 2.x에서 주석이 달린 클래스를 업데이트했다고 가정합니다. 속성 작업에 주석을 다는 방법에 대한 자세한 지침은 주석이 달린 데이터 클래스 사용을 참조하세요.

   > **Note**
   >
   > `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. 를 정의하는 데 사용되는 주석을 보여주는 예제는 SimpleClass4.java를 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`참조하세요.

3. 서명에서 제외할 속성을 정의합니다.
4. 버전 2.x 모델 클래스에 구성된 속성 작업의 명시적 맵을 구성합니다.

   이 단계에서는 이전 형식으로 항목을 작성하는 데 사용되는 속성 작업을 정의합니다.
5. DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 2.x에서 사용한 `DynamoDBEncryptor`을 구성합니다.
6. 레거시 동작을 구성합니다.
7. `DynamoDbEncryptionInterceptor`를 만듭니다.
8. 새 AWS SDK DynamoDB 클라이언트를 생성합니다.
9. `DynamoDBEnhancedClient`를 만들고 모델링된 클래스로 테이블을 생성합니다.

   DynamoDB Enhanced Client에 대한 자세한 내용은 향상된 클라이언트 생성을 참조하세요.

```
public class MigrationExampleStep1 {

    public static void MigrationStep1(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Create a Keyring.
        //    This example creates an AWS KMS Keyring that specifies the
        //    same kmsKeyId previously used in the version 2.x configuration.
        //    It uses the 'CreateMrkMultiKeyring' method to create the
        //    keyring, so that the keyring can correctly handle both single
        //    region and Multi-Region KMS Keys.
        //    Note that this example uses the AWS SDK for Java v2 KMS client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        // 2. Create a Table Schema over your annotated class.
        //    For guidance on using the new attribute actions
        //    annotations, see SimpleClass.java in the
        //    aws-database-encryption-sdk-dynamodb GitHub repository.
        //    All primary key attributes must be signed but not encrypted
        //    and by default all non-primary key attributes
        //    are encrypted and signed (ENCRYPT_AND_SIGN).
```

```
        //    If you want a particular non-primary key attribute to be signed but
        //    not encrypted, use the 'DynamoDbEncryptionSignOnly' annotation.
        //    If you want a particular attribute to be neither signed nor encrypted
        //    (DO_NOTHING), use the 'DynamoDbEncryptionDoNothing' annotation.
        final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

        // 3. Define which attributes the client should expect to be excluded
        //    from the signature when reading items.
        //    This value represents all unsigned attributes across the entire
        //    dataset.
        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        // 4. Configure an explicit map of the attribute actions configured
        //    in your version 2.x modeled class.
        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        // 5. Configure the DynamoDBEncryptor that you used in version 2.x.
        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 6. Configure the legacy behavior.
        //    Input the DynamoDBEncryptor and attribute actions created in
        //    the previous steps. For Legacy Policy, use
        //    'FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This policy continues to
read
        //    and write items using the old format, but will be able to read
        //    items written in the new format as soon as they appear.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 7. Create a DynamoDbEncryptionInterceptor with the above configuration.
```

```
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 8. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 7.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 9. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb client
        //    created in Step 8, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

2단계. 새 형식으로 항목 작성

1단계의 변경 사항을 모든 리더에 배포한 후 다음 단계를 완료하여 새 형식으로 항목을 작성하도록 AWS Database Encryption SDK 클라이언트를 구성합니다. 다음 변경 사항을 배포한 후 클라이언트는 이전 형식의 항목을 계속 읽고 새 형식의 항목을 쓰고 읽기 시작합니다.

다음 절차는 아래 코드 예제에 나와 있는 단계의 개요를 제공합니다.

1. 1단계에서 했던 것처럼 키링, 테이블 스키마, 레거시 속성 작업, `allowedUnsignedAttributes` 및 `DynamoDBEncryptor`를 계속 구성합니다.
2. 새 형식을 사용하여 새 항목만 작성하도록 레거시 동작을 업데이트합니다.
3. `DynamoDbEncryptionInterceptor` 생성
4. 새 AWS SDK DynamoDB 클라이언트를 생성합니다.
5. `DynamoDBEnhancedClient`를 만들고 모델링된 클래스로 테이블을 생성합니다.

   DynamoDB Enhanced Client에 대한 자세한 내용은 향상된 클라이언트 생성을 참조하세요.

```
public class MigrationExampleStep2 {

    public static void MigrationStep2(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema, legacy
        //    attribute actions, allowedUnsignedAttributes, and
        //    DynamoDBEncryptor as you did in Step 1.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
 kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);
```

```
        // 2. Update your legacy behavior to only write new items using the new
        //    format.
        //    For Legacy Policy, use 'FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This
policy
        //    continues to read items in both formats, but will only write items
        //    using the new format.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb Client
created
        //    in Step 4, and create a table with your modeled class.
```

```
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

2단계 변경 사항을 배포한 후에는 테이블의 모든 이전 항목을 새 형식으로 다시 암호화해야 3단계로 넘어갈 수 있습니다. 기존 항목을 빠르게 암호화하기 위해 실행할 수 있는 단일 지표나 쿼리는 없습니다. 시스템에 가장 적합한 프로세스를 사용하세요. 예를 들어, 테이블을 천천히 스캔한 다음 정의한 새 속성 작업 및 암호화 구성을 사용하여 항목을 다시 쓰는 비동기 프로세스를 사용할 수 있습니다.

3단계. 새 형식의 항목만 읽고 쓸 수 있습니다

테이블의 모든 항목을 새 형식으로 다시 암호화한 후 구성에서 기존 동작을 제거할 수 있습니다. 클라이언트가 새 형식의 항목만 읽고 쓰도록 구성하려면 다음 단계를 수행합니다.

다음 절차는 아래 코드 예제에 나와 있는 단계의 개요를 제공합니다.

1. 1단계에서 수행한 것처럼 키링, 테이블 스키마, `allowedUnsignedAttributes`을 계속 구성합니다. 구성에서 레거시 속성 작업 및 `DynamoDBEncryptor`을 제거합니다.
2. `DynamoDbEncryptionInterceptor`를 만듭니다.
3. 새 AWS SDK DynamoDB 클라이언트를 생성합니다.
4. `DynamoDBEnhancedClient`를 만들고 모델링된 클래스로 테이블을 생성합니다.

   DynamoDB Enhanced Client에 대한 자세한 내용은 향상된 클라이언트 생성을 참조하세요.

```
public class MigrationExampleStep3 {

    public static void MigrationStep3(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema,
        //    and allowedUnsignedAttributes as you did in Step 1.
        //    Do not include the configurations for the DynamoDBEncryptor or
        //    the legacy attribute actions.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
```

```
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");


        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        //    Do not configure any legacy behavior.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK Client
        //    created in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
```

```
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

.NET

이 주제에서는 DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 버전 3.x를 설치하고 사용하는 방법을 설명합니다. DynamoDB용 AWS Database Encryption SDK를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에 있는 .NET 예제를 참조하세요.

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리는 C# 및 기타 .NET 프로그래밍 언어로 애플리케이션을 작성하는 개발자를 위한 것입니다. 이는 Windows, macOS, Linux에서 지원됩니다.

AWS Database Encryption SDK for DynamoDB의 모든 프로그래밍 언어 구현은 상호 운용 가능합니다. 그러나 SDK for .NET 는 목록 또는 맵 데이터 유형에 빈 값을 지원하지 않습니다. 즉, DynamoDB용 Java 클라이언트 측 암호화 라이브러리를 사용하여 목록 또는 맵 데이터 유형에 대한 빈 값이 포함된 항목을 작성하는 경우 DynamoDB용 .NET 클라이언트 측 암호화 라이브러리를 사용하여 해당 항목을 해독하고 읽을 수 없습니다.

주제

- DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 설치
- .NET을 사용한 디버깅
- DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 사용
- .NET 예제
- DynamoDB용 AWS Database Encryption SDK를 사용하도록 기존 DynamoDB 테이블 구성

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 설치

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리는 NuGet의 AWS.Cryptography.DbEncryptionSDK.DynamoDb 패키지로 사용할 수 있습니다. 라이브러리 설치 및 빌드에 대한 자세한 내용은 aws-database-encryption-sdk-dynamodb 리포지토리의 .NET README.md 파일을 참조하세요. DynamoDB용 .NET 클라이언트 측 암호화 라이브러리에는 AWS Key Management Service (AWS KMS) 키를 사용하지 SDK for .NET 않더라도가 필요합니다. 는 SDK for .NET NuGet 패키지와 함께 설치됩니다.

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 버전 3.x는 .NET 6.0 및 .NET Framework net48 이상을 지원합니다.

.NET을 사용한 디버깅

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리는 로그를 생성하지 않습니다. DynamoDB용 .NET 클라이언트 측 암호화 라이브러리의 예외는 예외 메시지를 생성하지만 스택 트레이스는 생성하지 않습니다.

디버깅에 도움이 되도록 SDK for .NET에서 로그인을 활성화해야 합니다. 의 로그 및 오류 메시지는 에서 발생하는 오류를 DynamoDB용 .NET 클라이언트 측 암호화 라이브러리의 오류 SDK for .NET와 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.)

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 사용

이 주제에서는 DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 버전 3.x의 일부 함수 및 헬퍼 클래스에 대해 설명합니다.

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에서 .NET 예제를 참조하세요.

주제

- 항목 암호화 도구
- AWS Database Encryption SDK for DynamoDB의 속성 작업
- AWS Database Encryption SDK for DynamoDB의 암호화 구성
- AWS Database Encryption SDK로 항목 업데이트

항목 암호화 도구

코어에서 AWS Database Encryption SDK for DynamoDB는 항목 암호화 도구입니다. DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 버전 3.x를 사용하여 다음과 같은 방법으로 DynamoDB 테이블 항목을 암호화, 서명, 확인 및 해독할 수 있습니다.

DynamoDB API용 하위 수준 AWS Database Encryption SDK

테이블 암호화 구성을 사용하여 DynamoDB `PutItem` 요청으로 클라이언트 측 항목을 자동으로 암호화하고 서명하는 DynamoDB 클라이언트를 구성할 수 있습니다. 이 클라이언트를 직접 사용하거나 문서 모델 또는 객체 지속성 모델을 구성할 수 있습니다.

검색 가능한 암호화를 사용하려면 하위 수준 AWS Database Encryption SDK for DynamoDB API를 사용해야 합니다.

하위 수준 **`DynamoDbItemEncryptor`**

하위 수준 `DynamoDbItemEncryptor`에서는 DynamoDB를 호출하지 않고도 테이블 항목을 직접 암호화하고 서명 또는 복호화하고 확인합니다. DynamoDB `PutItem` 또는 `GetItem` 요청을 하지 않습니다. 예를 들어 하위 수준 `DynamoDbItemEncryptor`을 사용하여 이미 검색한 DynamoDB 항목을 직접 복호화하고 확인할 수 있습니다. 하위 수준를 사용하는 경우 DynamoDB와 통신하기 위해에서 SDK for .NET 제공하는 하위 수준 프로그래밍 모델을 사용하는 `DynamoDbItemEncryptor`것이 좋습니다.

하위 수준 `DynamoDbItemEncryptor`은 검색 가능한 암호화를 지원하지 않습니다.

AWS Database Encryption SDK for DynamoDB의 속성 작업

속성 작업은 암호화 및 서명되는 속성 값, 서명만 되는 속성 값, 암호화 컨텍스트에 서명 및 포함되는 속성 값, 무시되는 속성 값을 결정합니다.

.NET 클라이언트로 속성 작업을 지정하려면 객체 모델을 사용하여 속성 작업을 수동으로 정의합니다. 이름-값 페어가 속성 이름과 지정된 작업을 나타내는 `Dictionary` 객체를 생성하여 속성 작업을 지정합니다.

속성을 암호화하고 서명하도록 `ENCRYPT_AND_SIGN`을 지정합니다. 속성을 서명하되 암호화하지 않도록 `SIGN_ONLY`을 지정합니다. 를 지정`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`하여 속성에 서명하고 암호화 컨텍스트에 포함합니다. 서명하지 않으면 속성을 암호화할 수 없습니다. 속성을 무시하도록 `DO_NOTHING`을 지정합니다.

파티션 및 정렬 속성은 `SIGN_ONLY` 또는 중 하나여야 합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. 속성을 로 정의하면 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`파티션 및 정렬 속성도 여야 합니다`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> 속성 작업을 정의한 후에는 서명에서 제외할 속성을 정의해야 합니다. 나중에 서명되지 않은 새 속성을 더 쉽게 추가할 수 있도록 서명되지 않은 속성을 식별할 고유한 접두사(예: ":“)를 선택하는 것이 좋습니다. DynamoDB 스키마와 속성 작업을 정의할 때 DO_NOTHING로 표시된 모든 속성의 속성 이름에 이 접두사를 포함합니다.

다음 객체 모델은 .NET 클라이언트를 사용하여 ENCRYPT_AND_SIGN, SIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, 및 DO_NOTHING 속성 작업을 지정하는 방법을 보여줍니다. 이 예제에서는 접두사 ":"를 사용하여 DO_NOTHING 속성을 식별합니다.

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 암호화 작업을 사용하려면 AWS Database Encryption SDK 버전 3.3 이상을 사용해야 합니다. 를 포함하도록 데이터 모델을 업데이트하기 전에 모든 리더에 새 버전을 배포합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The
 partition attribute must be signed
    ["sort_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The sort
 attribute must be signed
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    ["attribute3"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT,
    [":attribute4"] = CryptoAction.DO_NOTHING
};
```

AWS Database Encryption SDK for DynamoDB의 암호화 구성

AWS Database Encryption SDK를 사용하는 경우 DynamoDB 테이블에 대한 암호화 구성을 명시적으로 정의해야 합니다. 암호화 구성에 필요한 값은 속성 작업을 수동으로 정의했는지 아니면 주석이 달린 데이터 클래스를 사용하여 정의했는지에 따라 달라집니다.

다음 코드 조각은 하위 수준 AWS Database Encryption SDK for DynamoDB API를 사용하여 DynamoDB 테이블 암호화 구성을 정의하고 고유한 접두사로 정의된 서명되지 않은 속성을 허용합니다.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: SearchConfig only required if you use beacons
    Search = new SearchConfig
    {
        WriteVersion = 1, // MUST be 1
        Versions = beaconVersions
    }
};
tableConfigs.Add(ddbTableName, config);
```

논리적 테이블 이름

DynamoDB 테이블의 논리적 테이블 이름.

논리적 테이블 이름은 테이블에 저장된 모든 데이터에 암호로 바인딩되어 DynamoDB 복원 작업을 간소화합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 항상 같은 논리적 테이블 이름을 지정해야 합니다. 복호화이 성공하려면 논리적 테이블 이름이 암호화에 지정된 이름과 일치해야 합니다. 백업에서 DynamoDB 테이블을 복원한 후 DynamoDB 테이블 이름이 변경되더라도 논리적 테이블 이름을 사용하면 복호화 작업에서 테이블을 계속 인식할 수 있습니다.

허용된 서명되지 않은 속성

속성 작업에 `DO_NOTHING`로 표시된 속성.

허용된 무서명 서명에서 제외되는 속성을 클라이언트에게 알려줍니다. 클라이언트는 다른 모든 속성이 서명에 포함되어 있다고 가정합니다. 그런 다음 레코드를 복호화할 때 클라이언트는 확인해야 할 속성과 지정한 허용된 무서명 속성 중에서 무시할 속성을 결정합니다. 허용된 무서명 속성에서는 속성을 제거할 수 없습니다.

모든 DO_NOTHING 속성을 나열하는 배열을 만들어 무서명 허용 속성을 명시적으로 정의할 수 있습니다. DO_NOTHING 속성의 이름을 지정할 때 고유한 접두사를 지정하고 이 접두사를 사용하여 무서명 속성을 클라이언트에게 알릴 수도 있습니다. 고유한 접두사를 지정하는 것이 좋습니다. 이렇게 하면 나중에 새 DO_NOTHING 속성을 추가하는 프로세스가 단순해지기 때문입니다. 자세한 내용은 데이터 모델 업데이트 단원을 참조하십시오.

모든 DO_NOTHING 속성에 접두사를 지정하지 않는 경우 클라이언트가 복호화 시 서명되지 않을 것으로 예상되는 모든 속성을 명시적으로 나열하는 allowedUnsignedAttributes 배열을 구성할 수 있습니다. 반드시 필요한 경우에만 허용된 서명되지 않은 속성을 명시적으로 정의해야 합니다.

검색 구성(선택 사항)

SearchConfig는 비컨 버전을 정의합니다.

검색 가능한 암호화 또는 서명된 비컨을 사용하려면 SearchConfig를 지정해야 합니다.

알고리즘 제품군(선택 사항)

algorithmSuiteId은 AWS Database Encryption SDK가 사용하는 알고리즘 제품군을 정의합니다.

대체 알고리즘 제품군을 명시적으로 지정하지 않는 한 AWS Database Encryption SDK는 기본 알고리즘 제품군을 사용합니다. 기본 알고리즘 제품군은 키 도출, 디지털 서명 및 키 커밋과 함께 AES-GCM 알고리즘을 사용합니다. 기본 알고리즘 제품군이 대부분의 애플리케이션에 적합할 가능성이 높지만 대체 알고리즘 제품군을 선택할 수도 있습니다. 예를 들어, 일부 신뢰 모델은 디지털 서명이 없는 알고리즘 제품군으로 충분할 수 있습니다. AWS Database Encryption SDK가 지원하는 알고리즘 제품군에 대한 자세한 내용은 섹션을 참조하세요AWS Database Encryption SDK에서 지원되는 알고리즘 제품군.

ECDSA 디지털 서명이 없는 AES-GCM 알고리즘 제품군을 선택하려면 테이블 암호화 구성에 다음 코드 조각을 포함합니다.

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

AWS Database Encryption SDK로 항목 업데이트

AWS Database Encryption SDK는 암호화되거나 서명된 속성이 포함된 항목에 대해 ddb:UpdateItem을 지원하지 않습니다. 암호화되거나 서명된 속성을 업데이트하려면 ddb:PutItem을 사용해야 합니다. PutItem 요청에 기존 항목과 동일한 프라이머리 키를 지정하면 새 항목이 기존 항목

을 완전히 대체합니다. 항목을 업데이트한 후 CLOBBER를 사용하여 저장 시 모든 속성을 지우고 바꿀 수도 있습니다.

.NET 예제

다음 예제에서는 DynamoDB용 .NET 클라이언트 측 암호화 라이브러리를 사용하여 애플리케이션의 테이블 항목을 보호하는 방법을 보여줍니다. 더 많은 예제를 찾으려면(자신에게 기여하려면) GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에서 .NET 예제를 참조하세요.

다음 예제에서는 채워지지 않은 새 Amazon DynamoDB 테이블에서 DynamoDB에 대한 .NET 클라이언트 측 암호화 라이브러리를 구성하는 방법을 보여줍니다. 클라이언트측 암호화를 위해 기존 Amazon DynamoDB 테이블을 구성하려면 기존 테이블에 버전 3.x 추가 섹션을 참조하세요.

주제

- DynamoDB API용 하위 수준 AWS 데이터베이스 암호화 SDK 사용
- 하위 수준 사용 DynamoDbItemEncryptor

DynamoDB API용 하위 수준 AWS 데이터베이스 암호화 SDK 사용

다음 예제에서는 하위 수준 AWS Database Encryption SDK for DynamoDB API를 AWS KMS 키링과 함께 사용하여 DynamoDB `PutItem` 요청으로 클라이언트 측 항목을 자동으로 암호화하고 서명하는 방법을 보여줍니다.

지원되는 모든 키링을 사용할 수 있지만 가능하면 AWS KMS 키링 중 하나를 사용하는 것이 좋습니다.

전체 코드 샘플: BasicPutGetExample.cs 참조

1단계: AWS KMS 키링 생성

다음 예제에서는 `CreateAwsKmsMrkMultiKeyring`를 사용하여 대칭 암호화 KMS AWS KMS 키로 키링을 생성합니다. 이 `CreateAwsKmsMrkMultiKeyring` 방법을 사용하면 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리할 수 있습니다.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

2단계: 속성 작업 구성

다음 예제에서는 테이블 항목에 대한 샘플 속성 작업을 나타내는 `attributeActionsOnEncrypt` 사전을 정의합니다.

> **Note**
>
> 다음 예제에서는 속성을 로 정의하지 않습니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT. SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

3단계: 시그니처에서 제외할 속성을 정의합니다.

다음 예제에서는 모든 DO_NOTHING 속성이 고유한 접두사 ":“를 공유한다고 가정하고 이 접두사를 사용하여 허용된 서명되지 않은 속성을 정의합니다. 클라이언트는 접두사가 ":"인 모든 속성 이름이 서명에서 제외된다고 가정합니다. 자세한 내용은 Allowed unsigned attributes 단원을 참조하십시오.

```
const String unsignAttrPrefix = ":";
```

4단계: DynamoDB 테이블 암호화 구성 정의

다음 예제는 이 DynamoDB 테이블의 암호화 구성을 나타내는 tableConfigs 맵을 정의합니다.

이 예제에서는 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB의 암호화 구성 단원을 참조하십시오.

> **Note**
>
> 검색 가능한 암호화 또는 서명된 비컨을 사용하려면 암호화 구성에도 SearchConfig을 포함해야 합니다.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
tableConfigs.Add(ddbTableName, config);
```

5단계: 새 AWS SDK DynamoDB 클라이언트 생성

다음 예제에서는 4단계`TableEncryptionConfigs`의를 사용하여 새 AWS SDK DynamoDB 클라이언트를 생성합니다.

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

6단계: DynamoDB 테이블 항목 암호화 및 서명

다음 예제에서는 샘플 테이블 항목을 나타내는 `item` 사전을 정의하고 DynamoDB 테이블에 해당 항목을 넣습니다. 항목은 DynamoDB로 전송되기 전에 클라이언트측에서 암호화되고 서명됩니다.

```
var item = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("BasicPutGetExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

PutItemRequest putRequest = new PutItemRequest
{
    TableName = ddbTableName,
    Item = item
};

PutItemResponse putResponse = await ddb.PutItemAsync(putRequest);
```

하위 수준 사용 **DynamoDbItemEncryptor**

다음 예제는 AWS KMS 키링이 있는 하위 수준 DynamoDbItemEncryptor을 사용하여 테이블 항목을 직접 암호화하고 서명하는 방법을 보여줍니다. DynamoDbItemEncryptor는 DynamoDB 테이블에 항목을 배치하지 않습니다.

DynamoDB Enhanced Client에서 지원되는 모든 키링을 사용할 수 있지만 가능하면 AWS KMS 키링 중 하나를 사용하는 것이 좋습니다.

> **Note**
>
> 하위 수준 DynamoDbItemEncryptor은 검색 가능한 암호화를 지원하지 않습니다. 하위 수준 AWS Database Encryption SDK for DynamoDB API를 사용하여 검색 가능한 암호화를 사용합니다.

전체 코드 샘플: ItemEncryptDecryptExample.cs 참조

1단계: AWS KMS 키링 생성

다음 예제에서는 CreateAwsKmsMrkMultiKeyring를 사용하여 대칭 암호화 KMS AWS KMS 키로 키링을 생성합니다. 이 CreateAwsKmsMrkMultiKeyring 방법을 사용하면 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리할 수 있습니다.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

2단계: 속성 작업 구성

다음 예제에서는 테이블 항목에 대한 샘플 속성 작업을 나타내는 attributeActionsOnEncrypt 사전을 정의합니다.

> **Note**
>
> 다음 예제에서는 속성을 로 정의하지 않습니
> 다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

```
var attributeActionsOnEncrypt = new Dictionary<String, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

3단계: 시그니처에서 제외할 속성을 정의합니다.

다음 예제에서는 모든 DO_NOTHING 속성이 고유한 접두사 ":“를 공유한다고 가정하고 이 접두사를 사용하여 허용된 서명되지 않은 속성을 정의합니다. 클라이언트는 접두사가 ":"인 모든 속성 이름이 서명에서 제외된다고 가정합니다. 자세한 내용은 Allowed unsigned attributes 단원을 참조하십시오.

```
String unsignAttrPrefix = ":";
```

4단계: **DynamoDbItemEncryptor** 구성 정의

다음 예제에서는 DynamoDbItemEncryptor의 구성을 정의합니다.

이 예제에서는 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB의 암호화 구성 단원을 참조하십시오.

```
var config = new DynamoDbItemEncryptorConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
```

5단계: **DynamoDbItemEncryptor** 생성

다음 예제에서는 4단계의 config를 사용하여 새 DynamoDbItemEncryptor을 만듭니다.

```
var itemEncryptor = new DynamoDbItemEncryptor(config);
```

6단계: 테이블 항목을 직접 암호화하고 서명합니다.

다음 예제에서는 DynamoDbItemEncryptor를 사용하여 항목을 직접 암호화하고 서명합니다. DynamoDbItemEncryptor는 DynamoDB 테이블에 항목을 배치하지 않습니다.

```
var originalItem = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("ItemEncryptDecryptExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

var encryptedItem = itemEncryptor.EncryptItem(
    new EncryptItemInput { PlaintextItem = originalItem }
).EncryptedItem;
```

DynamoDB용 AWS Database Encryption SDK를 사용하도록 기존 DynamoDB 테이블 구성

DynamoDB용 .NET 클라이언트 측 암호화 라이브러리 버전 3.x를 사용하면 클라이언트 측 암호화를 위해 기존 Amazon DynamoDB 테이블을 구성할 수 있습니다. 이 주제에서는 채워진 기존 DynamoDB 테이블에 버전 3.x를 추가하기 위해 수행해야 하는 세 가지 단계에 대한 지침을 제공합니다.

1단계: 암호화된 항목 읽기 및 쓰기 준비

다음 단계를 완료하여 AWS Database Encryption SDK 클라이언트가 암호화된 항목을 읽고 쓸 수 있도록 준비합니다. 다음 변경사항을 배포한 후에도 클라이언트는 계속해서 일반 텍스트 항목을 읽고 씁니다. 테이블에 기록된 새 항목을 암호화하거나 서명하지는 않지만 암호화된 항목이 나타나는 즉시 복호화할 수 있습니다. 이러한 변경으로 인해 클라이언트는 새 항목의 암호화를 시작할 수 있습니다. 다음 단계로 진행하기 전에 각 리더에 다음 변경 내용을 배포해야 합니다.

1. 속성 작업 정의

   객체 모델을 생성하여 암호화 및 서명할 속성 값, 서명만 할 속성 값, 무시할 속성 값을 정의합니다.

기본적으로 프라이머리 키 속성은 서명되지만 암호화되지는 않으며(SIGN_ONLY) 다른 모든 속성은 암호화되고 서명됩니다(ENCRYPT_AND_SIGN).

속성을 암호화하고 서명하도록 ENCRYPT_AND_SIGN을 지정합니다. 속성을 서명하되 암호화하지 않도록 SIGN_ONLY을 지정합니다. SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT를 지정하여 서명 및 속성을 지정하고 암호화 컨텍스트에 포함합니다. 서명하지 않으면 속성을 암호화할 수 없습니다. 속성을 무시하도록 DO_NOTHING을 지정합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB의 속성 작업 단원을 참조하십시오.

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 속성을 지정하는 경우 파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

2. 서명에서 제외할 속성을 정의합니다.

다음 예제에서는 모든 DO_NOTHING 속성이 고유한 접두사 ":“를 공유한다고 가정하고 이 접두사를 사용하여 허용된 서명되지 않은 속성을 정의합니다. 클라이언트는 접두사가 ":"인 모든 속성 이름은 서명에서 제외된 것으로 간주합니다. 자세한 내용은 Allowed unsigned attributes 단원을 참조하십시오.

```
const String unsignAttrPrefix = ":";
```

3. 키링 생성

다음 예제에서는 AWS KMS 키링을 생성합니다. AWS KMS 키링은 대칭 암호화 또는 비대칭 RSA AWS KMS keys 를 사용하여 데이터 키를 생성, 암호화 및 복호화합니다.

이 예제에서는 `CreateMrkMultiKeyring`를 사용하여 대칭 암호화 KSM 키를 포함한 AWS KMS 키링을 생성합니다. 이 `CreateAwsKmsMrkMultiKeyring` 방법을 사용하면 키링이 단일 리전 키와 다중 리전 키를 모두 올바르게 처리할 수 있습니다.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. DynamoDB 테이블 암호화 구성 정의

다음 예제는 이 DynamoDB 테이블의 암호화 구성을 나타내는 `tableConfigs` 맵을 정의합니다.

이 예제에서는 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다.

`FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`을 일반 텍스트 오버라이드로 지정해야 합니다. 이 정책은 계속해서 일반 텍스트 항목을 읽고 쓰고, 암호화된 항목을 읽고, 클라이언트가 암호화된 항목을 쓸 수 있도록 준비시킵니다.

테이블 암호화 구성에 포함된 값에 대한 자세한 내용은 섹션을 참조하세요AWS Database Encryption SDK for DynamoDB의 암호화 구성.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

5. 새 AWS SDK DynamoDB 클라이언트 생성

다음 예제에서는 4단계`TableEncryptionConfigs`의를 사용하여 새 AWS SDK DynamoDB 클라이언트를 생성합니다.

```
var ddb = new Client.DynamoDbClient(
```

```
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

2단계: 암호화되고 서명된 항목 쓰기

클라이언트가 암호화되고 서명된 항목을 작성할 수 있도록 테이블 암호화 구성에서 일반 텍스트 정책을 업데이트합니다. 다음 변경 사항을 배포하면 클라이언트는 1단계에서 구성한 속성 작업을 기반으로 새 항목을 암호화하고 서명합니다. 클라이언트는 일반 텍스트 항목과 암호화되고 서명된 항목을 읽을 수 있습니다.

3단계로 진행하기 전에 테이블의 기존 일반 텍스트 항목을 모두 암호화하고 서명해야 합니다. 기존 일반 텍스트 항목을 빠르게 암호화하기 위해 실행할 수 있는 단일 지표나 쿼리는 없습니다. 시스템에 가장 적합한 프로세스를 사용하세요. 예를 들어, 테이블을 천천히 스캔한 다음 정의한 속성 작업 및 암호화 구성을 사용하여 항목을 다시 쓰는 비동기 프로세스를 사용할 수 있습니다. 테이블의 일반 텍스트 항목을 식별하려면 암호화되고 서명될 때 AWS Database Encryption SDK가 항목에 추가하는 `aws_dbe_head` 및 `aws_dbe_foot` 속성이 포함되지 않은 모든 항목을 스캔하는 것이 좋습니다.

다음 예시에서는 1단계의 테이블 암호화 구성을 업데이트합니다. 일반 텍스트 오버라이드를 `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`로 업데이트해야 합니다. 이 정책은 일반 텍스트 항목을 계속 읽지만 암호화된 항목을 읽고 쓸 수도 있습니다. 업데이트된를 사용하여 새 AWS SDK DynamoDB 클라이언트를 생성합니다`TableEncryptionConfigs`.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

3단계: 암호화되고 서명된 항목만 읽기

모든 항목을 암호화하고 서명한 후에는 클라이언트가 암호화되고 서명된 항목만 읽고 쓸 수 있도록 테이블 암호화 구성에서 일반 텍스트 재정의를 업데이트합니다. 다음 변경 사항을 배포하면 클라이언트

는 1단계에서 구성한 속성 작업을 기반으로 새 항목을 암호화하고 서명합니다. 클라이언트는 암호화되고 서명된 항목만 읽을 수 있습니다.

다음 예시에서는 2단계의 테이블 암호화 구성을 업데이트합니다. `FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT`으로 일반 텍스트 재정의를 업데이트하거나 구성에서 일반 텍스트 정책을 제거할 수 있습니다. 클라이언트는 기본적으로 암호화되고 서명된 항목만 읽고 씁니다. 업데이트된를 사용하여 새 AWS SDK DynamoDB 클라이언트를 생성합니다`TableEncryptionConfigs`.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: you can also remove the plaintext policy from your configuration
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

Rust

이 주제에서는 DynamoDB용 Rust 클라이언트 측 암호화 라이브러리 버전 1.x를 설치하고 사용하는 방법을 설명합니다. DynamoDB용 AWS Database Encryption SDK를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에 있는 Rust 예제를 참조하세요.

AWS Database Encryption SDK for DynamoDB의 모든 프로그래밍 언어 구현은 상호 운용 가능합니다.

주제

- 사전 조건
- 설치
- DynamoDB용 Rust 클라이언트 측 암호화 라이브러리 사용

사전 조건

DynamoDB용 Rust 클라이언트 측 암호화 라이브러리를 설치하기 전에 다음 사전 요구 사항이 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

DynamoDB용 Rust 클라이언트 측 암호화 라이브러리는 Crates.io aws-db-esdk 크레이트로 사용할 수 있습니다. 라이브러리 설치 및 빌드에 대한 자세한 내용은 aws-database-encryption-sdk-dynamodb GitHub 리포지토리의 README.md 파일을 참조하세요.

직접

DynamoDB용 Rust 클라이언트 측 암호화 라이브러리를 설치하려면 aws-database-encryption-sdk-dynamodb GitHub 리포지토리를 복제하거나 다운로드합니다.

최신 버전 설치

프로젝트 디렉터리에서 다음 Cargo 명령을 실행합니다.

```
cargo add aws-db-esdk
```

또는 Cargo.toml에 다음 줄을 추가합니다.

```
aws-db-esdk = "<version>"
```

DynamoDB용 Rust 클라이언트 측 암호화 라이브러리 사용

이 주제에서는 DynamoDB용 Rust 클라이언트 측 암호화 라이브러리 버전 1.x의 일부 함수 및 헬퍼 클래스에 대해 설명합니다.

DynamoDB용 Rust 클라이언트 측 암호화 라이브러리를 사용한 프로그래밍에 대한 자세한 내용은 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에 있는 Rust 예제를 참조하세요.

주제

항목 암호화 도구

코어에서 AWS Database Encryption SDK for DynamoDB는 항목 암호화 도구입니다. DynamoDB용 Rust 클라이언트 측 암호화 라이브러리 버전 1.x를 사용하여 다음과 같은 방법으로 DynamoDB 테이블 항목을 암호화, 서명, 확인 및 해독할 수 있습니다.

DynamoDB API용 하위 수준 AWS 데이터베이스 암호화 SDK

테이블 암호화 구성을 사용하여 DynamoDB `PutItem` 요청으로 클라이언트 측 항목을 자동으로 암호화하고 서명하는 DynamoDB 클라이언트를 구성할 수 있습니다.

검색 가능한 암호화를 사용하려면 DynamoDB API용 하위 수준 AWS Database Encryption SDK를 사용해야 합니다.

DynamoDB API용 하위 수준 AWS Database Encryption SDK를 사용하는 방법을 보여주는 예제는 GitHub의 aws-database-encryption-sdk-dynamodb 리포지토리에서 basic_get_put_example.rs를 참조하세요.

하위 수준 **`DynamoDbItemEncryptor`**

하위 수준 `DynamoDbItemEncryptor`에서는 DynamoDB를 호출하지 않고도 테이블 항목을 직접 암호화하고 서명 또는 복호화하고 확인합니다. DynamoDB `PutItem` 또는 `GetItem` 요청을 하지 않습니다. 예를 들어 하위 수준 `DynamoDbItemEncryptor`을 사용하여 이미 검색한 DynamoDB 항목을 직접 복호화하고 확인할 수 있습니다.

하위 수준 `DynamoDbItemEncryptor`은 검색 가능한 암호화를 지원하지 않습니다.

하위 수준를 사용하는 방법을 보여주는 예제는 GitHub의 aws-database-encryption-sdk-dynamodbencryption-sdk-dynamodb 리포지토리에서 item_encrypt_decrypt.rs를 `DynamoDbItemEncryptor`참조하세요.

AWS Database Encryption SDK for DynamoDB의 속성 작업

속성 작업은 암호화 및 서명되는 속성 값, 서명만 되는 속성 값, 암호화 컨텍스트에 서명 및 포함되는 속성 값, 무시되는 속성 값을 결정합니다.

Rust 클라이언트를 사용하여 속성 작업을 지정하려면 객체 모델을 사용하여 속성 작업을 수동으로 정의합니다. 이름-값 페어가 속성 이름과 지정된 작업을 나타내는 HashMap 객체를 생성하여 속성 작업을 지정합니다.

속성을 암호화하고 서명하도록 ENCRYPT_AND_SIGN을 지정합니다. 속성을 서명하되 암호화하지 않도록 SIGN_ONLY을 지정합니다. 를 지정SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT하여 속성에 서명하고 암호화 컨텍스트에 포함합니다. 서명하지 않으면 속성을 암호화할 수 없습니다. 속성을 무시하도록 DO_NOTHING을 지정합니다.

파티션 및 정렬 속성은 SIGN_ONLY 또는 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT. 속성을 로 정의하는 경우 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT파티션 및 정렬 속성도 여야 합니다SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

> **Note**
>
> 속성 작업을 정의한 후에는 서명에서 제외할 속성을 정의해야 합니다. 나중에 서명되지 않은 새 속성을 더 쉽게 추가할 수 있도록 서명되지 않은 속성을 식별할 고유한 접두사(예: ":“)를 선택하는 것이 좋습니다. DynamoDB 스키마와 속성 작업을 정의할 때 DO_NOTHING로 표시된 모든 속성의 속성 이름에 이 접두사를 포함합니다.

다음 객체 모델은 Rust 클라이언트를 사용하여 ENCRYPT_AND_SIGN, SIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, 및 DO_NOTHING 속성 작업을 지정하는 방법을 보여줍니다. 이 예제에서는 접두사 ":"를 사용하여 DO_NOTHING 속성을 식별합니다.

```
let attribute_actions_on_encrypt = HashMap::from([
    ("partition_key".to_string(), CryptoAction::SignOnly),
    ("sort_key".to_string(), CryptoAction::SignOnly),
    ("attribute1".to_string(), CryptoAction::EncryptAndSign),
    ("attribute2".to_string(), CryptoAction::SignOnly),
    (":attribute3".to_string(), CryptoAction::DoNothing),
]);
```

AWS Database Encryption SDK for DynamoDB의 암호화 구성

AWS Database Encryption SDK를 사용하는 경우 DynamoDB 테이블에 대한 암호화 구성을 명시적으로 정의해야 합니다. 암호화 구성에 필요한 값은 속성 작업을 수동으로 정의했는지 아니면 주석이 달린 데이터 클래스를 사용하여 정의했는지에 따라 달라집니다.

다음 코드 조각은 하위 수준 AWS Database Encryption SDK for DynamoDB API를 사용하여 DynamoDB 테이블 암호화 구성을 정의하고 고유한 접두사로 정의된 서명되지 않은 속성을 허용합니다.

```
let table_config = DynamoDbTableEncryptionConfig::builder()
    .logical_table_name(ddb_table_name)
    .partition_key_name("partition_key")
    .sort_key_name("sort_key")
    .attribute_actions_on_encrypt(attribute_actions_on_encrypt)
    .keyring(kms_keyring)
    .allowed_unsigned_attribute_prefix(UNSIGNED_ATTR_PREFIX)
    // Specifying an algorithm suite is optional
    .algorithm_suite_id(
        DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
    )
    .build()?;

let table_configs = DynamoDbTablesEncryptionConfig::builder()
    .table_encryption_configs(HashMap::from([(ddb_table_name.to_string(),
 table_config)]))
    .build()?;
```

논리적 테이블 이름

DynamoDB 테이블의 논리적 테이블 이름.

논리적 테이블 이름은 테이블에 저장된 모든 데이터에 암호로 바인딩되어 DynamoDB 복원 작업을 간소화합니다. 암호화 구성을 처음 정의할 때 DynamoDB 테이블 이름을 논리적 테이블 이름으로 지정하는 것이 좋습니다. 항상 같은 논리적 테이블 이름을 지정해야 합니다. 복호화이 성공하려면 논리적 테이블 이름이 암호화에 지정된 이름과 일치해야 합니다. 백업에서 DynamoDB 테이블을 복원한 후 DynamoDB 테이블 이름이 변경되더라도 논리적 테이블 이름을 사용하면 복호화 작업에서 테이블을 계속 인식할 수 있습니다.

허용된 서명되지 않은 속성

속성 작업에 `DO_NOTHING`로 표시된 속성.

허용된 무서명 서명에서 제외되는 속성을 클라이언트에게 알려줍니다. 클라이언트는 다른 모든 속성이 서명에 포함되어 있다고 가정합니다. 그런 다음 레코드를 복호화할 때 클라이언트는 확인해야 할 속성과 지정한 허용된 무서명 속성 중에서 무시할 속성을 결정합니다. 허용된 무서명 속성에서는 속성을 제거할 수 없습니다.

모든 DO_NOTHING 속성을 나열하는 배열을 만들어 무서명 허용 속성을 명시적으로 정의할 수 있습니다. DO_NOTHING 속성의 이름을 지정할 때 고유한 접두사를 지정하고 이 접두사를 사용하여 무서명 속성을 클라이언트에게 알릴 수도 있습니다. 고유한 접두사를 지정하는 것이 좋습니다. 이렇게 하면 나중에 새 DO_NOTHING 속성을 추가하는 프로세스가 단순해지기 때문입니다. 자세한 내용은 데이터 모델 업데이트 단원을 참조하십시오.

모든 DO_NOTHING 속성에 접두사를 지정하지 않는 경우 클라이언트가 복호화 시 서명되지 않을 것으로 예상되는 모든 속성을 명시적으로 나열하는 allowedUnsignedAttributes 배열을 구성할 수 있습니다. 반드시 필요한 경우에만 허용된 서명되지 않은 속성을 명시적으로 정의해야 합니다.

검색 구성(선택 사항)

SearchConfig는 비컨 버전을 정의합니다.

검색 가능한 암호화 또는 서명된 비컨을 사용하려면 SearchConfig를 지정해야 합니다.

알고리즘 제품군(선택 사항)

algorithmSuiteId은 AWS Database Encryption SDK가 사용하는 알고리즘 제품군을 정의합니다.

대체 알고리즘 제품군을 명시적으로 지정하지 않는 한 AWS Database Encryption SDK는 기본 알고리즘 제품군을 사용합니다. 기본 알고리즘 제품군은 키 도출, 디지털 서명 및 키 커밋과 함께 AES-GCM 알고리즘을 사용합니다. 기본 알고리즘 제품군이 대부분의 애플리케이션에 적합할 가능성이 높지만 대체 알고리즘 제품군을 선택할 수도 있습니다. 예를 들어, 일부 신뢰 모델은 디지털 서명이 없는 알고리즘 제품군으로 충분할 수 있습니다. AWS Database Encryption SDK가 지원하는 알고리즘 제품군에 대한 자세한 내용은 섹션을 참조하세요AWS Database Encryption SDK에서 지원되는 알고리즘 제품군.

ECDSA 디지털 서명이 없는 AES-GCM 알고리즘 제품군을 선택하려면 테이블 암호화 구성에 다음 코드 조각을 포함합니다.

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

AWS Database Encryption SDK로 항목 업데이트

AWS Database Encryption SDK는 암호화되거나 서명된 속성이 포함된 항목에 대해 ddb:UpdateItem을 지원하지 않습니다. 암호화되거나 서명된 속성을 업데이트하려면 ddb:PutItem을 사

용해야 합니다. `PutItem` 요청에 기존 항목과 동일한 프라이머리 키를 지정하면 새 항목이 기존 항목을 완전히 대체합니다.

레거시 DynamoDB Encryption Client

2023년 6월 9일에 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. AWS Database Encryption SDK는 레거시 DynamoDB Encryption Client 버전을 계속 지원합니다. 이름 변경과 함께 변경된 클라이언트측 암호화 라이브러리의 여러 부분에 대한 자세한 내용은 Amazon DynamoDB Encryption Client 이름 변경 섹션을 참조하세요.

DynamoDB용 Java 클라이언트측 암호화 라이브러리의 최신 버전으로 마이그레이션하려면 버전 3.x로 마이그레이션 섹션을 참조하세요.

주제

- AWS DynamoDB용 Database Encryption SDK 버전 지원
- DynamoDB Encryption Client의 작동 방식
- Amazon DynamoDB Encryption Client 개념
- 암호화 자료 공급자
- Amazon DynamoDB Encryption Client에서 사용할 수 있는 프로그래밍 언어
- 데이터 모델 변경
- DynamoDB Encryption Client 애플리케이션의 문제 해결

AWS DynamoDB용 Database Encryption SDK 버전 지원

레거시 장의 주제에서는 DynamoDB Encryption Client for Java 버전 1.x~2.x와 DynamoDB Encryption Client for Python 버전 1.x~3.x 에 대한 정보를 제공합니다.

다음 표에는 Amazon DynamoDB에서 클라이언트측 암호화를 지원하는 언어 및 버전이 나와 있습니다.

프로그래밍 언어	버전	SDK 메이저 버전 수명 주기 단계
Java	버전 1.x	지원 종료 단계, 2022년 7월부터 적용

프로그래밍 언어	버전	SDK 메이저 버전 수명 주기 단계
Java	버전 2.x	일반 가용성(GA)
Java	버전 3.x	일반 가용성(GA)
Python	버전 1.x	지원 종료 단계, 2022년 7월부터 적용
Python	버전 2.x	지원 종료 단계, 2022년 7월부터 적용
Python	버전 3.x	일반 가용성(GA)

DynamoDB Encryption Client의 작동 방식

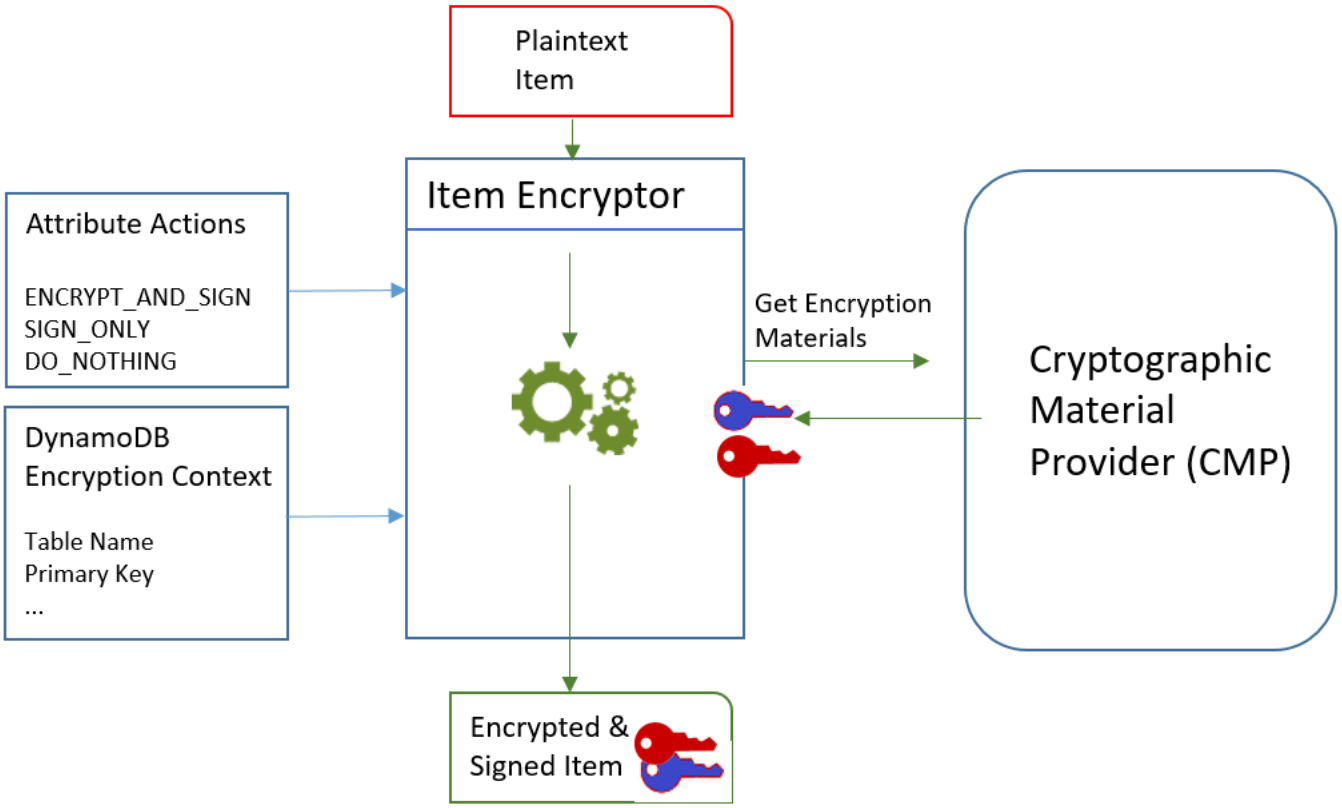
> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

DynamoDB Encryption Client는 DynamoDB에 저장하는 데이터를 보호하도록 특별히 설계되었습니다. 라이브러리에는 변경 없이 확장하거나 사용할 수 있는 보안 구현이 포함되어 있습니다. 그리고 대부분의 요소는 추상 요소로 표현되므로 호환되는 사용자 지정 구성 요소를 생성하고 사용할 수 있습니다.

테이블 항목 암호화 및 서명

DynamoDB Encryption Client의 핵심에는 테이블 항목을 암호화, 서명, 확인 및 복호화하는 항목 암호화 도구가 있습니다. 항목 암호화 도구는 테이블 항목에 대한 정보와 암호화 및 서명할 항목에 대한 지침을 사용합니다. 또한 암호화 자료와 이 자료를 사용하는 방법에 대한 지침을 사용자가 선택 및 구성하는 암호화 자료 공급자에서 가져옵니다.

다음 다이어그램은 이 프로세스에 대한 개략적인 보기를 보여줍니다.



테이블 항목을 암호화하고 서명하려면 DynamoDB Encryption Client에 다음이 필요합니다.

- 테이블에 대한 정보입니다. 사용자가 제공하는 DynamoDB 암호화 컨텍스트에서 테이블에 대한 정보를 가져옵니다. 일부 도우미는 DynamoDB에서 필요한 정보를 가져오고 DynamoDB 암호화 컨텍스트를 생성합니다.

  > **Note**
  >
  > DynamoDB Encryption Client의 DynamoDB 암호화 컨텍스트는 AWS Key Management Service (AWS KMS) 및의 암호화 컨텍스트와 관련이 없습니다 AWS Encryption SDK. DynamoDB

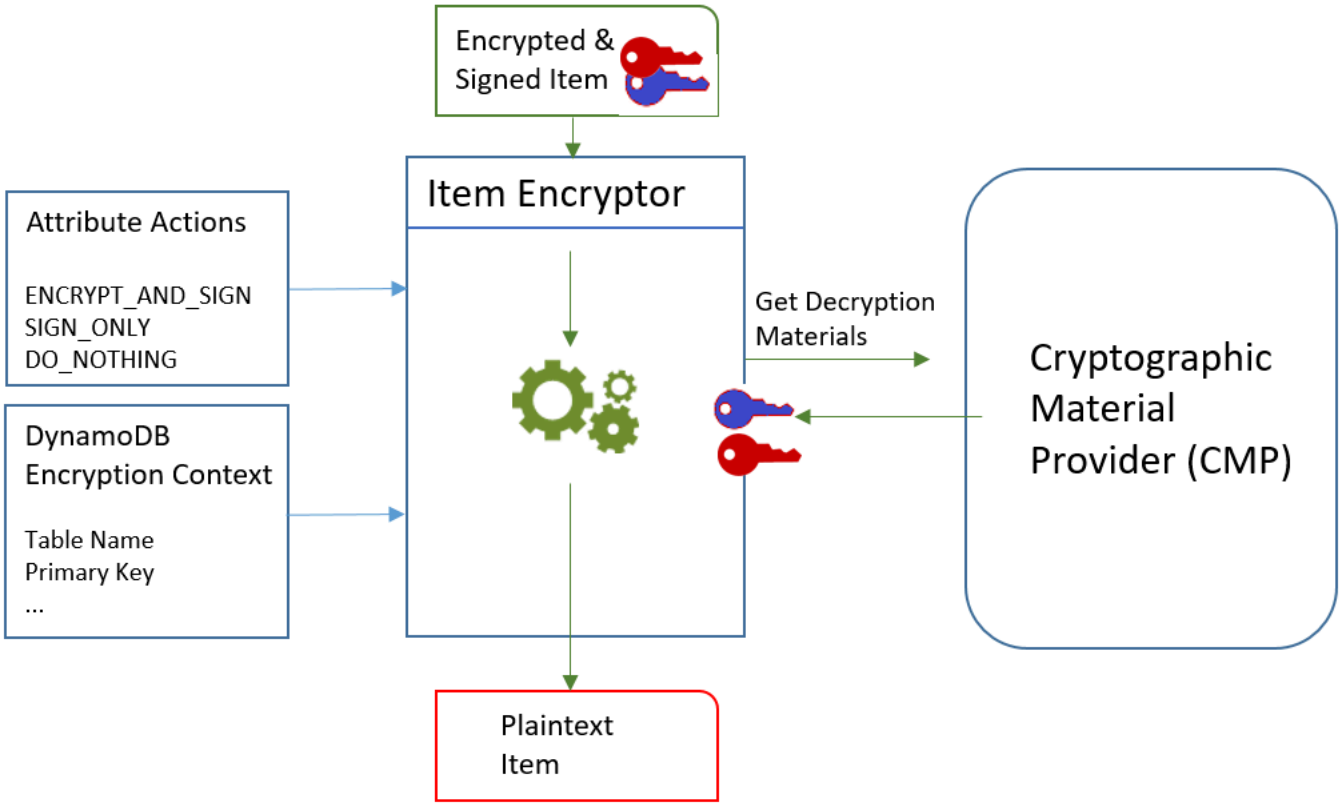
- 암호화 및 서명할 속성. 제공하는 속성 작업에서 이 정보를 가져옵니다.
- 암호화 및 서명 키를 포함하는 암호화 자료. 이러한 자료는 선택 및 구성하는 암호화 자료 공급자(CMP)에서 가져옵니다.
- 항목 암호화 및 서명 지침. CMP는 암호화 및 서명 알고리즘을 비롯하여 암호화 자료 사용 관련 지침을 실제 자료 설명에 추가합니다.

항목 암호화 도구는 이러한 요소를 모두 사용하여 항목을 암호화하고 서명합니다. 또한 항목 암호화 도구는 암호화 및 서명 지침(실제 자료 설명)을 포함하는 자료 설명 속성과, 서명을 포함하는 속성, 이 두 가지 속성을 항목에 추가합니다. 항목 암호화 도구와 직접 상호 작용하거나 항목 암호화 도구와 상호 작용하는 도우미 기능을 사용하여 안전한 기본 동작을 구현할 수 있습니다.

결과는 암호화되고 서명된 데이터를 포함하는 DynamoDB 항목입니다.

테이블 항목 확인 및 복호화

다음 다이어그램에 나와 있듯이 이러한 구성 요소는 함께 작동하여 항목을 확인하고 복호화합니다.



항목을 확인하고 복호화하려면 DynamoDB Encryption Client에 다음과 같이 동일한 구성 요소, 동일한 구성의 구성 요소 또는 항목 복호화을 위해 특별히 설계된 구성 요소가 필요합니다.

- DynamoDB 암호화 컨텍스트의 테이블에 대한 정보입니다.
- 확인하고 복호화할 속성. 이러한 정보는 속성 작업에서 가져옵니다.
- 확인 및 복호화 키를 포함하는 복호화 자료. 이러한 정보는 사용자가 선택 및 구성하는 암호화 자료 공급자(CMP)에서 가져옵니다.

암호화된 항목에는 이를 암호화하는 데 사용된 CMP 레코드가 포함되어 있지 않습니다. 동일한 CMP, 동일한 구성의 CMP 또는 항목을 복호화하도록 설계된 CMP를 제공해야 합니다.

- 항목의 암호화 및 서명된 방식에 대한 정보(암호화 및 서명 알고리즘 포함). 클라이언트는 항목의 자료 설명 속성에서 이러한 정보를 가져옵니다.

항목 암호화 도구는 이러한 요소를 모두 사용하여 항목을 확인 및 복호화합니다. 또한 자료 설명 및 서명 속성도 제거됩니다. 결과는 일반 텍스트 DynamoDB 항목입니다.

Amazon DynamoDB Encryption Client 개념

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

이 주제에서는 Amazon DynamoDB Encryption Client에서 사용되는 개념과 용어에 대해 설명합니다.

DynamoDB Encryption Client의 구성 요소가 어떻게 상호 작용하는지에 대해 알아보려면 DynamoDB Encryption Client의 작동 방식 섹션을 참조하세요.

주제

- 암호화 자료 공급자(CMP)
- 항목 암호화 도구
- 속성 작업
- 자료 설명
- DynamoDB 암호화 컨텍스트
- 공급자 스토어

암호화 자료 공급자(CMP)

DynamoDB Encryption Client를 구현할 때 가장 먼저 해야 할 작업 중 하나는 암호화 자료 공급자(CMP)(암호화 자료 공급자라고도 함)를 선택하는 것입니다. 사용자의 선택에 따라 나머지 구현의 상당 부분이 결정됩니다.

암호화 자료 공급자(CMP)는 항목 암호화 도구에서 테이블 항목을 암호화 및 서명하는 데 사용하는 암호화 자료를 수집, 조합 및 반환합니다. CMP는 사용할 암호화 알고리즘과 암호화 및 서명 키를 생성하고 보호하는 방법을 결정합니다.

CMP는 항목 암호화와 상호 작용합니다. 항목 암호화 도구는 CMP에 암호화 또는 복호화 자료를 요청하고, CMP는 이를 항목 암호화 도구에 반환합니다. 그러면 항목 암호화 도구는 암호화 자료를 사용하여 항목을 암호화하고 서명하거나 확인 및 복호화합니다.

CMP는 클라이언트를 구성할 때 지정합니다. 호환되는 사용자 지정 CMP를 만들거나 라이브러리에 있는 여러 CMP 중 하나를 사용할 수 있습니다. 대부분의 CMP는 여러 프로그래밍 언어에 사용할 수 있습니다.

항목 암호화 도구

항목 암호화 도구는 DynamoDB Encryption Client에 대한 암호화 작업을 수행하는 하위 수준 구성 요소입니다. 이 도구는 암호화 자료 공급자(CMP)로부터 암호화 자료를 요청한 다음, CMP에서 반환하는 자료를 사용하여 테이블 항목을 암호화 및 서명하거나 확인 및 복호화합니다.

항목 암호화 도구와 직접 상호 작용하거나 라이브러리에서 제공하는 헬퍼를 사용할 수 있습니다. 예를 들면 DynamoDB Encryption Client for Java에는 `DynamoDBEncryptor` 항목 암호화 도구와 직접 상호 작용하지 않고 `DynamoDBMapper`과 함께 사용할 수 있는 `AttributeEncryptor` 헬퍼 클래스가 포함되어 있습니다. Python 라이브러리에는 항목 암호기와 상호 작용하는 `EncryptedTable`, `EncryptedClient`, 및 `EncryptedResource` 헬퍼 클래스가 포함되어 있습니다.

속성 작업

속성 작업은 항목의 각 속성에 대해 수행할 작업을 항목 암호화 도구에 알려줍니다.

속성 작업 값은 다음 중 하나일 수 있습니다.

- 암호화 및 서명 - 속성 값을 암호화합니다. 항목 서명에 속성(이름 및 값)을 포함합니다.
- 서명만 - 항목 서명에 속성을 포함합니다.
- 아무 작업 안 함 - 속성을 암호화거나 서명하지 않습니다.

중요한 데이터를 저장할 수 있는 속성의 경우 Encrypt and sign(암호화 및 서명)을 사용합니다. 기본 키 속성(파티션 키 및 정렬 키)의 경우 Sign only(서명만)를 사용합니다. 자료 설명 속성 및 서명 속성은 서명되거나 암호화되지 않습니다. 이러한 속성에 대해 속성 작업을 지정할 필요가 없습니다.

속성 작업을 신중하게 선택합니다. 확실하지 않은 경우 Encrypt and sign(암호화 및 서명)을 사용합니다. DynamoDB Encryption Client를 사용하여 테이블 항목을 보호한 후 속성에 대한 작업을 변경하면 서명 유효성 검사 오류가 발생할 위험이 있습니다. 자세한 내용은 데이터 모델 변경을 참조하세요.

> **Warning**
>
> 기본 키 속성은 암호화하지 마십시오. 일반 텍스트로 남겨 두어야 DynamoDB에서 전체 테이블 스캔을 실행하지 않고 해당 항목을 찾을 수 있습니다.

DynamoDB 암호화 컨텍스트에서 프라이머리 키 속성이 식별되는 경우 이러한 속성을 암호화하려 할 때 클라이언트에서 오류가 발생합니다.

속성 작업을 지정하는 데 사용하는 기법은 프로그래밍 언어마다 다릅니다. 사용하는 도우미 클래스에만 해당될 수도 있습니다.

자세한 내용은 프로그래밍 언어 설명서를 참조하세요.

- Python
- Java

자료 설명

암호화된 테이블 항목에 대한 자료 설명은 테이블 항목의 암호화 및 서명 방법에 대한 정보(예: 암호화 알고리즘)로 구성됩니다. 암호화 자료 공급자(CMP)는 암호화 및 서명을 위해 암호화 자료를 결합할 때 자료 설명을 기록합니다. 나중에 항목을 확인하고 복호화하기 위해 암호화 자료를 조립해야 할 때 자료 설명을 가이드로 사용합니다.

DynamoDB Encryption Client에서 자료 설명은 다음과 같은 세 가지 관련 요소를 참조하세요.

요청한 자료 설명

특정 암호화 자료 공급자(CMP)에서는 암호화 알고리즘 같은 고급 옵션을 지정할 수 있습니다. 선택 사항을 지정하려면 테이블 항목을 암호화하기 위해 요청의 DynamoDB 암호화 컨텍스트 자료 설명 속성에 이름-값 페어를 추가합니다. 이 요소를 요청한 자료 설명이라고 합니다. 요청된 자료 설명의 유효한 값은 선택한 CMP에 의해 정의됩니다.

> **Note**
>
> 자료 설명은 보안 기본값보다 우선할 수 있으므로 요청된 자료 설명을 사용해야 하는 설득력 있는 이유가 없는 한 생략하는 것이 좋습니다.

실제 자료 설명

암호화 자료 공급자(CMP)에서 반환하는 자료 설명을 실제 자료 설명이라고 합니다. 여기에는 CMP가 암호화 자료를 조합할 때 사용한 실제 값이 설명되어 있습니다. 일반적으로 요청된 자료 설명(있는 경우)과 추가 및 변경 내용으로 구성됩니다.

자료 설명 속성

클라이언트는 암호화된 항목의 자료 설명 속성에 실제 자료 설명을 저장합니다. 자료 설명 속성 이름은 `amzn-ddb-map-desc`이고 속성 값은 실제 자료 설명입니다. 클라이언트는 자료 설명 속성의 값을 사용하여 항목을 확인하고 복호화합니다.

DynamoDB 암호화 컨텍스트

DynamoDB 암호화 컨텍스트는 테이블 및 항목에 대한 정보를 암호화 자료 공급자(CMP)에게 제공합니다. 고급 구현에서는 DynamoDB 암호화 컨텍스트에 요청한 자료 설명이 포함될 수 있습니다.

테이블 항목을 암호화하는 경우 DynamoDB 암호화 컨텍스트는 암호화된 속성 값에 암호로 바인딩됩니다. 복호화할 때 DynamoDB 암호화 컨텍스트가 암호화에 사용된 DynamoDB 암호화 컨텍스트와 대/소문자가 정확히 일치하지 않으면 복호화 작업이 실패합니다. 항목 암호화 도구와 직접 상호 작용하는 경우 암호화 또는 복호화 메서드를 호출할 때 DynamoDB 암호화 컨텍스트를 제공해야 합니다. 대부분의 헬퍼는 DynamoDB 암호화 컨텍스트를 자동으로 생성합니다.

> **Note**
>
> DynamoDB Encryption Client의 DynamoDB 암호화 컨텍스트는 AWS Key Management Service (AWS KMS) 및의 암호화 컨텍스트와 관련이 없습니다 AWS Encryption SDK. DynamoDB

DynamoDB 암호화 컨텍스트에는 다음 필드가 포함될 수 있습니다. 모든 필드와 값은 선택 사항입니다.

- 테이블 이름

- 파티션 키 이름
- 정렬 키 이름
- 속성 이름-값 페어
- 요청한 자료 설명

공급자 스토어

공급자 스토어는 암호화 자료 공급자(CMP)를 반환하는 구성 요소입니다. 공급자 스토어는 CMP를 생성하거나 다른 공급자 스토어와 같은 다른 소스에서 가져올 수 있습니다. 공급자 스토어는 생성한 CMP 버전을 영구 스토어에 저장합니다. 영구 스토어에는 저장된 각 CMP가 요청자의 자료 이름 및 버전 번호로 식별됩니다.

DynamoDB Encryption Client의 Most Recent Provider는 공급자 스토어에서 CMP를 가져오지만, 공급자 스토어를 사용하여 모든 구성 요소에 CMP를 제공할 수 있습니다. 각 Most Recent Provider는 공급자 스토어 하나와 연결되지만, 공급자 스토어 하나는 여러 요청자의 여러 공급자로 CMP를 제공할 수 있습니다.

공급자 스토어는 요청 시 새 버전의 CMP를 생성하고 새 버전과 기존 버전을 반환합니다. 또한 해당 자료얼 이름의 최신 버전 번호도 반환합니다. 이를 통해 요청자는 공급자 스토어에 요청할 수 있는 새 버전의 CMP가 있을 때 이를 알 수 있습니다.

DynamoDB Encryption Client에는 DynamoDB에 저장되고 내부 DynamoDB Encryption Client를 사용하여 암호화된 키로 래핑된 CMP를 생성하는 공급자 스토어인 MetaStore가 포함되어 있습니다.

자세히 알아보기:

- 공급자 스토어: Java, Python
- MetaStore: Java, Python

암호화 자료 공급자

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

DynamoDB Encryption Client를 사용할 때 내려야 하는 가장 중요한 결정 중 하나는 암호화 자료 공급자(CMP)를 선택하는 것입니다. CMP는 암호화 자료를 조합하여 항목 암호화 도구에 반환합니다. 또한 암호화 및 서명 키의 생성 방법, 각 항목에 대해 새로운 키 자료를 생성하거나 재사용할지 여부, 사용되는 암호화 및 서명 알고리즘도 결정합니다.

DynamoDB Encryption Client 라이브러리에 제공된 구현에서 CMP를 선택하거나 호환 가능한 사용자 지정 CMP를 구축할 수 있습니다. CMP 선택은 사용하는 프로그래밍 언어에 따라 달라질 수도 있습니다.

이 주제에서는 가장 일반적인 CMP를 설명하고 애플리케이션에 가장 적합한 CMP를 선택하는 데 도움이 되는 몇 가지 조언을 제공합니다.

Direct KMS Materials Provider

Direct KMS Materials Provider는 AWS Key Management Service(AWS KMS)를 암호화되지 않은 상태로 두는 AWS KMS key에 따라 테이블 항목을 보호합니다. 애플리케이션에서 암호화 자료를 생성하거나 관리할 필요가 없습니다. AWS KMS key 를 사용하여 각 항목에 대한 고유한 암호화 및 서명 키를 생성하기 때문에이 공급자는 항목을 암호화하거나 해독할 AWS KMS 때마다를 호출합니다.

AWS KMS 를 사용하고 트랜잭션당 하나의 AWS KMS 호출이 애플리케이션에 유용한 경우이 공급자가 좋은 선택입니다.

자세한 내용은 Direct KMS Materials Provider을 참조하세요.

래핑된 자료 공급자(래핑된 CMP)

래핑된 자료 공급자(래핑된 CMP)는 DynamoDB Encryption Client 외부에서 래핑 및 서명 키를 생성 및 관리할 수 있도록 해줍니다.

래핑된 CMP는 각 항목에 대해 고유한 암호화 키를 생성합니다. 그런 다음 사용자가 제공한 래핑(또는 언래핑) 및 서명 키를 사용합니다. 따라서 래핑 및 서명 키의 생성 방법과 각 항목에 고유한지 또는 재사용되는지를 결정합니다. 래핑된 CMP는 암호화 자료를 사용하지 않고 안전하게 관리할 AWS KMS 수 있는 애플리케이션을 위한 Direct KMS Provider의 안전한 대안입니다.

자세한 내용은 Wrapped Materials Provider을 참조하세요.

Most Recent Provider

Most Recent Provider는 공급자 스토어와 함께 작동하도록 설계된 암호화 자료 공급자(CMP)입니다. 공급자 스토어에서 CMP를 가져오고 CMP에서 반환한 암호화 자료를 가져옵니다. Most Recent Provider는 일반적으로 각각의 CMP를 사용하여 암호화 자료에 대한 여러 요청을 충족하지만, 공

급자 스토어의 기능을 사용하여 자료의 재사용 범위를 제어하고, CMP 교체 빈도를 결정하며, Most Recent Provider를 변경하지 않고 사용되는 CMP 유형도 변경합니다.

호환되는 모든 공급자 스토어에서 Most Recent Provider를 사용할 수 있습니다. DynamoDB Encryption Client에는 래핑된 CMP를 반환하는 공급자 스토어인 MetaStore가 포함됩니다.

Most Recent Provider는 관련 암호화 소스에 대한 호출을 최소화해야 하는 애플리케이션과, 보안 요구 사항을 위반하지 않으면서 일부 암호화 자료를 재사용할 수 있는 애플리케이션에 적합합니다. 예를 들어 항목을 암호화하거나 해독할 AWS KMS 때마다를 호출하지 않고 AWS Key Management Service (AWS KMS)의 AWS KMS key에서 암호화 자료를 보호할 수 있습니다.

자세한 내용은 Most Recent Provider을 참조하세요.

Static Materials Provider

Static Materials Provider는 테스트, 개념 증명 데모 및 레거시 호환성 목적으로 설계되었습니다. 각 항목에 대해 고유한 암호화 자료를 생성하지는 않습니다. 입력한 것과 동일한 암호화 및 서명 키를 반환하며, 이러한 키는 테이블 항목을 암호화, 복호화 및 서명하는 데 직접 사용됩니다.

> **Note**
>
> Java 라이브러리의 Asymmetric Static Provider는 Static Provider가 아닙니다. 단순히 Wrapped CMP에 대한 대체 생성자를 제공할 뿐입니다. 프로덕션 환경에서는 안전하지만 가능하면 래핑된 CMP를 직접 사용해야 합니다.

주제

- Direct KMS Materials Provider
- Wrapped Materials Provider
- Most Recent Provider
- Static Materials Provider

Direct KMS Materials Provider

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용

DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

Direct KMS Materials Provider(Direct KMS Provider)는 AWS Key Management Service (AWS KMS)을 암호화되지 않은 상태로 유지하지 않는 AWS KMS key에 따라 테이블 항목을 보호합니다. 이 암호화 자료 공급자는 각 테이블 항목에 대해 고유한 암호화 키 및 서명 키를 반환합니다. 이를 위해 항목을 암호화하거나 해독할 AWS KMS 때마다가 호출됩니다.

DynamoDB 항목을 높은 빈도로 대규모로 처리하는 경우 AWS KMS requests-per-second 수 제한을 초과하여 처리가 지연될 수 있습니다. 제한을 초과해야 하는 경우 AWS Support 센터에서 사례를 생성합니다. Most Recent Provider와 같이 키 재사용이 제한된 암호화 자료 공급자를 사용하는 것도 고려해 볼 수 있습니다.

Direct KMS Provider를 사용하려면 호출자에게 에서 AWS 계정 GenerateDataKey AWS KMS key및 Decrypt 작업을 호출할 수 있는 하나 이상의 및 권한이 있어야 합니다 AWS KMS key. AWS KMS key는 대칭 암호화 키여야 합니다. DynamoDB Encryption Client는 비대칭 암호화를 지원하지 않습니다. DynamoDB 글로벌 테이블을 사용하는 경우 AWS KMS 다중 리전 키를 지정하는 것이 좋습니다. 자세한 내용은 사용 방법을 참조하세요.

> **Note**
>
> Direct KMS Provider를 사용하면 기본 키 속성의 이름과 값이 AWS KMS 암호화 컨텍스트 및 관련 AWS KMS 작업 AWS CloudTrail 로그에 일반 텍스트로 표시됩니다. 그러나 DynamoDB Encryption Client는 암호화된 어떤 속성 값도 일반 텍스트로 노출하지 않습니다.

Direct KMS Provider는 DynamoDB Encryption Client가 지원하는 여러 암호화 자료 공급자(CMP) 중 하나입니다. 기타 CMP에 대한 자세한 내용은 암호화 자료 공급자 섹션을 참조하세요.

예제 코드는 다음을 참조하십시오.

- Java: AwsKmsEncryptedItem
- Python: aws-kms-encrypted-table, aws-kms-encrypted-item

주제

- 사용 방법
- 작동 방식

사용 방법

Direct KMS Provider를 생성하려면 키 ID 파라미터를 사용하여 계정에 대칭 암호화 KMS 키를 지정합니다. 키 ID 파라미터의 값은 AWS KMS key의 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN일 수 있습니다. 키 식별자에 대한 자세한 내용은 AWS Key Management Service 개발자 가이드의 키 식별자를 참조하세요.

Direct KMS Provider에는 대칭 암호화 KMS 키가 필요합니다. 비대칭 KMS 키를 사용할 수 없습니다. 그러나 다중 리전 KMS 키, 가져온 키 자료가 있는 KMS 키 또는 사용자 지정 키 스토어의 KMS 키를 사용할 수 있습니다. KMS 키에 대한 kms:GenerateDataKey 및 kms:Decrypt 권한이 있어야 합니다. 따라서 관리형 또는 AWS 소유 KMS 키가 아닌 고객 AWS 관리형 키를 사용해야 합니다.

DynamoDB Encryption Client for Python은 키 ID 파라미터 값이 포함된 경우 해당 리전 AWS KMS 에서 호출할 리전을 결정합니다. 그렇지 않으면 AWS KMS 클라이언트에서 리전을 지정하거나에서 구성하는 리전을 사용합니다 AWS SDK for Python (Boto3). Python의 리전 선택에 대한 자세한 내용은 Python용 AWS SDK(Boto3) API 참조의 구성을 참조하세요.

지정한 클라이언트에 리전이 포함된 경우 Java용 DynamoDB 암호화 클라이언트는 클라이언트의 리전 AWS KMS AWS KMS 에서 호출할 리전을 결정합니다. 그렇지 않으면 AWS SDK for Java에서 구성한 리전을 사용합니다. 의 리전 선택에 대한 자세한 내용은 AWS SDK for Java 개발자 안내서의 AWS 리전 선택을 AWS SDK for Java참조하세요.

Java

```
// Replace the example key ARN and Region with valid values for your application
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Python

다음 예에서는 ARN 키를 사용하여 AWS KMS key를 지정합니다. 키 식별자에이 포함되지 않은 경우 AWS 리전 DynamoDB Encryption Client는 구성된 Botocore 세션에서 리전을 가져오거나 Boto 기본값에서 리전을 가져옵니다.

```
# Replace the example key ID with a valid value
kms_key = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
```

```
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key)
```

Amazon DynamoDB 글로벌 테이블을 사용하는 경우 AWS KMS 다중 리전 키로 데이터를 암호화하는 것이 좋습니다. 다중 리전 키는 키 ID와 키 구성 요소가 동일하기 때문에 서로 교환하여 사용할 수 AWS 리전 있는 다른 AWS KMS keys 에 있습니다. 자세한 내용을 알아보려면 AWS Key Management Service 개발자 안내서의 다중 리전 키 사용을 참조하세요.

> **Note**
>
> 글로벌 테이블 버전 2017.11.29를 사용하는 경우 예약된 복제 필드가 암호화되거나 서명되지 않도록 속성 작업을 설정해야 합니다. 자세한 내용은 이전 버전 글로벌 테이블 관련 문제을 참조하세요.

DynamoDB Encryption Client에서 다중 리전 키를 사용하려면 다중 리전 키를 생성하여 애플리케이션이 실행되는 리전에 복제합니다. 그런 다음 DynamoDB Encryption Client가 AWS KMS을 호출하는 리전의 다중 리전 키를 사용하도록 Direct KMS Provider를 구성합니다.

다음 예제에서는 다중 리전 키를 사용하여 미국 동부(버지니아 북부)(us-east-1) 리전의 데이터를 암호화하고 미국 서부(오레곤)(us-west-2) 리전에서 이를 복호화하도록 DynamoDB Encryption Client를 구성합니다.

Java

이 예제에서 DynamoDB Encryption Client는 AWS KMS 클라이언트의 리전 AWS KMS 에서 호출할 리전을 가져옵니다. 이 `keyArn` 값은 동일한 리전의 다중 리전 키를 식별합니다.

```
// Encrypt in us-east-1

// Replace the example key ARN and Region with valid values for your application
final String usEastKey = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-east-1'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usEastKey);
```

```
// Decrypt in us-west-2
```

```
// Replace the example key ARN and Region with valid values for your application
final String usWestKey = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usWestKey);
```

Python

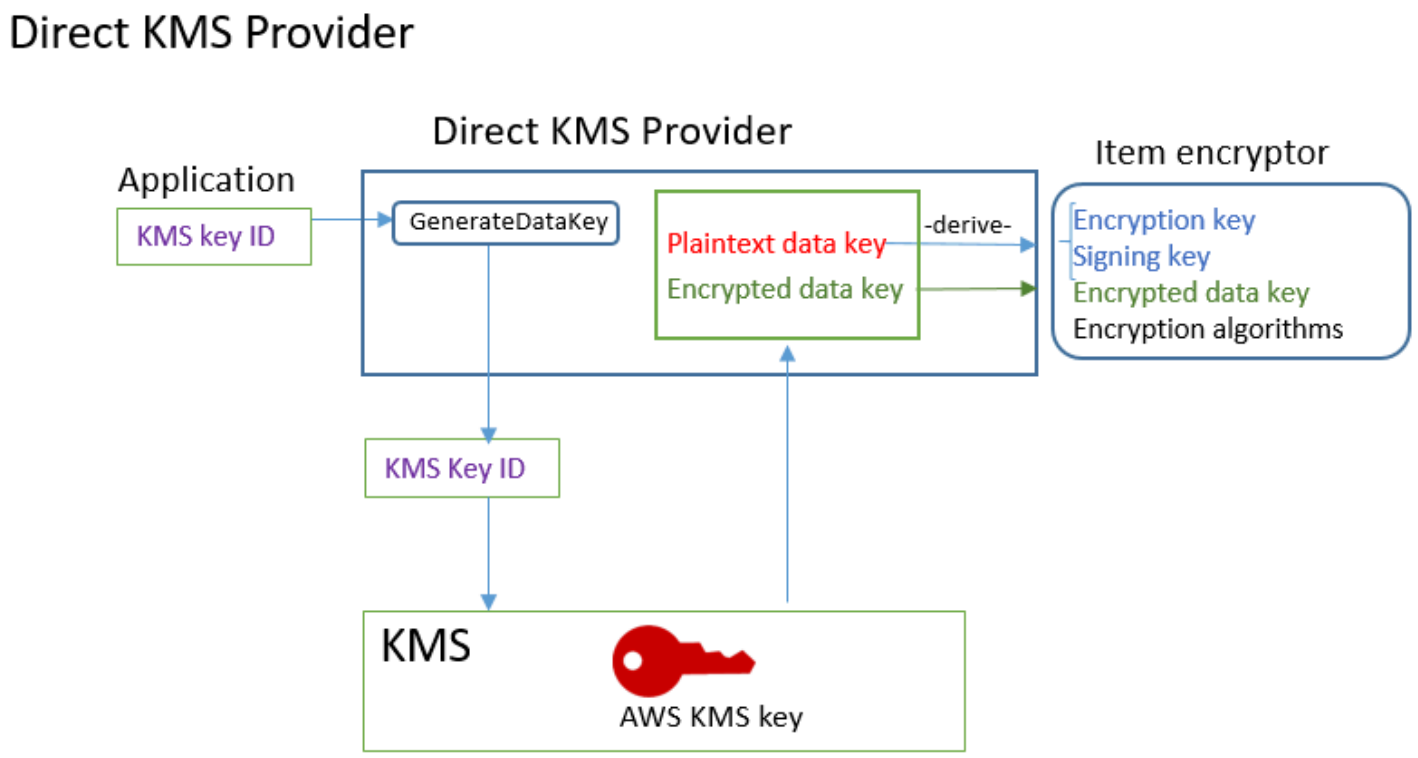
이 예제에서 DynamoDB Encryption Client는 키 ARN의 리전 AWS KMS 에서 호출할 리전을 가져옵니다.

```
# Encrypt in us-east-1

# Replace the example key ID with a valid value
us_east_key = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_east_key)
```

```
# Decrypt in us-west-2

# Replace the example key ID with a valid value
us_west_key = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_west_key)
```

작동 방식

Direct KMS Provider는 다음 다이어그램에서처럼 사용자가 지정하는 AWS KMS key 에 의해 보호되는 암호화 및 서명 키를 반환합니다.



- 암호화 자료를 생성하기 위해 Direct KMS Provider는 AWS KMS key 사용자가 지정한 를 사용하여 각 항목에 대해 고유한 데이터 키를 생성 AWS KMS 하도록 요청합니다. 이 공급자는 데이터 키의 일반 텍스트 복사본에서 항목의 암호화 및 서명 키를 추출한 다음 항목의 자료 설명 속성에 저장된 암호화된 데이터 키와 함께 암호화 및 서명 키를 반환합니다.

  항목 암호화기는 암호화 및 서명 키를 사용하여 가능한 한 빨리 메모리에서 암호화 및 서명 키를 제거합니다. 파생된 데이터 키의 암호화된 복사본만 암호화된 항목에 저장됩니다.

- 복호화 자료를 생성하기 위해 Direct KMS Provider는 암호화된 데이터 키를 복호화 AWS KMS 하도록 요청합니다. 그런 다음 일반 텍스트 데이터 키에서 확인 및 서명 키를 추출하여 항목 암호화 도구에 반환합니다.

  항목 암호화기는 항목을 확인하고 확인에 성공하면 암호화된 값을 복호화합니다. 그런 다음 가능한 빨리 메모리에서 키를 제거합니다.

암호화 자료 가져오기

이 단원에서는 항목 암호화 도구에서 암호화 자료 요청 수신 시 Direct KMS Provider의 입력, 출력 및 처리에 대해 자세히 설명합니다.

입력(애플리케이션에서)

- 의 키 ID입니다 AWS KMS key.

입력(항목 암호화 도구에서)

- DynamoDB 암호화 컨텍스트

출력(항목 암호화 도구로)

- 암호화 키(일반 텍스트)
- 서명 키
- 실제 자료 설명: 이러한 값은 클라이언트가 항목에 추가하는 자료 설명 속성에 저장됩니다.
  - amzn-ddb-env-key: 로 암호화된 Base64-encoded 데이터 키 AWS KMS key
  - amzn-ddb-env-alg: 암호화 알고리즘, 기본값은 AES/256
  - amzn-ddb-sig-alg: 서명 알고리즘, 기본값은 HmacSHA256/256
  - amzn-ddb-wrap-alg: kms

처리

1. 다이렉트 KMS 공급자는 지정된 AWS KMS 를 사용하여 항목에 대한 고유한 데이터 키를 AWS KMS key 생성하라는 요청을 보냅니다. https://docs.aws.amazon.com/kms/latest/APIReference/API_GenerateDataKey.html 이 작업은 일반 텍스트 키와 AWS KMS key로 암호화된 복사본을 반환합니다. 이 자료를 초기 키 자료라고 합니다.

   이 요청은 AWS KMS 암호화 컨텍스트에 다음 값을 일반 텍스트로 포함합니다. 이러한 비밀이 아닌 값은 암호화된 개체에 암호화 방식으로 바인딩되므로 복호화 시 동일한 암호화 컨텍스트가 필요합니다. 이러한 값을 사용하여 AWS CloudTrail 로그에서에 대한 호출을 식별할 수 AWS KMS 있습니다.

   - amzn-ddb-env-alg – 암호화 알고리즘, 기본값은 AES/256
   - amzn-ddb-sig-alg – 서명 알고리즘, 기본값은 HmacSHA256/256
   - (선택 사항) aws-kms-table – *### ##*
   - (선택 사항) *### # ##* – *### # #*(이진 값은 Base64로 인코딩됨)
   - (선택 사항) *## # ##* – *## # #*(이진 값은 Base64로 인코딩됨)

Direct KMS Provider는 항목의 DynamoDB AWS KMS 암호화 컨텍스트에서 암호화 컨텍스트 값을 가져옵니다. DynamoDB DynamoDB 암호화 컨텍스트에 테이블 이름과 같은 값이 포함되지 않은 경우 해당 이름-값 페어는 AWS KMS 암호화 컨텍스트에서 생략됩니다.

2. Direct KMS Provider는 데이터 키에서 대칭 암호화 키 및 서명 키를 추출합니다. 기본적으로 Secure Hash Algorithm(SHA) 256 및 RFC5869 HMAC 기반 키 추출 함수를 사용하여 256비트 AES 대칭 암호화 키 및 256비트 HMAC-SHA-256 서명 키를 추출합니다.
3. Direct KMS Provider는 출력을 항목 암호화 도구로 반환합니다.
4. 항목 암호화기는 암호화 키를 사용하여 지정된 속성을 암호화하고 서명 키를 사용하여 실제 자료 설명에 지정된 알고리즘을 사용하여 서명합니다. 가능한 한 빨리 메모리에서 일반 텍스트 키를 제거합니다.

복호화 자료 가져오기

이 단원에서는 항목 암호화 도구에서 복호화 자료 요청 수신 시 Direct KMS Provider의 입력, 출력 및 처리를 자세히 설명합니다.

입력(애플리케이션에서)

- 의 키 ID입니다 AWS KMS key.

  키 ID의 값은 AWS KMS key의 키 ID, 키 ARN, 별칭 이름 또는 별칭 ARN일 수 있습니다. 리전과 같이 키 ID에 포함되지 않은 모든 값은 AWS 명명된 프로필에서 사용할 수 있어야 합니다. 키 ARN은 에 AWS KMS 서 필요로 하는 모든 값을 제공합니다.

입력(항목 암호화 도구에서)

- 자료 설명 속성의 내용을 포함하는 DynamoDB 암호화 컨텍스트의 복사본입니다.

출력(항목 암호화 도구로)

- 암호화 키(일반 텍스트)
- 서명 키

처리

1. Direct KMS Provider는 암호화된 항목의 자료 설명 속성에서 암호화된 데이터 키를 가져옵니다.

2. 지정된를 사용하여 암호화된 데이터 키를 AWS KMS key 해독 AWS KMS 하도록 요청합니다. 이 작업은 일반 텍스트 키를 반환합니다.

   이 요청은 데이터 키를 생성 및 암호화하는 데 사용된 것과 동일한 AWS KMS 암호화 컨텍스트를 사용해야 합니다.

   - aws-kms-table – ### ##
   - ### # ## – ### # #(이진 값은 Base64로 인코딩됨)
   - (선택 사항) ## # ## – ## # #(이진 값은 Base64로 인코딩됨)
   - amzn-ddb-env-alg – 암호화 알고리즘, 기본값은 AES/256
   - amzn-ddb-sig-alg – 서명 알고리즘, 기본값은 HmacSHA256/256
3. Direct KMS Provider는 Secure Hash Algorithm(SHA) 256 및 RFC5869 HMAC 기반 키 파생 기능을 사용하여 데이터 키에서 256비트 AES 대칭 암호화 키와 256비트 HMAC-SHA-256 서명 키를 파생합니다.
4. Direct KMS Provider는 출력을 항목 암호화 도구로 반환합니다.
5. 항목 암호화기는 서명 키를 사용하여 항목을 확인합니다. 성공하면 대칭 암호화 키를 사용하여 암호화된 속성 값을 복호화합니다. 이러한 작업은 실제 자료 설명에 지정된 암호화 및 서명 알고리즘을 사용합니다. 항목 암호화 도구는 가능한 한 빨리 메모리에서 일반 텍스트 키를 제거합니다.

Wrapped Materials Provider

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

래핑된 자료 공급자(래핑된 CMP)를 사용하면 DynamoDB Encryption Client를 통해 모든 소스의 래핑 및 서명 키를 사용할 수 있습니다. 래핑된 CMP는 AWS 서비스에 종속되지 않습니다. 그러나 항목을 확인하고 복호화하기 위해 올바른 키를 제공하는 것을 포함하여 클라이언트 외부에서 래핑 및 서명 키를 생성하고 관리해야 합니다.

래핑된 CMP는 각 항목에 대해 고유한 항목 암호화 키를 생성합니다. 사용자가 제공하는 래핑 키와 항목 암호화 키를 래핑하여 래핑된 항목 암호화 키를 항목의 자료 설명 속성에 저장합니다. 래핑 및 서명

키를 제공하므로 래핑 및 서명 키가 생성되는 방법과 해당 키가 각 항목에 고유한지 또는 재사용되는지 여부를 결정합니다.

래핑된 CMP는 안전한 구현이며 암호화 자료를 관리할 수 있는 애플리케이션에 적합한 선택입니다.

래핑된 CMP는 DynamoDB Encryption Client가 지원하는 여러 암호화 자료 공급자(CMP) 중 하나입니다. 기타 CMP에 대한 자세한 내용은 암호화 자료 공급자 섹션을 참조하세요.

예제 코드는 다음을 참조하십시오.

- Java: AsymmetricEncryptedItem
- Python: wrapped-rsa-encrypted-table, wrapped-symmetric-encrypted-table

주제

사용 방법

래핑된 CMP를 생성하려면 래핑 키(암호화에 필요), 래핑 해제 키(복호화에 필요) 및 서명 키를 지정합니다. 항목을 암호화하고 복호화할 때 키를 제공해야 합니다.

래핑, 래핑 해제 및 서명 키는 대칭 키 또는 비대칭 키 페어일 수 있습니다.

Java

```
// This example uses asymmetric wrapping and signing key pairs
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

final WrappedMaterialsProvider cmp =
    new WrappedMaterialsProvider(wrappingKeys.getPublic(),
                                 wrappingKeys.getPrivate(),
                                 signingKeys);
```

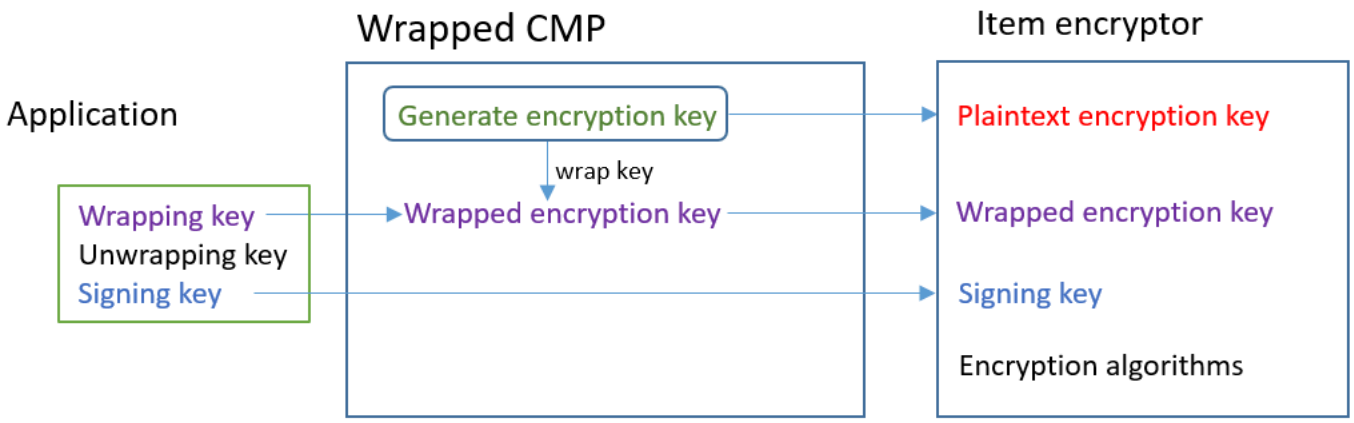
Python

```
# This example uses symmetric wrapping and signing keys
wrapping_key = ...
```

```
signing_key  = ...

wrapped_cmp = WrappedCryptographicMaterialsProvider(
    wrapping_key=wrapping_key,
    unwrapping_key=wrapping_key,
    signing_key=signing_key
)
```

작동 방식

래핑된 CMP는 모든 항목에 대해 새 항목 암호화 키를 생성합니다. 다음 다이어그램과 같이 사용자가 제공하는 래핑, 래핑 해제 및 서명 키를 사용합니다.



암호화 자료 가져오기

이 단원에서는 암호화 자료 요청 수신 시 래핑된 자료 공급자(래핑된 CMP)의 입력, 출력 및 처리에 대해 자세히 설명합니다.

입력(애플리케이션에서)

- 래핑 키: Advanced Encryption Standard(AES) 대칭 키 또는 RSA 퍼블릭 키. 속성 값이 암호화된 경우 필수입니다. 그렇지 않으면 선택 사항이며 무시됩니다.
- 래핑 해제 키: 선택 사항이며 무시됩니다.
- 서명 키

입력(항목 암호화 도구에서)

- DynamoDB 암호화 컨텍스트

출력(항목 암호화 도구로):

- 일반 텍스트 항목 암호화 키
- 서명 키(변경되지 않음)
- 실제 자료 설명: 이러한 값은 클라이언트가 항목에 추가하는 자료 설명 속성에 저장됩니다.
  - `amzn-ddb-env-key`: Base64로 인코딩되고 래핑된 항목 암호화 키
  - `amzn-ddb-env-alg`: 항목을 암호화하는 데 사용되는 암호화 알고리즘입니다. 기본값은 AES-256-CBC입니다.
  - `amzn-ddb-wrap-alg`: 래핑된 CMP가 항목 암호화 키를 래핑하는 데 사용한 래핑 알고리즘입니다. 래핑 키가 AES 키인 경우 RFC 3394에 정의된 대로 패딩되지 않은 `AES-Keywrap`를 사용하여 키가 래핑됩니다. 래핑 키가 RSA 키인 경우 MGF1 패딩이 포함된 RSA OAEP를 사용하여 키가 암호화됩니다.

처리

항목을 암호화할 때 래핑 키와 서명 키를 전달합니다. 래핑 해제 키는 선택 사항이며 무시됩니다.

1. 래핑된 CMP는 테이블 항목에 대한 고유한 대칭 항목 암호화 키를 생성합니다.
2. 항목 암호화 키를 래핑하기 위해 지정한 래핑 키를 사용합니다. 그런 다음 가능한 한 빨리 메모리에서 제거합니다.
3. Wrapped CMP는 일반 텍스트 항목 암호화 키, 제공한 서명 키, 그리고 래핑된 항목 암호화 키와 암호화 및 래핑 알고리즘을 포함하는 실제 자료 설명을 반환합니다.
4. 항목 암호화 도구는 일반 텍스트 암호화 키를 사용하여 항목을 암호화합니다. 항목에 서명하기 위해 제공한 서명 키를 사용합니다. 그런 다음 가능한 한 빨리 메모리에서 일반 텍스트 키를 제거합니다. 래핑된 암호화 키(`amzn-ddb-env-key`)를 포함하여 실제 자료 설명의 필드를 항목의 자료 설명 속성에 복사합니다.

복호화 자료 가져오기

이 단원에서는 복호화 자료 요청 수신 시 래핑된 자료 공급자(래핑된 CMP)의 입력, 출력 및 처리에 대해 자세히 설명합니다.

입력(애플리케이션에서)

- 래핑 키: 선택 사항이며 무시됩니다.

- 언래핑 키: 암호화하는 데 사용되는 RSA 퍼블릭 키에 해당하는 동일한 Advanced Encryption Standard(AES) 대칭 키 또는 RSA 프라이빗 키입니다. 속성 값이 암호화된 경우 필수입니다. 그렇지 않으면 선택 사항이며 무시됩니다.
- 서명 키

입력(항목 암호화 도구에서)

- 자료 설명 속성의 내용을 포함하는 DynamoDB 암호화 컨텍스트의 복사본입니다.

출력(항목 암호화 도구로)

- 일반 텍스트 항목 암호화 키
- 서명 키(변경되지 않음)

처리

항목의 암호를 복호화할 때 래핑 해제 키와 서명 키를 전달합니다. 래핑 키는 선택 사항이며 무시됩니다.

1. 래핑된 CMP는 항목의 자료 설명 속성에서 래핑된 항목 암호화 키를 가져옵니다.
2. 래핑 해제 키와 알고리즘을 사용하여 항목 암호화 키를 래핑 해제합니다.
3. 일반 텍스트 항목 암호화 키, 서명 키, 암호화 및 서명 알고리즘을 항목 암호화 도구에 반환합니다.
4. 항목 암호화기는 서명 키를 사용하여 항목을 확인합니다. 성공하면 항목 암호화 키를 사용하여 항목의 암호를 복호화합니다. 그런 다음 가능한 한 빨리 메모리에서 일반 텍스트 키를 제거합니다.

Most Recent Provider

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

Most Recent Provider는 공급자 스토어와 함께 작동하도록 설계된 암호화 자료 공급자(CMP)입니다. 공급자 스토어에서 CMP를 가져오고 CMP에서 반환한 암호화 자료를 가져옵니다. 일반적으로 각 CMP를 사용하여 암호화 자료에 대한 여러 요청을 충족합니다. 하지만 공급자 스토어의 기능을 사용하여 자료가 재사용되는 범위를 제어하고 CMP 교체 빈도를 결정할 수 있으며, Most Recent Provider를 변경하지 않고 사용하는 CMP 유형을 변경할 수도 있습니다.

> **Note**
>
> Most Recent Provider의 `MostRecentProvider` 기호와 연관된 코드는 프로세스 수명 동안 암호화 자료를 메모리에 저장할 수 있습니다. 호출자가 더 이상 사용 권한이 없는 키를 사용하도록 허용할 수도 있습니다.
>
> `MostRecentProvider` 기호는 지원되는 이전 버전의 DynamoDB Encryption Client에서 더 이상 사용되지 않으며 버전 2.0.0에서 제거되었습니다. `CachingMostRecentProvider` 기호로 대체됩니다. 자세한 내용은 Most Recent Provider 업데이트을 참조하세요.

Most Recent Provider는 공급자 스토어 및 관련 암호화 소스에 대한 호출을 최소화해야 하는 애플리케이션과, 보안 요구 사항을 위반하지 않으면서 일부 암호화 자료를 재사용할 수 있는 애플리케이션에 적합합니다. 예를 들어 항목을 암호화하거나 해독할 AWS KMS 때마다를 호출하지 않고 AWS Key Management Service (AWS KMS)의 AWS KMS key에서 암호화 자료를 보호할 수 있습니다.

선택하는 공급자 스토어는 Most Recent Provider에서 사용하는 CMP 유형과 새 CMP를 가져오는 빈도를 결정합니다. 사용자가 설계한 사용자 정의 공급자 저장소를 포함하여 Most Recent Provider와 호환되는 모든 공급자 스토어를 사용할 수 있습니다.

DynamoDB Encryption Client에는 래핑된 자료 공급자(래핑된 CMP)를 생성하고 반환하는 MetaStore가 포함되어 있습니다. MetaStore는 생성한 래핑된 CMP의 여러 버전을 내부 DynamoDB 테이블에 저장하고 DynamoDB Encryption Client의 내부 인스턴스를 통한 클라이언트측 암호화로 이를 보호합니다.

모든 유형의 내부 CMP를 사용하여 테이블의 자료를 보호하도록 MetaStore를 구성할 수 있습니다. 여기에는 로 보호되는 암호화 자료를 생성하는 다이렉트 KMS 공급자 AWS KMS key, 사용자가 제공하는 래핑 및 서명 키를 사용하는 래핑된 CMP 또는 사용자가 설계하는 호환되는 사용자 지정 CMP가 포함됩니다.

예제 코드는 다음을 참조하십시오.

- Java: MostRecentEncryptedItem
- Python: most_recent_provider_encrypted_table

주제

사용 방법

Most Recent Provider를 생성하려면 공급자 저장소를 생성 및 구성한 다음 공급자 스토어를 사용하는 Most Recent Provider를 생성해야 합니다.

다음 예제에서는 MetaStore를 사용하고 Direct KMS Provider의 암호화 자료로 내부 DynamoDB 테이블의 버전을 보호하는 Most Recent Provider를 생성하는 방법을 보여줍니다. 이 예제에서는 `CachingMostRecentProvider` 기호를 사용합니다.

각 Most Recent Provider에는 MetaStore 테이블에서 CMP를 식별하는 이름, time-to-live(TTL) 설정, 캐시가 보유할 수 있는 항목 수를 결정하는 캐시 크기 설정이 있습니다. 이 예제에서는 캐시 크기를 항목 1,000개와 TTL 한 개를 60초로 설정합니다.

Java

```
// Set the name for MetaStore's internal table
final String keyTableName = 'metaStoreTable'

// Set the Region and AWS KMS key
final String region = 'us-west-2'
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

// Set the TTL and cache size
final long ttlInMillis = 60000;
final long cacheSize = 1000;

// Name that identifies the MetaStore's CMPs in the provider store
final String materialName = 'testMRP'

// Create an internal DynamoDB client for the MetaStore
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

// Create an internal Direct KMS Provider for the MetaStore
final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
```

```
final DirectKmsMaterialProvider kmsProv = new DirectKmsMaterialProvider(kms,
 keyArn);

// Create an item encryptor for the MetaStore,
// including the Direct KMS Provider
final DynamoDBEncryptor keyEncryptor = DynamoDBEncryptor.getInstance(kmsProv);

// Create the MetaStore
final MetaStore metaStore = new MetaStore(ddb, keyTableName, keyEncryptor);

//Create the Most Recent Provider
final CachingMostRecentProvider cmp = new CachingMostRecentProvider(metaStore,
 materialName, ttlInMillis, cacheSize);
```

Python

```
# Designate an AWS KMS key
kms_key_id = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

# Set the name for MetaStore's internal table
meta_table_name = 'metaStoreTable'

# Name that identifies the MetaStore's CMPs in the provider store
material_name = 'testMRP'

# Create an internal DynamoDB table resource for the MetaStore
meta_table = boto3.resource('dynamodb').Table(meta_table_name)

# Create an internal Direct KMS Provider for the MetaStore
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)

# Create the MetaStore with the Direct KMS Provider
meta_store = MetaStore(
    table=meta_table,
    materials_provider=kms_cmp
)

# Create a Most Recent Provider using the MetaStore
#    Sets the TTL (in seconds) and cache size (# entries)
most_recent_cmp = MostRecentProvider(
    provider_store=meta_store,
    material_name=material_name,
```

```
    version_ttl=60.0,
    cache_size=1000
)
```

작동 방식

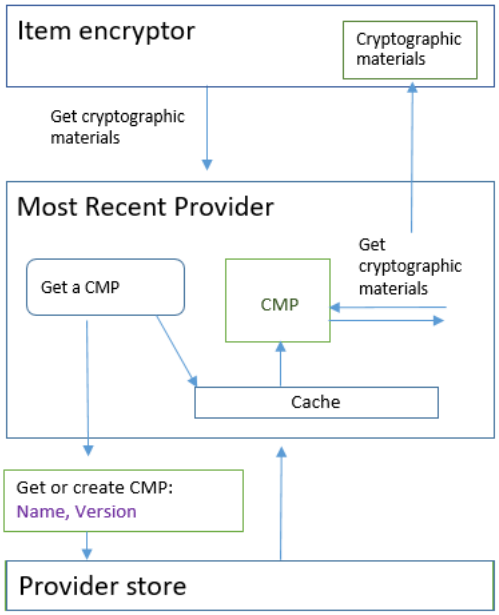
Most Recent Provider는 공급자 스토어에서 CMP를 가져옵니다. 그런 다음 CMP를 사용하여 항목 암호화 도구에 반환되는 암호화 자료를 생성합니다.

최신 제공자 정보

Most Recent Provider는 공급자 스토어에서 암호화 자료 공급자(CMP)를 가져옵니다. 그런 다음 CMP를 사용하여 반환되는 암호화 자료를 생성합니다. 각 Most Recent Provider는 하나의 공급자 스토어와 연결되어 있지만 공급자 스토어는 여러 호스트에 걸쳐 여러 공급자에게 CMP를 제공할 수 있습니다.

Most Recent Provider는 모든 공급자 스토어의 호환되는 CMP와 함께 사용할 수 있습니다. CMP에서 암호화 또는 복호화 자료를 요청하고 출력을 항목 암호화 도구로 반환합니다. 어떠한 암호화 작업도 수행하지 않습니다.

Most Recent Provider는 공급자 스토어에서 CMP를 요청하기 위해 관련 자료 이름과 사용할 기존 CMP의 버전을 제공합니다. 암호화 자료의 경우 Most Recent Provider는 항상 최대("최신") 버전을 요청합니다. 복호화 자료의 경우 다음 다이어그램과 같이 암호화 자료를 생성하는 데 사용된 CMP 버전을 요청합니다.



Most Recent Provider는 공급자 스토어에서 반환하는 CMP의 버전을 메모리의 로컬 Least Recently Used(LRU) 캐시에 저장합니다. 이 캐시에서는 Most Recent Provider가 모든 항목에 대해 공급자 스토어를 호출하지 않고 필요한 CMP를 가져올 수 있습니다. 요청 시 캐시를 지울 수 있습니다.

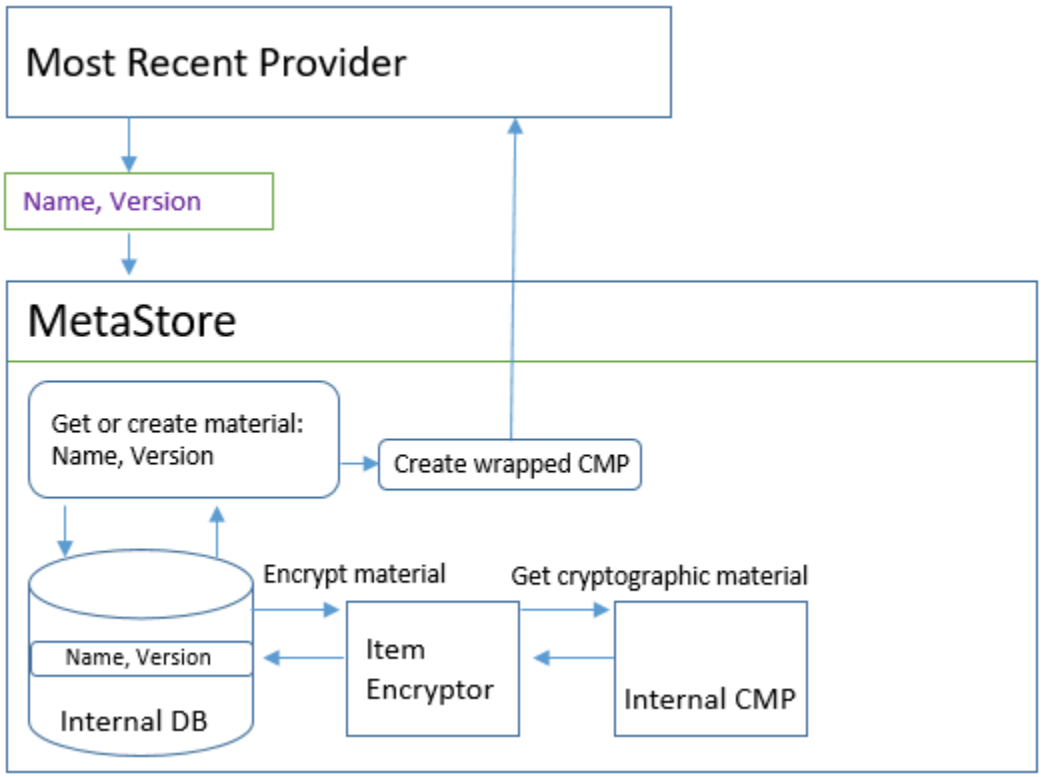
Most Recent Provider는 애플리케이션의 특성에 따라 조정할 수 있는 구성 가능한 time-to-live 값을 사용합니다.

MetaStore 정보

호환되는 사용자 지정 공급자 스토어를 비롯하여 모든 공급자 스토어와 함께 Most Recent Provider를 사용할 수 있습니다. DynamoDB Encryption Client에는 구성하고 사용자 지정할 수 있는 보안 구현인 MetaStore가 포함되어 있습니다.

MetaStore는 래핑된 CMP에 필요한 래핑 키, 언래핑 키, 서명 키로 구성된 래핑된 CMP를 생성하고 반환하는 공급자 스토어입니다. 래핑된 CMP는 항상 모든 항목에 대해 고유한 항목 암호화 키를 생성하므로 MetaStore는 Most Recent Provider를 위한 안전한 옵션입니다. 항목 암호화 키와 서명 키를 보호하는 래핑 키만 재사용됩니다.

다음 다이어그램은 MetaStore의 구성 요소, 그리고 이러한 구성 요소가 Most Recent Provider와 어떻게 상호 작용하는지를 보여줍니다.



MetaStore는 래핑된 CMP를 생성한 다음 이를 암호화된 형식으로 내부 DynamoDB 테이블에 저장합니다. 파티션 키는 Most Recent Provider 자료의 이름입니다. 정렬 키의 버전 번호입니다. 테이블의 자료는 항목 암호화 도구 및 내부 암호화 자료 공급자(CMP)를 포함한 내부 DynamoDB Encryption Client에 의해 보호됩니다.

Direct KMS Provider, Wrapped CMP(사용자가 제공하는 암호화 자료 포함) 또는 호환되는 사용자 지정 CMP 등 MetaStore에서 원하는 내부 CMP 유형을 사용할 수 있습니다. MetaStore의 내부 CMP가 Direct KMS Provider인 경우 재사용 가능한 래핑 및 서명 키는 AWS Key Management Service(AWS KMS)의 AWS KMS key에 따라 보호됩니다. MetaStore는 내부 테이블에 새 CMP 버전을 추가하거나 내부 테이블에서 CMP 버전을 가져올 AWS KMS 때마다를 호출합니다.

time-to-live 값 설정

생성한 각 Most Recent Provider에 대해 time-to-live(TTL) 값을 설정할 수 있습니다. 일반적으로 애플리케이션에 실용적인 가장 낮은 TTL 값을 사용합니다.

Most Recent Provider에 대한 `CachingMostRecentProvider` 기호에서 TTL 값의 사용이 변경되었습니다.

> **Note**
>
> Most Recent Provider에 대한 `MostRecentProvider` 기호는 지원되는 이전 버전의 DynamoDB Encryption Client에서 더 이상 사용되지 않으며 버전 2.0.0에서 제거됩니다. `CachingMostRecentProvider` 기호로 대체됩니다. 가능한 한 빨리 코드를 업데이트하는 것이 좋습니다. 자세한 내용은 Most Recent Provider 업데이트을 참조하세요.

`CachingMostRecentProvider`

`CachingMostRecentProvider`는 TTL 값은 두 가지 다른 방식으로 사용됩니다.

- TTL은 Most Recent Provider가 공급자 스토어에서 CMP의 새 버전을 확인하는 빈도를 결정합니다. 새 버전을 사용할 수 있는 경우 Most Recent Provider는 CMP를 교체하고 암호화 자료를 새로 고칩니다. 그렇지 않으면 현재 CMP 및 암호화 자료를 계속 사용합니다.
- TTL은 캐시의 CMP를 사용할 수 있는 기간을 결정합니다. Most Recent Provider는 암호화를 위해 캐시된 CMP를 사용하기 전에 캐시에 있는 시간을 평가합니다. CMP 캐시 시간이 TTL을 초과하면 CMP가 캐시에서 제거되고 Most Recent Provider는 해당 공급자 저장소에서 새로운 최신 버전의 CMP를 가져옵니다.

`MostRecentProvider`

`MostRecentProvider`에서 TTL은 Most Recent Provider가 공급자 스토어에서 CMP의 새 버전을 확인하는 빈도를 결정합니다. 새 버전을 사용할 수 있는 경우 Most Recent Provider는 CMP를 교체하고 암호화 자료를 새로 고칩니다. 그렇지 않으면 현재 CMP 및 암호화 자료를 계속 사용합니다.

TTL은 새 CMP 버전이 생성되는 빈도를 결정하지 않습니다. 암호화 자료를 교체하여 새로운 CMP 버전을 생성합니다.

이상적인 TTL 값은 애플리케이션과 해당 지연 시간 및 가용성 목표에 따라 다릅니다. TTL이 낮을수록 암호화 자료가 메모리에 저장되는 시간이 줄어들어 보안 프로필이 향상됩니다. 또한 TTL이 낮을수록 중요한 정보가 더 자주 새로 고쳐집니다. 예를 들어 내부 CMP가 Direct KMS Provider인 경우, 발신자가 여전히 AWS KMS key를 사용할 권한이 있는지 더 자주 확인합니다.

그러나 TTL이 너무 짧은 경우 공급자 스토어를 자주 호출하면 비용이 증가하고 공급자 스토어가 애플리케이션 및 서비스 계정을 공유하는 기타 애플리케이션의 요청을 제한할 수 있습니다. 암호화 자료를 회전하는 속도에 따라 TTL을 조정하면 이점을 얻을 수도 있습니다.

테스트하는 동안 애플리케이션과 보안 및 성능 표준에 적합한 구성을 찾을 때까지 다양한 작업 부하에 따라 TTL과 캐시 크기를 다양하게 변경합니다.

암호화 자료 교체

Most Recent Provider는 암호화 자료가 필요할 때 항상 자신이 알고 있는 CMP의 최신 버전을 사용합니다. 최신 버전을 확인하는 빈도는 가장 Most Recent Provider를 구성할 때 설정한 time-to-live(TTL) 값에 따라 결정됩니다.

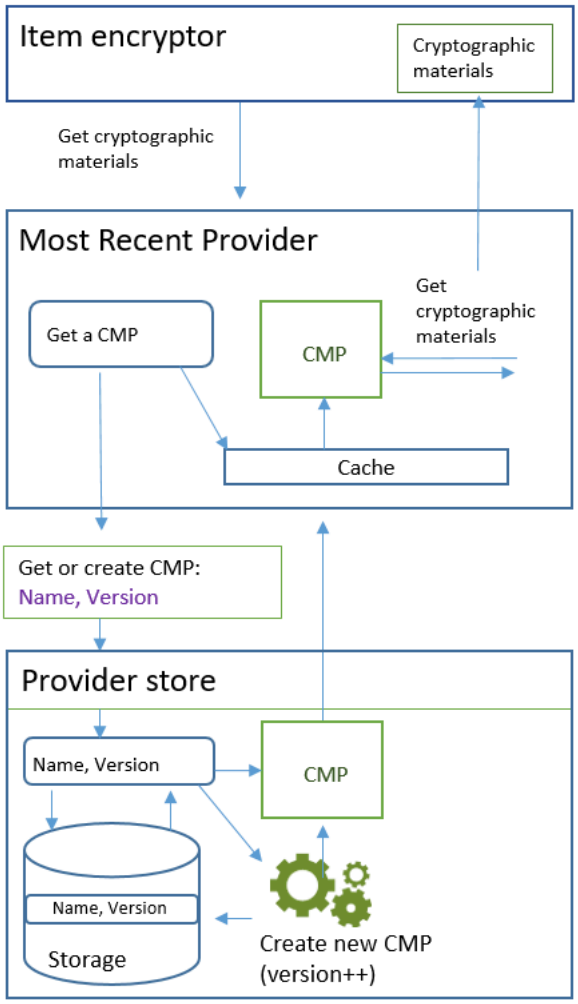
TTL이 만료되면 Most Recent Provider는 공급자 저장소에서 최신 버전의 CMP를 확인합니다. 사용 가능한 CMP가 있는 경우 Most Recent Provider는 이를 가져와 캐시의 CMP를 교체합니다. 공급자 스토어에 최신 버전이 있다는 것을 발견할 때까지 이 CMP와 해당 암호화 자료를 사용합니다.

공급자 스토어에 Most Recent Provider로 새 CMP 버전을 만들도록 알리려면 Most Recent Provider의 자료 이름을 사용하여 공급자 스토어의 새 공급자 만들기 작업을 호출합니다. 공급자 스토어는 새 CMP를 생성하고 더 높은 버전 번호로 내부 스토어에 암호화된 복사본을 저장합니다. (CMP도 반환하지만 삭제할 수 있습니다.) 결과적으로 Most Recent Provider는 다음 번에 공급자 스토어에 CMP의 최대 버전 번호를 쿼리할 때 더 큰 새로운 버전 번호를 얻고 이를 저장소에 대한 후속 요청에 사용하여 CMP의 새 버전이 생성되었는지 확인합니다.

시간, 처리된 항목이나 속성의 수, 애플리케이션에 적합한 기타 지표를 기준으로 새 공급자 생성 호출을 예약할 수 있습니다.

암호화 자료 가져오기

Most Recent Provider는 이 다이어그램에 표시된 다음과 같은 프로세스를 통해 항목 암호화 도구로 반환되는 암호화 자료를 가져옵니다. 출력은 공급자 스토어가 반환하는 CMP 유형에 따라 달라집니다. Most Recent Provider는 DynamoDB Encryption Client에 포함된 MetaStore를 포함하여 호환되는 모든 공급자 스토어를 사용할 수 있습니다.



`CachingMostRecentProvider` 기호를 사용하여 Most Recent Provider를 생성할 때 공급자 스토어, Most Recent Provider의 이름 및 time-to-live(TTL) 값을 지정합니다. 캐시에 존재할 수 있는 암호화 자료의 최대 수를 결정하는 캐시 크기를 선택적으로 지정할 수도 있습니다.

이 항목이 Most Recent Provider에 암호화 자료를 요청하면 Most Recent Provider는 캐시에서 CMP의 최신 버전을 검색하는 것부터 시작합니다.

- 캐시에서 최신 버전의 CMP를 찾았고 CMP가 TTL 값을 초과하지 않은 경우 Most Recent Provider는 CMP를 사용하여 암호화 자료를 생성합니다. 그런 다음 암호화 자료를 항목 암호화 도구에 반환합니다. 이 작업에는 공급자 스토어를 호출할 필요가 없습니다.
- 최신 버전의 CMP가 캐시에 없거나 캐시에 있지만 TTL 값을 초과한 경우 Most Recent Provider는 공급자 스토어에서 CMP를 요청합니다. 요청에는 Most Recent Provider 자료 이름과 알고 있는 최대 버전 번호가 포함됩니다.
  1. 공급자 스토어는 영구 스토리지에서 CMP를 반환합니다. 공급자 스토어가 MetaStore인 경우 가장 최근 공급자 자료 이름을 파티션 키로 사용하고, 버전 번호를 정렬 키로 사용하여 내부 DynamoDB 테이블에서 암호화되고 래핑된 CMP를 가져옵니다. MetaStore는 내부 항목 암호화 도구와 내부 CMP를 사용하여 래핑된 CMP를 복호화합니다. 그런 다음 일반 텍스트 CMP를 Most Recent Provider에 반환합니다. 내부 CMP가 Direct KMS Provider이면 이 단계에는 AWS Key Management Service(AWS KMS)에 대한 호출이 포함됩니다.
  2. CMP 는 `amzn-ddb-meta-id` 필드를 실제 자료 설명에 추가합니다. 해당 값은 내부 테이블에 있는 CMP의 자료 이름과 버전입니다. 공급자 스토어는 CMP를 Most Recent Provider에게 반환합니다.
  3. Most Recent Provider는 CMP를 메모리에 캐시합니다.
  4. Most Recent Provider는 CMP를 사용하여 암호화 자료를 생성합니다. 그런 다음 암호화 자료를 항목 암호화 도구에 반환합니다.

복호화 자료 가져오기

항목 암호화 도구가 Most Recent Provider에 복호화 자료를 요청하는 경우 Most Recent Provider는 다음 프로세스를 사용하여 이러한 자료를 가져와서 반환합니다.

1. Most Recent Provider는 공급자 스토어에 항목을 암호화하는 데 사용되었던 암호화 자료의 버전 번호를 요청합니다. Most Recent Provider는 항목의 자료 설명 속성에서 실제 자료 설명을 전달합니다.
2. 공급자 스토어는 실제 자료 설명의 `amzn-ddb-meta-id` 필드에서 암호화 CMP 버전 번호를 가져와서 Most Recent Provider로 반환합니다.
3. Most Recent Provider는 항목을 암호화 및 서명하는 데 사용되었던 CMP 버전을 캐시에서 검색합니다.

- CMP의 일치하는 버전이 캐시에 있고 CMP가 time-to-live(TTL) 값을 초과하지 않은 경우 Most Recent Provider는 CMP를 사용하여 복호화 자료를 생성합니다. 그런 다음 복호화 자료를 항목 암호화 도구에 반환합니다. 이 작업에는 공급자 스토어나 다른 CMP에 대한 호출이 필요하지 않습니다.

- CMP의 일치하는 버전이 캐시에 없거나 캐시된 AWS KMS key 이 TTL 값을 초과한 경우 Most Recent Provider는 공급자 스토어에서 CMP를 요청합니다. 요청에 자료 이름과 암호화 CMP 버전 번호를 보냅니다.

  1. 공급자 스토어는 Most Recent Provider 이름을 파티션 키로 사용하고 버전 번호를 정렬 키로 사용하여 CMP에 대한 영구 스토리지를 검색합니다.

     - 이름과 버전 번호가 영구 스토리지에 없으면 공급자 스토어에서 예외가 발생합니다. 공급자 스토어를 사용하여 CMP를 생성한 경우 의도적으로 삭제하지 않는 한 CMP는 영구 스토리지에 저장되어야 합니다.
     - CMP와 해당 이름 및 버전 번호가 공급자 스토어의 영구 스토리지 안에 있으면 공급자 스토어가 지정된 CMP를 Most Recent Provider로 반환합니다.

       공급자 스토어가 MetaStore인 경우 DynamoDB 테이블에서 암호화된 CMP를 가져옵니다. 그런 다음 CMP를 Most Recent Provider로 반환하기 전에 내부 CMP의 암호화 자료를 사용하여 암호화된 CMP를 복호화합니다. 내부 CMP가 Direct KMS Provider이면 이 단계에는 AWS Key Management Service(AWS KMS)에 대한 호출이 포함됩니다.

  2. Most Recent Provider는 CMP를 메모리에 캐시합니다.
  3. Most Recent Provider는 CMP를 사용하여 복호화 자료를 생성합니다. 그런 다음 복호화 자료를 항목 암호화 도구에 반환합니다.

Most Recent Provider 업데이트

Most Recent Provider의 기호가 `MostRecentProvider`에서 `CachingMostRecentProvider`로 변경되었습니다.

> **Note**
>
> Most Recent Provider를 나타내는 `MostRecentProvider` 기호는 Java용 DynamoDB Encryption Client 버전 1.15 및 Python용 DynamoDB Encryption Client 버전 1.3에서 더 이상 사용되지 않으며 두 언어 구현 모두의 DynamoDB Encryption Client 버전 2.0.0에서 제거됩니다. 그 대신 `CachingMostRecentProvider`을 사용합니다.

`CachingMostRecentProvider`에서는 다음 변경 사항을 구현합니다.

- `CachingMostRecentProvider`는 메모리 내 시간이 구성된 time-to-live(TTL) 값을 초과하는 경우 메모리에서 암호화 자료를 주기적으로 제거합니다.

`MostRecentProvider`는 프로세스 수명 동안 암호화 자료를 메모리에 저장할 수 있습니다. 결과적으로 Most Recent Provider는 인증 변경 사항을 인식하지 못할 수도 있습니다. 호출자의 암호화 키 사용 권한이 취소된 후 암호화 키를 사용할 수 있습니다.

이 새 버전으로 업데이트할 수 없는 경우 주기적으로 캐시에서 `clear()` 방법를 호출하면 비슷한 효과를 얻을 수 있습니다. 이 방법를 사용하면 캐시 콘텐츠를 수동으로 플러시하고 Most Recent Provider가 새 CMP 및 새 암호화 자료를 요청해야 합니다.

- 또한 `CachingMostRecentProvider`에는 캐시를 더 효과적으로 제어할 수 있는 캐시 크기 설정도 포함되어 있습니다.

`CachingMostRecentProvider`로 업데이트하려면 코드에서 기호 이름을 변경해야 합니다. 다른 모든 측면에서 `CachingMostRecentProvider`는 `MostRecentProvider`와 완전히 역호환됩니다. 테이블 항목을 다시 암호화할 필요가 없습니다.

그러나 `CachingMostRecentProvider`는 기본 키 인프라에 대한 더 많은 호출을 생성합니다. 각 time-to-live(TTL) 간격마다 공급자 스토어를 한 번 이상 호출합니다. 활성 CMP가 많은 애플리케이션(빈번한 교체로 인해) 또는 대규모 플릿이 있는 애플리케이션은 이러한 변화에 민감할 가능성이 높습니다.

업데이트된 코드를 릴리스하기 전에 철저히 테스트하여 더 빈번한 호출로 인해 애플리케이션이 손상되거나 AWS Key Management Service (AWS KMS) 또는 Amazon DynamoDB와 같이 공급자가 의존하는 서비스에 의한 제한이 발생하지 않는지 확인합니다. 성능 문제를 완화하려면 관찰한 성능 특성에 따라 캐시 크기와 `CachingMostRecentProvider`의 수명을 조정합니다. 자세한 지침은 time-to-live 값 설정을 참조하세요.

Static Materials Provider

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

Static Materials Provider(정적 CMP)는 테스트, 개념 증명 데모 및 레거시 호환성을 위한 매우 간단한 암호화 자료 공급자(CMP)입니다.

Static CMP를 사용하여 테이블 항목을 암호화하려면 Advanced Encryption Standard(AES) 대칭 암호화 키 및 서명 키 또는 키 페어를 제공합니다. 암호화된 항목을 복호화하려면 동일한 키를 제공해야 합니다. 정적 CMP는 어떠한 암호화 작업도 수행하지 않습니다. 대신, 사용자가 제공한 암호화 키를 항목 암호화 도구에 변경되지 않은 상태로 전달합니다. 항목 암호화 도구는 암호화 키 바로 아래에 있는 항목을 암호화합니다. 그런 다음 서명 키를 직접 사용하여 서명합니다.

정적 CMP는 고유한 암호화 자료를 생성하지 않기 때문에 처리하는 모든 테이블 항목은 동일한 암호화 키로 암호화되고 동일한 서명 키로 서명됩니다. 동일한 키를 사용하여 수많은 항목의 속성 값을 암호화하거나 동일한 키 또는 키 페어를 사용하여 모든 항목에 서명하면 키의 암호화 제한을 초과할 위험이 있습니다.

> **Note**
>
> Java 라이브러리의 Asymmetric Static Provider는 Static Provider가 아닙니다. 단순히 Wrapped CMP에 대한 대체 생성자를 제공할 뿐입니다. 프로덕션 환경에서는 안전하지만 가능하면 래핑된 CMP를 직접 사용해야 합니다.

정적 CMP는 DynamoDB Encryption Client가 지원하는 여러 암호화 자료 공급자(CMP) 중 하나입니다. 기타 CMP에 대한 자세한 내용은 암호화 자료 공급자 섹션을 참조하세요.

예제 코드는 다음을 참조하십시오.

- Java: SymmetricEncryptedItem

주제

- 사용 방법
- 작동 방식

사용 방법

정적 공급자를 생성하려면 암호화 키 또는 키 페어와 서명 키 또는 키 페어를 제공합니다. 테이블 항목을 암호화하고 복호화하려면 키 자료를 제공해야 합니다.

Java

```
// To encrypt
SecretKey cek = ...;        // Encryption key
```

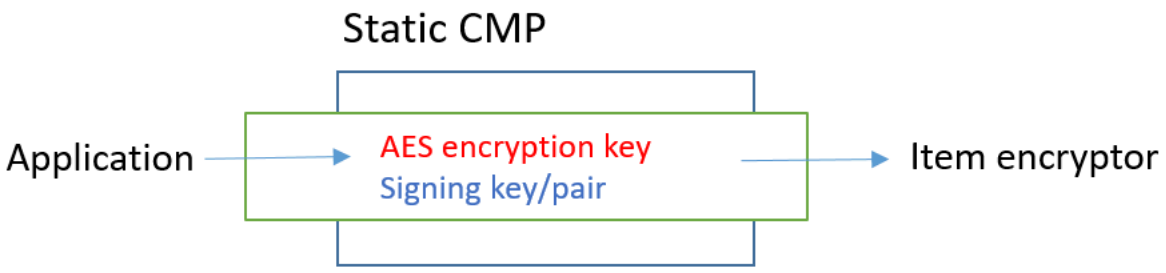
```
SecretKey macKey =  ...;    // Signing key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);

// To decrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Verification key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);
```

Python

```
# You can provide encryption materials, decryption materials, or both
encrypt_keys = EncryptionMaterials(
    encryption_key = ...,
    signing_key = ...
)

decrypt_keys = DecryptionMaterials(
    decryption_key = ...,
    verification_key = ...
)

static_cmp = StaticCryptographicMaterialsProvider(
    encryption_materials=encrypt_keys
    decryption_materials=decrypt_keys
)
```

작동 방식

정적 공급자는 항목 암호화 도구에 제공하는 암호화 및 서명 키를 전달합니다(암호화 및 서명 키가 테이블 항목을 암호화 및 서명하는 데 직접 사용된 경우). 각 항목에 대해 다른 키를 제공하지 않는 한 모든 항목에 동일한 키가 사용됩니다.



암호화 자료 가져오기

이 단원에서는 암호화 자료 요청 수신 시 Static Materials Provider(정적 CMP)의 입력, 출력 및 처리에 대해 자세히 설명합니다.

입력(애플리케이션에서)

- 암호화 키 – Advanced Encryption Standard(AES) 키와 같은 대칭 키여야 합니다.
- 서명 키 – 대칭 키 또는 비대칭 키 페어일 수 있습니다.

입력(항목 암호화 도구에서)

- DynamoDB 암호화 컨텍스트

출력(항목 암호화 도구로)

- 암호화 키가 입력으로 전달되었습니다.
- 서명 키가 입력으로 전달되었습니다.
- 실제 자료 설명: 변경되지 않은 요청한 자료 설명(있는 경우).

복호화 자료 가져오기

이 단원에서는 복호화 자료 요청 수신 시 Static Materials Provider(정적 CMP)의 입력, 출력 및 처리에 대해 자세히 설명합니다.

암호화 자료를 가져오는 방법과 복호화 자료를 가져오는 방법이 별도로 포함되어 있지만 동작은 동일합니다.

입력(애플리케이션에서)

- 암호화 키 – Advanced Encryption Standard(AES) 키와 같은 대칭 키여야 합니다.
- 서명 키 – 대칭 키 또는 비대칭 키 페어일 수 있습니다.

입력(항목 암호화 도구에서)

- DynamoDB 암호화 컨텍스트(사용되지 않음)

출력(항목 암호화 도구로)

- 암호화 키가 입력으로 전달되었습니다.
- 서명 키가 입력으로 전달되었습니다.

Amazon DynamoDB Encryption Client에서 사용할 수 있는 프로그래밍 언어

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

Amazon DynamoDB Encryption Client는 다음 프로그래밍 언어에 사용할 수 있습니다. 언어별 라이브러리는 다양하지만 결과 구현은 상호 운용이 가능합니다. 예를 들어 Java 클라이언트로 항목을 암호화(및 서명)하고 Python 클라이언트로 항목을 복호화할 수 있습니다.

자세한 내용은 해당 주제를 참조하세요.

주제

- Amazon DynamoDB Encryption Client for Java
- DynamoDB Encryption Client for Python

Amazon DynamoDB Encryption Client for Java

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

이 주제에서는 Amazon DynamoDB Encryption Client for Java를 설치하고 사용하는 방법을 설명합니다. DynamoDB Encryption Client를 사용한 프로그래밍에 대한 자세한 내용은 Java 예제, GitHub의 aws-dynamodb-encryption-java 리포지토리에 있는 예제 및 DynamoDB Encryption Client용 Javadoc을 참조하세요.

> **Note**
>
> Java용 DynamoDB Encryption Client 버전 1.x.x는 2022년 7월부터 지원 종료 단계에 있습니다. 가능한 한 빨리 최신 버전으로 업그레이드하세요.

주제

사전 조건

Amazon DynamoDB Encryption Client for Java를 설치하기 전에 다음 사전 조건이 충족되었는지 확인합니다.

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.

AWS SDK for Java

애플리케이션이 DynamoDB와 상호 작용 AWS SDK for Java 하지 않더라도 DynamoDB Encryption Client에는의 DynamoDB 모듈이 필요합니다. 전체 SDK를 설치하거나 이 모듈만 설치할 수 있습니다. Maven을 사용하는 경우 `pom.xml` 파일에 `aws-java-sdk-dynamodb`을 추가합니다.

설치 및 구성에 대한 자세한 내용은 섹션을 AWS SDK for Java참조하세요AWS SDK for Java.

설치

다음과 같은 방법으로 Amazon DynamoDB Encryption Client for Java를 설치할 수 있습니다.

직접

Amazon DynamoDB Encryption Client for Java를 설치하려면 aws-dynamodb-encryption-java GitHub 리포지토리를 복제하거나 다운로드하세요.

Apache Maven 사용

Amazon DynamoDB Encryption Client for Java는 다음 종속성 정의와 함께 Apache Maven을 통해 사용할 수 있습니다.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

SDK를 설치한 후에는 이 가이드의 예제 코드와 GitHub의 DynamoDB Encryption Client Javadoc을 살펴보는 것부터 시작합니다.

DynamoDB Encryption Client for Java 사용

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

이 주제에서는 다른 프로그래밍 언어 구현에서는 찾을 수 없는 Java의 DynamoDB Encryption Client 기능 중 일부를 설명합니다.

DynamoDB Encryption Client를 사용한 프로그래밍에 대한 자세한 내용은 Java 예제, GitHub에 대한 `aws-dynamodb-encryption-java repository`의 예제 및 DynamoDB Encryption Client용 Javadoc를 참조하세요.

주제

- 항목 암호화 도구: AttributeEncryptor 및 DynamoDBEncryptor

- 저장 동작 구성
- Java의 속성 작업
- 테이블 이름 재정의

항목 암호화 도구: AttributeEncryptor 및 DynamoDBEncryptor

Java의 DynamoDB Encryption Client에는 두 개의 항목 암호화 도구, 즉 하위 수준 DynamoDBEncryptor 및 AttributeEncryptor가 있습니다.

`AttributeEncryptor`는 DynamoDB Encryption Client의에서 DynamoDBMapper를 사용하는 데 도움이 `DynamoDB Encryptor` 되는 헬퍼 클래스입니다. AWS SDK for Java DynamoDB `DynamoDBMapper`와 함께 `AttributeEncryptor`를 사용하면 사용자가 항목을 저장할 때 항목을 사용자 모르게 암호화하고 서명합니다. 또한 사용자가 항목을 로드할 때 항목을 사용자 모르게 확인하고 복호화합니다.

저장 동작 구성

`AttributeEncryptor` 및 `DynamoDBMapper`를 사용하여 서명만 있거나 암호화 및 서명된 속성이 있는 테이블 항목을 추가하거나 교체할 수 있습니다. 이러한 작업의 경우 다음 예와 같이 `PUT` 저장 동작을 사용하도록 구성하는 것이 좋습니다. 그렇지 않으면 데이터를 복호화하지 못할 수 있습니다.

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

테이블 항목에서 모델링된 속성만 업데이트하는 기본 저장 동작을 사용하는 경우 모델링되지 않은 속성은 서명에 포함되지 않으며 테이블 쓰기에 의해 변경되지 않습니다. 따라서 모델링되지 않은 속성을 포함하지 않기 때문에 나중에 모든 속성을 읽을 때 서명이 검증되지 않습니다.

`CLOBBER` 저장 동작을 사용할 수도 있습니다. 이 동작은 낙관적 잠금을 비활성화하고 테이블의 항목을 덮어쓴다는 점을 제외하면 `PUT` 저장 동작과 동일합니다.

서명 오류를 방지하기 위해 DynamoDB Encryption Client는 `AttributeEncryptor`를 저장 동작이 `CLOBBER` 또는 `PUT`로 구성되지 않은 `DynamoDBMapper`와 함께 사용하는 경우 런타임 예외를 발생시킵니다.

예제에 사용된 이 코드를 보려면 DynamoDBMapper 사용 및 GitHub의 `aws-dynamodb-encryption-java` 리포지토리에 있는 AwsKmsEncryptedObject.java 예제를 참조하세요.

Java의 속성 작업

속성 작업은 암호화 및 서명되는 속성 값, 서명만 되는 속성 값 및 무시되는 속성 값을 결정합니다. 속성 작업을 지정하는 데 사용하는 메서드는 `DynamoDBMapper` 및 `AttributeEncryptor`, 또는 하위 수준 DynamoDBEncryptor 중 어느 것을 사용하는지에 따라 달라집니다.

> **Important**
>
> 속성 작업을 사용하여 테이블 항목을 암호화한 후 데이터 모델에서 속성을 추가하거나 제거하면 서명 검증 오류가 발생하여 데이터를 복호화하지 못할 수 있습니다. 자세한 내용은 데이터 모델 변경 단원을 참조하십시오.

DynamoDBMapper에 대한 속성 작업

`DynamoDBMapper` 및 `AttributeEncryptor`를 사용하는 경우 주석을 사용하여 속성 작업을 지정합니다. DynamoDB Encryption Client는 속성 유형을 정의하는 표준 DynamoDB 속성 주석을 사용하여 속성을 보호하는 방법을 결정합니다. 기본적으로 기본 키(서명되지만 암호화되지 않음)를 제외하고는 모든 속성이 암호화 및 서명됩니다.

> **Note**
>
> 서명할 수 있고 서명해야 하더라도 @DynamoDBVersionAttribute 주석을 사용하여 속성 값을 암호화하지 마십시오. 그렇지 않으면 해당 값을 사용하는 조건이 의도하지 않은 영향을 미치게 됩니다.

```
// Attributes are encrypted and signed
@DynamoDBAttribute(attributeName="Description")

// Partition keys are signed but not encrypted
@DynamoDBHashKey(attributeName="Title")

// Sort keys are signed but not encrypted
@DynamoDBRangeKey(attributeName="Author")
```

예외를 지정하려면 DynamoDB Encryption Client for Java에 정의된 암호화 주석을 사용합니다. 클래스 수준에서 지정하면 해당 값이 클래스의 기본값이 됩니다.

```
// Sign only
```

```
@DoNotEncrypt

// Do nothing; not encrypted or signed
@DoNotTouch
```

예를 들어 이러한 주석은 PublicationYear 속성에 서명하지만 암호화하지는 않으며 ISBN 속성 값을 암호화하거나 서명하지 않습니다.

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

DynamoDBEncryptor에 대한 속성 작업

DynamoDBEncryptor를 직접 사용하는 경우 속성 작업을 지정하려면 이름-값 페어가 속성 이름 및 지정한 작업을 표현하는 HashMap 객체를 만듭니다.

속성 작업에 유효한 값은 EncryptionFlags 열거 유형으로 정의되어 있습니다. ENCRYPT 및 SIGN와 함께 사용하거나 SIGN 단독으로 사용하거나 둘 다 생략할 수 있습니다. 하지만 ENCRYPT 단독으로 사용하는 경우 DynamoDB Encryption Client에서 오류가 발생합니다. 서명하지 않은 속성은 암호화할 수 없습니다.

```
ENCRYPT
SIGN
```

> **Warning**
>
> 기본 키 속성은 암호화하지 마십시오. 일반 텍스트로 남겨 두어야 DynamoDB에서 전체 테이블 스캔을 실행하지 않고 해당 항목을 찾을 수 있습니다.

암호화 컨텍스트에서 프라이머리 키를 지정하고 나서 프라이머리 키 속성에 대한 속성 작업에서 ENCRYPT를 지정하는 경우 DynamoDB Encryption Client에서 예외가 발생합니다.

예를 들어 다음 Java 코드는 `record` 항목의 모든 속성을 암호화하고 서명하는 `actions` HashMap을 만듭니다. 서명되었지만 암호화되지 않은 파티션 키 및 정렬 키 속성 및 서명되거나 암호화되지 않은 `test` 속성은 예외입니다.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // no break; falls through to next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Don't encrypt or sign
      break;
    default:
      // Encrypt and sign everything else
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

그런 다음 `DynamoDBEncryptor`의 encryptRecord 방법을 호출할 때 맵을 `attributeFlags` 파라미터의 값으로 지정합니다. 예를 들어, `encryptRecord`에 대한 이 호출은 `actions` 맵을 사용합니다.

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

테이블 이름 재정의

DynamoDB Encryption Client에서 DynamoDB 테이블의 이름은 암호화 및 복호화 메서드에 전달되는 DynamoDB 암호화 컨텍스트의 요소입니다. 테이블 항목을 암호화하거나 서명할 때 테이블 이름을 포함한 DynamoDB 암호화 컨텍스트는 암호화 텍스트에 암호로 바인딩됩니다. decrypt 방법에 전달된 DynamoDB 암호화 컨텍스트가 encrypt 방법에 전달된 DynamoDB 암호화 컨텍스트와 일치하지 않으면 복호화 작업이 실패합니다.

테이블을 백업하거나 특정 시점으로 복구를 수행할 때와 같이 테이블 이름이 변경되는 경우도 있습니다. 이러한 항목의 서명을 복호화하거나 확인할 때 원래 테이블 이름을 포함하여 항목을 암호화하고 서명하는 데 사용된 것과 동일한 DynamoDB 암호화 컨텍스트를 전달해야 합니다. 현재 테이블 이름은 필요하지 않습니다.

`DynamoDBEncryptor`를 사용하는 경우 DynamoDB 암호화 컨텍스트를 수동으로 결합합니다. 그러나 `DynamoDBMapper`를 사용하는 경우 `AttributeEncryptor`는 현재 테이블 이름을 포함하여 DynamoDB 암호화 컨텍스트를 만듭니다. `AttributeEncryptor`에서 다른 테이블 이름으로 암호화 컨텍스트를 만들도록 지정하려면 `EncryptionContextOverrideOperator`를 사용합니다.

예를 들어 다음 코드에서는 CMP(암호화 자료 공급자) 및 `DynamoDBEncryptor`의 인스턴스를 만듭니다. 그런 다음 `DynamoDBEncryptor`의 `setEncryptionContextOverrideOperator` 메서드를 호출합니다. 하나의 테이블 이름을 재정의하는 `overrideEncryptionContextTableName` 연산자를 사용합니다. 이 방법으로 구성되면 `AttributeEncryptor`는 `oldTableName` 대신 `newTableName`을 포함하는 DynamoDB 암호화 컨텍스트를 생성합니다. 전체 예제는 EncryptionContextOverridesWithDynamoDBMapper.java를 참조하십시오.

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);

encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionCon
                oldTableName, newTableName));
```

항목을 복호화하고 확인하는 `DynamoDBMapper`의 load 메서드를 호출할 때 원래 테이블 이름을 지정합니다.

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTabl
                .build());
```

여러 테이블 이름을 재정의하는 `overrideEncryptionContextTableNameUsingMap` 연산자를 사용할 수도 있습니다.

테이블 이름 재정의 연산자는 일반적으로 데이터를 복호화하고 서명을 확인할 때 사용됩니다. 그러나 암호화 및 서명 시 DynamoDB 암호화 컨텍스트의 테이블 이름을 다른 값으로 설정하는 데 사용할 수 있습니다.

`DynamoDBEncryptor`를 사용하는 경우 테이블 이름 재정의 연산자를 사용하지 마십시오. 대신 원래 테이블 이름으로 암호화 컨텍스트를 만들고 복호화 메서드에 제출하십시오.

DynamoDB Encryption Client for Java의 예제 코드

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

다음 예제에서는 DynamoDB Encryption Client for Java를 사용하여 애플리케이션에서 DynamoDB 테이블 항목을 보호하는 방법을 보여줍니다. GitHub의 aws-dynamodb-encryption-java 리포지토리의 예제 디렉터리에서 더 많은 예제를 찾고 직접 제공할 수 있습니다.

주제

- DynamoDBEncryptor 사용
- DynamoDBMapper 사용

DynamoDBEncryptor 사용

이 예제에서는 Direct KMS Provider와 함께 하위 수준 DynamoDBEncryptor를 사용하는 방법을 보여줍니다. 다이렉트 KMS 공급자는 사용자가 지정한 AWS KMS key in AWS Key Management Service (AWS KMS)에서 암호화 자료를 생성하고 보호합니다.

`DynamoDBEncryptor`와 함께 호환 가능한 암호화 자료 공급자(CMP)를 사용할 수 있고 `DynamoDBMapper` 및 AttributeEncryptor에서는 Direct KMS Provider를 사용할 수 있습니다.

전체 코드 샘플 보기: AwsKmsEncryptedItem.java

1단계: Direct KMS Provider 생성

지정된 리전으로 AWS KMS 클라이언트의 인스턴스를 생성합니다. 그런 다음 클라이언트 인스턴스를 사용하여 원하는 AWS KMS key로 Direct KMS Provider의 인스턴스를 생성합니다.

이 예제에서는 Amazon 리소스 이름(ARN)을 사용하여를 식별 AWS KMS key하지만 유효한 키 식별자를 사용할 수 있습니다.

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

```
final String region = "us-west-2";

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

2단계: 항목 생성

이 예제는 샘플 테이블 항목을 나타내는 `record` HashMap을 정의합니다.

```
final String partitionKeyName = "partition_attribute";
final String sortKeyName = "sort_attribute";

final Map<String, AttributeValue> record = new HashMap<>();
record.put(partitionKeyName, new AttributeValue().withS("value1"));
record.put(sortKeyName, new AttributeValue().withN("55"));
record.put("example", new AttributeValue().withS("data"));
record.put("numbers", new AttributeValue().withN("99"));
record.put("binary", new AttributeValue().withB(ByteBuffer.wrap(new byte[]{0x00,
 0x01, 0x02})));
record.put("test", new AttributeValue().withS("test-value"));
```

3단계: DynamoDBEncryptor 생성

Direct KMS Provider를 사용하여 `DynamoDBEncryptor` 인스턴스를 생성합니다.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

4단계: DynamoDB 암호화 컨텍스트 생성

DynamoDB 암호화 컨텍스트에는 테이블 구조와 암호화 및 서명 방법에 대한 정보가 포함되어 있습니다. `DynamoDBMapper`를 사용하는 경우 `AttributeEncryptor`에서 자동으로 암호화 컨텍스트를 생성합니다.

```
final String tableName = "testTable";

final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

5단계: 속성 작업 객체 생성

속성 작업은 암호화 및 서명되는 항목 속성, 서명되기만 하는 속성, 암호화 및 서명되지 않는 속성을 결정합니다.

Java에서 속성 작업을 지정하려면 속성 이름 및 `EncryptionFlags` 값 페어로 구성된 HashMap을 생성합니다.

예를 들어, 다음 Java 코드는 서명되었지만 암호화되지 않은 파티션 키 및 정렬 키 속성과 서명되거나 암호화되지 않은 `test` 속성을 제외한 `record` 항목의 모든 속성을 암호화하고 서명하는 `actions` HashMap을 생성합니다.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // fall through to the next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Neither encrypted nor signed
      break;
    default:
      // Encrypt and sign all other attributes
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

6단계: 항목 암호화 및 서명

테이블 항목을 암호화하고 서명하려면 `DynamoDBEncryptor`의 인스턴스에서 `encryptRecord` 방법을 호출합니다. 테이블 항목(`record`), 속성 작업(`actions`) 및 암호화 컨텍스트(`encryptionContext`)를 지정합니다.

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

7단계: DynamoDB 테이블에 항목 넣기

마지막으로 암호화되고 서명된 항목을 DynamoDB 테이블에 넣습니다.

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

DynamoDBMapper 사용

다음 예에서는 Direct KMS Provider와 함께 DynamoDB 매퍼 도우미 클래스를 사용하는 방법을 보여 줍니다. Direct KMS Provider는 사용자가 지정한 AWS Key Management Service (AWS KMS)의 AWS KMS key로 암호화 자료를 생성 및 보호합니다.

`DynamoDBMapper`와 함께 호환 가능한 암호화 자료 공급자(CMP)를 사용할 수 있으며, 하위 수준 `DynamoDBEncryptor`오 함께 Direct KMS Provider를 사용할 수 있습니다.

전체 코드 샘플 보기: AwsKmsEncryptedObject.java

1단계: Direct KMS Provider 생성

지정된 리전으로 AWS KMS 클라이언트의 인스턴스를 생성합니다. 그런 다음 클라이언트 인스턴스를 사용하여 원하는 AWS KMS key로 Direct KMS Provider의 인스턴스를 생성합니다.

이 예제에서는 Amazon 리소스 이름(ARN)을 사용하여를 식별 AWS KMS key하지만 유효한 키 식별자를 사용할 수 있습니다.

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

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

2단계: DynamoDB Encryptor 및 DynamoDBMapper 만들기

이전 단계에서 생성한 Direct KMS Provider를 사용하여 DynamoDB Encryptor의 인스턴스를 생성합니다. DynamoDB Mapper를 사용하려면 하위 수준의 DynamoDB Encryptor를 인스턴스화해야 합니다.

그런 다음 DynamoDB 데이터베이스 인스턴스와 매퍼 구성을 만들고 이를 사용하여 DynamoDB Mapper의 인스턴스를 만듭니다.

> **Important**
>
> DynamoDBMapper를 사용하여 서명된(또는 암호화 및 서명된) 항목을 추가하거나 편집하는 경우 다음 예와 같이 모든 속성을 포함하는 PUT와 같은 저장 동작을 사용하도록 구성합니다. 그렇지 않으면 데이터를 복호화하지 못할 수 있습니다.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp)
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

3단계: DynamoDB 테이블 정의

다음으로 DynamoDB 테이블을 정의합니다. 주석을 사용하여 속성 작업을 지정합니다. 이 예제에서는 DynamoDB 테이블, `ExampleTable` 및 테이블 항목을 나타내는 `DataPoJo` 클래스를 만듭니다.

이 샘플 테이블에서는 기본 키 속성이 서명되지만 암호화되지는 않습니다. 이는 `@DynamoDBHashKey` 주석이 달린 `partition_attribute` 및 `@DynamoDBRangeKey` 주석이 달린 `sort_attribute`에 적용됩니다.

`@DynamoDBAttribute` 주석이 달린 속성(예: `some numbers`)은 암호화 및 서명됩니다. DynamoDB Encryption Client에서 정의한 `@DoNotEncrypt`(기호만 해당) 또는 `@DoNotTouch`(암호화 또는 서명 안 함) 암호화 주석을 사용하는 속성은 예외입니다. 예를 들어 `leave me` 속성에 `@DoNotTouch` 주석이 있으므로 암호화되거나 서명되지 않습니다.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String example;
  private long someNumbers;
  private byte[] someBinary;
  private String leaveMe;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
```

```
    return partitionAttribute;
}

public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
}

@DynamoDBRangeKey(attributeName = "sort_attribute")
public int getSortAttribute() {
    return sortAttribute;
}

public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
}

@DynamoDBAttribute(attributeName = "example")
public String getExample() {
    return example;
}

public void setExample(String example) {
    this.example = example;
}

@DynamoDBAttribute(attributeName = "some numbers")
public long getSomeNumbers() {
    return someNumbers;
}

public void setSomeNumbers(long someNumbers) {
    this.someNumbers = someNumbers;
}

@DynamoDBAttribute(attributeName = "and some binary")
public byte[] getSomeBinary() {
    return someBinary;
}

public void setSomeBinary(byte[] someBinary) {
    this.someBinary = someBinary;
}

@DynamoDBAttribute(attributeName = "leave me")
```

```
  @DoNotTouch
  public String getLeaveMe() {
    return leaveMe;
  }

  public void setLeaveMe(String leaveMe) {
    this.leaveMe = leaveMe;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ", sortAttribute="
        + sortAttribute + ", example=" + example + ", someNumbers=" + someNumbers
        + ", someBinary=" + Arrays.toString(someBinary) + ", leaveMe=" + leaveMe +
 "]";
  }
}
```

4단계: 테이블 항목 암호화 및 저장

이제 테이블 항목을 만들고 DynamoDB Mapper를 사용하여 저장하면 항목이 테이블에 추가되기 전에 자동으로 암호화되고 서명됩니다.

이 예제에서는 `record`라는 테이블 항목을 정의합니다. 테이블에 저장되기 전에 해당 속성은 `DataPoJo` 클래스의 주석을 기반으로 암호화되고 서명됩니다. 이 경우 `PartitionAttribute`, `SortAttribute` 및 `LeaveMe`를 제외한 모든 속성은 암호화되고 서명됩니다. `PartitionAttribute` 및 `SortAttributes`는 서명만 됩니다. `LeaveMe` 속성은 암호화되거나 서명되지 않습니다.

`record` 항목을 암호화하고 서명한 다음 `ExampleTable`에 추가하려면 `DynamoDBMapper` 클래스의 `save` 메서드를 호출합니다. DynamoDB Mapper는 `PUT` 저장 동작을 사용하도록 구성되어 있으므로 항목을 업데이트하는 대신에 동일한 프라이머리 키로 항목을 대체합니다. 이렇게 하면 서명이 일치하고 테이블에서 항목을 가져올 때 해당 항목의 암호를 복호화할 수 있습니다.

```
DataPoJo record = new DataPoJo();
record.setPartitionAttribute("is this");
record.setSortAttribute(55);
record.setExample("data");
record.setSomeNumbers(99);
record.setSomeBinary(new byte[]{0x00, 0x01, 0x02});
record.setLeaveMe("alone");
```

```
mapper.save(record);
```

DynamoDB Encryption Client for Python

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

이 주제에서는 DynamoDB Encryption Client for Pytho를 설치하고 사용하는 방법을 설명합니다. 시작하는 데 도움이 되는 전체 및 테스트된 샘플 코드를 포함하여 GitHub의 aws-dynamodb-encryption-python 리포지토리에서 코드를 찾을 수 있습니다.

> **Note**
>
> DynamoDB Encryption Client for Python의 버전 1.x.x 및 2.x.x는 2022년 7월부터 지원 종료 단계에 있습니다. 가능한 한 빨리 최신 버전으로 업그레이드하세요.

주제

사전 조건

Amazon DynamoDB Encryption Client for Python를 설치하기 전에 다음 사전 조건이 충족되었는지 확인합니다.

지원되는 Python 버전

Amazon DynamoDB Encryption Client for Python 버전 3.3.0 이상에는 Python 3.8 이상이 필요합니다. Python을 다운로드하려면 Python 다운로드를 참조하세요.

이전 버전의 Amazon DynamoDB Encryption Client for Python는 Python 2.7 및 Python 3.4 이상을 지원하지만 최신 버전의 DynamoDB Encryption Client를 사용하는 것이 좋습니다.

Python용 pip 설치 도구

Python 3.6 이상에는 pip가 포함되어 있지만 업그레이드가 필요할 수도 있습니다. pip 업그레이드 또는 설치에 대한 자세한 내용은 pip 설명서의 설치를 참조하세요.

설치

다음 예제와 같이 pip를 사용하여 Amazon DynamoDB Encryption Client for Python를 설치합니다.

최신 버전 설치

```
pip install dynamodb-encryption-sdk
```

pip를 사용하여 패키지를 설치 및 업그레이드하는 방법에 대한 자세한 내용은 패키지 설치를 참조하십시오.

DynamoDB Encryption Client에는 모든 플랫폼에 암호화 라이브러리가 필요합니다. 모든 버전의 pip는 Windows에 암호화 라이브러리를 설치하고 빌드합니다. pip 8.1 이상은 Linux에 암호화를 설치하고 구축합니다. 이전 버전의 pip를 사용 중이고 Linux 환경에 암호화 라이브러리를 빌드하는 데 필요한 도구가 없는 경우 해당 도구를 설치해야 합니다. 자세한 내용은 Linux에서 암호화 빌드를 참조하세요.

GitHub의 aws-dynamodb-encryption-python 리포지토리에서 DynamoDB Encryption Client의 최신 개발 버전을 얻을 수 있습니다.

DynamoDB Encryption Client를 설치한 후 이 가이드의 Python 코드 예제를 살펴보고 시작하세요.

DynamoDB Encryption Client for Python 사용

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

이 주제에서는 다른 프로그래밍 언어 구현에서는 찾을 수 없는 DynamoDB Encryption Client for Python의 일부 기능에 대해 설명합니다. 이러한 기능은 가장 안전한 방법으로 DynamoDB Encryption Client를 더 쉽게 사용할 수 있도록 설계되었습니다. 특별한 사용 사례가 아니라면 이를 사용하는 것이 좋습니다.

DynamoDB Encryption Client를 사용한 프로그래밍에 대한 자세한 내용은 이 가이드의 Python 예제, GitHub의 ws-dynamodb-encryption-pytho 리포지토리의 예제 및 DynamoDB Encryption Client에 대한 Python 설명서를 참조하세요.

주제

- 클라이언트 헬퍼 클래스
- TableInfo 클래스
- Python의 속성 작업

클라이언트 헬퍼 클래스

DynamoDB Encryption Client for Python에는 DynamoDB용 Boto 3 클래스를 미러링하는 여러 클라이언트 헬퍼 클래스가 포함되어 있습니다. 이러한 헬퍼 클래스는 다음과 같이 기존 DynamoDB 애플리케이션에 암호화 및 서명을 더 쉽게 추가하고 가장 일반적인 문제를 방지할 수 있도록 설계되었습니다.

- 기본 키에 대한 재정의 작업을 AttributeActions 객체에 추가하거나 `AttributeActions` 객체가 기본 키를 암호화하도록 클라이언트에 명시적으로 지시하는 경우 예외를 발생시키는 방식으로 항목에서 기본 키를 암호화하지 못하도록 합니다. `AttributeActions` 객체의 기본 작업이 `DO_NOTHING`이면 클라이언트 헬퍼 클래스는 프라이머리 키에 대해 해당 작업을 사용합니다. 그렇지 않으면 `SIGN_ONLY`를 사용합니다.
- TableInfo 객체를 생성하고 DynamoDB 호출을 기반으로 DynamoDB 암호화 컨텍스트를 채웁니다. 이는 DynamoDB 암호화 컨텍스트가 정확하고 클라이언트가 프라이머리 키를 식별할 수 있도록 하는 데 도움이 됩니다.
- 테이블에 쓰거나 읽어올 때 사용자 모르게 테이블 항목을 암호화 및 복호화하는 `put_item` 및 `get_item`와 같은 방법을 지원합니다. `update_item` 메소드만 지원되지 않습니다.

하위 수준의 항목 암호화 도구를 사용하여 직접 상호 작용하는 대신에 클라이언트 헬퍼 클래스를 사용할 수 있습니다. 항목 암호화 도구에서 고급 옵션을 설정해야 하는 경우가 아니면 이 클래스를 사용합니다.

클라이언트 헬퍼 클래스에는 다음이 포함됩니다.

- DynamoDB의 테이블 리소스를 사용하여 한 번에 하나의 테이블을 처리하는 애플리케이션을 위한 EncryptedTable.
- 일괄 처리를 위해 DynamoDB의 서비스 리소스 클래스를 사용하는 애플리케이션을 위한 EncryptedResource.
- DynamoDB에서 하위 수준 클라이언트를 사용하는 애플리케이션을 위한 EncryptedClient.

클라이언트 헬퍼 클래스를 사용하려면 호출자에게 대상 테이블에서 DynamoDB DescribeTable 작업을 호출할 수 있는 권한이 있어야 합니다.

TableInfo 클래스

TableInfo 클래스는 프라이머리 키 및 보조 인덱스에 대한 필드가 포함된 DynamoDB 테이블을 나타내는 헬퍼 클래스입니다. 테이블에 대한 정확한 실시간 정보를 얻는 데 도움이 됩니다.

클라이언트 헬퍼 클래스를 사용하는 경우 이 클래스에서 사용자를 대신하여 `TableInfo` 객체를 만들고 사용합니다. 그렇지 않으면 명시적으로 생성할 수 있습니다. 예제는 항목 암호화 도구 사용 섹션을 참조하세요.

`TableInfo` 객체에 대해 `refresh_indexed_attributes` 방법을 호출하면 DynamoDB DescribeTable 작업을 호출하여 객체의 속성 값이 채워집니다. 테이블 쿼리는 하드 코딩된 인덱스 이름보다 훨씬 더 안정적입니다. `TableInfo` 클래스에는 DynamoDB 암호화 컨텍스트에 필요한 값을 제공하는 `encryption_context_values` 속성도 포함되어 있습니다.

`refresh_indexed_attributes` 방법을 사용하려면 호출자에게 대상 테이블에서 DynamoDB DescribeTable 작업을 호출할 수 있는 권한이 있어야 합니다.

Python의 속성 작업

속성 작업은 항목의 각 속성에 대해 수행할 작업을 항목 암호화 도구에 알려줍니다. Python에서 속성 작업을 지정하려면 기본 작업과 특정 속성에 대한 예외가 포함된 `AttributeActions` 객체를 만듭니다. 유효한 값은 `CryptoAction` 열거 유형에 정의됩니다.

> **Important**
>
> 속성 작업을 사용하여 테이블 항목을 암호화한 후 데이터 모델에서 속성을 추가하거나 제거하면 서명 검증 오류가 발생하여 데이터를 복호화하지 못할 수 있습니다. 자세한 내용은 데이터 모델 변경 단원을 참조하십시오.

```
DO_NOTHING = 0
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

예를 들어, 이 AttributeActions 객체는 모든 속성에 대한 기본값을 ENCRYPT_AND_SIGN으로 설정하고 ISBN 및 PublicationYear 속성에 대한 예외를 지정합니다.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

클라이언트 헬퍼 클래스를 사용하는 경우 기본 키 속성에 대한 속성 작업을 지정할 필요가 없습니다. 클라이언트 헬퍼 클래스는 프라이머리 키를 암호화하는 것을 방지합니다.

클라이언트 헬퍼 클래스를 사용하지 않고 프라이머리 작업이ENCRYPT_AND_SIGN 인 경우 프라이머리 키에 대한 작업을 지정해야 합니다. 프라이머리 키에 대한 권장 조치는 SIGN_ONLY입니다. 이를 쉽게 하려면 프라이머리 키에 SIGN_ONLY를 사용하거나 기본 작업인 경우 DO_NOTHING을 사용하는 set_index_keys 방법을 사용합니다.

> **Warning**
>
> 기본 키 속성은 암호화하지 마십시오. 일반 텍스트로 남겨 두어야 DynamoDB에서 전체 테이블 스캔을 실행하지 않고 해당 항목을 찾을 수 있습니다.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

DynamoDB Encryption Client for Python의 예제 코드

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용

DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

다음 예제에서는 DynamoDB Encryption Client for Python를 사용하여 애플리케이션에서 DynamoDB 데이터를 보호하는 방법을 보여줍니다. GitHub에 있는 aws-dynamodb-encryption-python 리포지토리의 예제 디렉터리에서 더 많은 예제를 찾고 직접 기여할 수 있습니다.

주제

- EncryptedTable 클라이언트 헬퍼 클래스 사용
- 항목 암호화 도구 사용

EncryptedTable 클라이언트 헬퍼 클래스 사용

다음 예제에서는 Direct KMS Provider를 `EncryptedTable` 클라이언트 헬퍼 클래스와 함께 사용하는 방법을 보여줍니다. 이 예제에서는 다음 항목 암호화 도구 사용 예제와 동일한 암호화 자료 공급자를 사용합니다. 그러나 하위 수준 항목 암호화 도구와 직접 상호 작용하는 대신 `EncryptedTable` 클래스를 사용합니다.

이러한 예제를 비교하면 클라이언트 도우미 클래스가 수행하는 작업을 확인할 수 있습니다. 여기에는 DynamoDB 암호화 컨텍스트를 생성하거나 프라이머리 키 속성을 항상 서명하되 절대로 암호화되지 않은 상태로 유지하는 등의 작업이 포함됩니다. 암호화 컨텍스트를 만들고 프라이머리 키를 검색하기 위해 클라이언트 헬퍼 클래스는 DynamoDB DescribeTable 작업을 호출합니다. 이 코드를 실행하려면 이 작업을 호출할 수 있는 권한이 있어야 합니다.

전체 코드 샘플 보기: aws_kms_encrypted_table.py

1단계: 테이블 만들기

테이블 이름을 사용하여 표준 DynamoDB 테이블의 인스턴스를 생성하는 것부터 시작합니다.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

2단계: 암호화 자료 공급자 생성

선택한 암호화 자료 공급자(CMP)의 인스턴스를 생성합니다.

이 예제에서는 Direct KMS Provider를 사용하지만, 사용자는 모든 호환되는 CMP를 사용할 수 있습니다. Direct KMS Provider를 생성하려면 AWS KMS key를 지정합니다. 이 예제에서는의 Amazon 리소스 이름(ARN)을 사용하지 AWS KMS key만 유효한 키 식별자를 사용할 수 있습니다.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

3단계: 속성 작업 객체 생성

속성 작업은 항목의 각 속성에 대해 수행할 작업을 항목 암호화 도구에 알려줍니다. 이 예제의 `AttributeActions` 객체는 무시되는 `test` 속성을 제외한 모든 항목을 암호화하고 서명합니다.

클라이언트 도우미 클래스를 사용할 때 프라이머리 키 속성에 대한 속성 작업을 지정하지 않습니다. `EncryptedTable` 클래스는 프라이머리 키 속성을 서명하지만 암호화하지는 않습니다.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

4단계: 암호화된 테이블 생성

표준 테이블, Direct KMS Provider 및 속성 작업을 사용하여 암호화된 테이블을 생성합니다. 이 단계로 구성이 완료됩니다.

```
encrypted_table = EncryptedTable(
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
)
```

5단계: 테이블에 일반 텍스트 항목 넣기

`encrypted_table`에 대한 `put_item` 방법을 호출하면 테이블 항목이 투명하게 암호화되고 서명되어 DynamoDB 테이블에 추가됩니다.

먼저 테이블 항목을 정의합니다.

```
plaintext_item = {
```

```
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

그런 다음 테이블에 넣습니다.

```
encrypted_table.put_item(Item=plaintext_item)
```

암호화된 형식으로 DynamoDB 테이블에서 항목을 가져오려면 `table` 객체에 대한 `get_item` 방법을 호출합니다. 복호화된 항목을 얻으려면 `encrypted_table` 객체에 대한 `get_item` 메서드를 호출합니다.

항목 암호화 도구 사용

이 예제에서는 테이블 항목을 암호화할 때 항목 암호화 도구와 상호 작용하는 클라이언트 헬퍼 클래스를 사용하는 대신 DynamoDB Encryption Client의 항목 암호화 도구와 직접 상호 작용하는 방법을 보여줍니다.

이 기술을 사용하면 DynamoDB 암호화 컨텍스트와 구성 객체(`CryptoConfig`)를 수동으로 생성합니다. 또한 한 번의 호출로 항목을 암호화하고 별도의 호출로 DynamoDB 테이블에 넣습니다. 이를 통해 `put_item` 호출을 사용자 지정하고 DynamoDB Encryption Client를 사용하여 DynamoDB로 전송되지 않는 구조화된 데이터를 암호화하고 서명할 수 있습니다.

이 예제에서는 Direct KMS Provider를 사용하지만, 사용자는 모든 호환되는 CMP를 사용할 수 있습니다.

전체 코드 샘플 보기: aws_kms_encrypted_item.py

1단계: 테이블 만들기

테이블 이름을 사용하여 표준 DynamoDB 테이블 리소스의 인스턴스를 생성하는 것부터 시작합니다.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

2단계: 암호화 자료 공급자 생성

선택한 암호화 자료 공급자(CMP)의 인스턴스를 생성합니다.

이 예제에서는 Direct KMS Provider를 사용하지만, 사용자는 모든 호환되는 CMP를 사용할 수 있습니다. Direct KMS Provider를 생성하려면 AWS KMS key를 지정합니다. 이 예제에서는의 Amazon 리소스 이름(ARN)을 사용하지 AWS KMS key만 유효한 키 식별자를 사용할 수 있습니다.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

3단계: TableInfo 헬퍼 클래스 사용

DynamoDB에서 테이블에 대한 정보를 얻으려면 TableInfo 헬퍼 클래스의 인스턴스를 생성합니다. 항목 암호화 도구로 직접 작업하는 경우 `TableInfo` 인스턴스를 생성하고 해당 메서드를 호출해야 합니다. 클라이언트 헬퍼 클래스가 사용자를 대신하여 이 작업을 수행합니다.

`TableInfo`의 `refresh_indexed_attributes` 메소드는 DescribeTable DynamoDB 작업을 사용하여 테이블에 대한 정확한 정보를 실시간으로 가져옵니다. 여기에는 프라이머리 키와 로컬 및 글로벌 보조 인덱스가 포함됩니다. 호출자에게 `DescribeTable`을 호출할 수 있는 권한이 있어야 합니다.

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

4단계: DynamoDB 암호화 컨텍스트 생성

DynamoDB 암호화 컨텍스트에는 테이블 구조와 암호화 및 서명 방법에 대한 정보가 포함되어 있습니다. 이 예제에서는 항목 암호화 도구와 상호 작용하므로 DynamoDB 암호화 컨텍스트를 명시적으로 생성합니다. 클라이언트 헬퍼 클래스는 DynamoDB 암호화 컨텍스트를 생성합니다.

TableInfo 헬퍼 클래스의 속성을 사용하여 파티션 키와 정렬 키를 가져올 수 있습니다.

```
index_key = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
}

encryption_context = EncryptionContext(
    table_name=table_name,
```

```
    partition_key_name=table_info.primary_index.partition,
    sort_key_name=table_info.primary_index.sort,
    attributes=dict_to_ddb(index_key)
)
```

5단계: 속성 작업 객체 생성

속성 작업은 항목의 각 속성에 대해 수행할 작업을 항목 암호화 도구에 알려줍니다. 이 예제의 `AttributeActions` 객체는 서명되었지만 암호화되지 않은 프라이머리 키 속성과 무시되는 `test` 속성을 제외한 모든 항목을 암호화하고 서명합니다.

항목 암호화 도구와 직접 상호작용하고 기본 작업이 `ENCRYPT_AND_SIGN`인 경우 프라이머리 키에 대한 대체 작업을 지정해야 합니다. 프라이머리 키에 대한 `SIGN_ONLY`을 사용하거나 기본 작업인 경우 `DO_NOTHING`을 사용하는 `set_index_keys` 메서드를 사용할 수 있습니다.

프라이머리 키를 지정하기 위해 이 예제에서는 DynamoDB 호출로 채워지는 TableInfo 객체의 인덱스 키를 사용합니다. 이 기술은 프라이머리 키 이름을 하드 코딩하는 것보다 안전합니다.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
actions.set_index_keys(*table_info.protected_index_keys())
```

6단계: 항목에 대한 구성 만들기

DynamoDB Encryption Client를 구성하려면 테이블 항목에 대한 CryptoConfig 구성에서 방금 생성한 객체를 사용합니다. 클라이언트 헬퍼 클래스는 사용자를 위해 CryptoConfig를 만듭니다.

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
)
```

7단계: 항목 암호화

이 단계에서는 항목을 암호화하고 서명하지만 DynamoDB 테이블에 저장하지는 않습니다.

클라이언트 헬퍼 클래스를 사용할 경우 항목이 사용자 모르게 암호화 및 서명된 다음, 사용자가 헬퍼 클래스의 `put_item` 메서드를 호출할 때 DynamoDB 테이블에 추가됩니다. 항목 암호화 도구를 직접 사용하는 경우 암호화 및 입력 작업은 독립적입니다.

먼저 일반 텍스트 항목을 만듭니다.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

그런 다음 암호화하고 서명합니다. `encrypt_python_item` 방법에는 `CryptoConfig` 구성 객체가 필요합니다.

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

8단계: 테이블에 항목 넣기

이 단계에서는 암호화되고 서명된 항목을 DynamoDB 테이블에 넣습니다.

```
table.put_item(Item=encrypted_item)
```

암호화된 항목을 보려면 `encrypted_table` 객체 대신 원본 `table` 객체에 대한 `get_item` 방법을 호출합니다. 항목을 확인 및 복호화하지 않고 DynamoDB 테이블에서 항목을 가져옵니다.

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

다음 이미지는 암호화되고 서명된 테이블 항목 예제의 일부를 보여줍니다.

암호화된 속성 값은 이진 데이터입니다. 프라이머리 키 속성(`partition_attribute` 및 `sort_attribute`)의 이름과 값 및 `test` 속성은 일반 텍스트로 유지됩니다. 이 출력은 서명(`*amzn-ddb-map-sig*`)을 포함하는 속성과 자료 설명 속성(`*amzn-ddb-map-desc*`)을 보여줍니다.

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-alg\
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydmoJD
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHmacS
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eHmac
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf\xe
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x1e\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\xf7\
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```

데이터 모델 변경

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용 DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

항목을 암호화하거나 복호화할 때마다 암호화하고 서명할 속성, 서명할 속성(암호화하지 않음), 무시할 속성을 DynamoDB Encryption Client에 알려주는 속성 작업을 제공해야 합니다. 속성 작업은 암호화된 항목에 저장되지 않으며 DynamoDB Encryption Client는 속성 작업을 자동으로 업데이트하지 않습니다.

> **Important**
>
> DynamoDB Encryption Client는 암호화되지 않은 기존 DynamoDB 테이블 데이터의 암호화를 지원하지 않습니다.

데이터 모델을 변경할 때마다, 즉 테이블 항목에서 속성을 추가하거나 제거할 때 오류가 발생할 수 있습니다. 지정한 속성 작업이 항목의 모든 속성을 고려하지 않는 경우 항목이 의도한 방식으로 암호화되고 서명되지 않을 수 있습니다. 특히, 항목 복호화 시 제공하는 속성 작업과 항목 암호화 시 제공했던 속성 작업이 다른 경우 서명 확인이 실패할 수 있습니다.

예를 들어, 항목을 암호화하는 데 사용된 속성 동작에서 `test` 속성에 서명하도록 지정하는 경우 항목의 서명에는 `test` 속성이 포함됩니다. 그러나 항목을 복호화하는 데 사용된 속성 작업이 `test` 속성을 고려하지 않는 경우 클라이언트가 `test` 속성을 포함하지 않는 서명을 확인하려고 하기 때문에 확인이 실패합니다.

이는 DynamoDB Encryption Client가 모든 애플리케이션의 항목에 대해 동일한 서명을 계산해야 하기 때문에 여러 애플리케이션이 동일한 DynamoDB 항목을 읽고 쓸 때 특히 문제가 됩니다. 또한 속성 작업의 변경 사항이 모든 호스트에 전파되어야 하기 때문에 분산 애플리케이션에서도 문제가 됩니다. 한 프로세스에서 한 호스트가 DynamoDB 테이블에 액세스하는 경우에도 모범 사례 프로세스를 설정하면 프로젝트가 더욱 복잡해지더라도 오류를 방지하는 데 도움이 됩니다.

테이블 항목을 읽을 수 없도록 하는 서명 유효성 검사 오류를 방지하려면 다음 지침을 따르십시오.

- 속성 추가 - 새 속성이 속성 작업을 변경하는 경우 항목에 새 속성을 포함하기 전에 속성 작업 변경을 완전히 배포합니다.
- 속성 제거 - 항목에서 속성 사용을 중지하는 경우 속성 작업을 변경하지 마십시오.
- 작업 변경 - 속성 작업 구성을 사용하여 테이블 항목을 암호화한 후에는 테이블의 모든 항목을 다시 암호화하지 않고는 기존 속성에 대한 기본 작업이나 작업을 안전하게 변경할 수 없습니다.

서명 유효성 검사 오류는 해결하기가 매우 어려울 수 있으므로 가장 좋은 방법은 오류를 방지하는 것입니다.

주제

속성 추가

테이블 항목에 새 속성을 추가할 때 속성 작업을 변경해야 할 수 있습니다. 서명 유효성 검사 오류를 방지하려면 2단계 프로세스로 이 변경을 구현하는 것이 좋습니다. 첫 번째 단계가 완료되었는지 확인한 후 두 번째 단계를 시작합니다.

1. 테이블을 읽거나 쓰는 모든 애플리케이션에서 속성 작업을 변경합니다. 이러한 변경 사항을 배포하고 업데이트가 모든 대상 호스트에 전파되었는지 확인합니다.
2. 테이블 항목의 새 속성에 값을 씁니다.

이 2단계 접근 방식은 모든 애플리케이션과 호스트에 동일한 속성 작업이 있도록 하며, 새 속성이 발생하기 전에 동일한 서명을 계산합니다. 일부 암호화의 기본값은 암호화 및 서명이기 때문에 속성에 대한 작업이 아무 작업 안 함(암호화 또는 서명 안 함)인 경우에도 중요합니다.

다음 예제에서는 이 프로세스의 첫 번째 단계에 대한 코드를 보여 줍니다. 다른 테이블 항목에 대한 링크를 저장하는 새 항목 속성인 `link`를 추가합니다. 이 링크는 일반 텍스트로 유지되어야 하므로 이 예제에서는 서명 전용 작업을 할당합니다. 이 변경 사항을 완전히 배포한 다음 모든 애플리케이션과 호스트에 새 속성 작업이 있는지 확인한 후 테이블 항목에서 `link` 속성을 사용할 수 있습니다.

Java DynamoDB Mapper

DynamoDB Mapper 및 `AttributeEncryptor`를 사용하는 경우 기본적으로 기본 키(서명되지만 암호화되지 않음)를 제외하고는 모든 속성이 암호화 및 서명됩니다. 서명 전용 작업을 지정하려면 `@DoNotEncrypt` 주석을 사용합니다.

이 예제에서는 새 `link` 속성에 대한 `@DoNotEncrypt` 주석을 사용합니다.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String link;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "link")
  @DoNotEncrypt
```

```
  public String getLink() {
    return link;
  }

  public void setLink(String link) {
    this.link = link;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ",
        sortAttribute=" + sortAttribute + ",
        link=" + link + "]";
  }
}
```

Java DynamoDB encryptor

하위 수준 DynamoDB 암호화 도구에서 각 속성에 대한 작업을 설정해야 합니다. 이 예제에서는 기본값이 `encryptAndSign`인 switch 문을 사용하며, 파티션 키, 정렬 키 및 새 `link` 속성에 대해서는 예외가 지정됩니다. 이 예제에서 링크 속성 코드가 사용되기 전에 완전히 배포되지 않은 경우 링크 속성은 일부 애플리케이션에서 암호화되고 서명되지만 다른 애플리케이션에서는 서명만 됩니다.

```
for (final String attributeName : record.keySet()) {
    switch (attributeName) {
        case partitionKeyName:
            // fall through to the next case
        case sortKeyName:
            // partition and sort keys must be signed, but not encrypted
            actions.put(attributeName, signOnly);
            break;
        case "link":
            // only signed
            actions.put(attributeName, signOnly);
            break;
        default:
            // Encrypt and sign all other attributes
            actions.put(attributeName, encryptAndSign);
            break;
    }
}
```

Python

DynamoDB Encryption Client for Python에서는 모든 속성에 대한 기본 작업을 지정한 다음 예외를 지정할 수 있습니다.

Python 클라이언트 헬퍼 클래스를 사용하는 경우 기본 키 속성에 대한 속성 작업을 지정할 필요가 없습니다. 클라이언트 헬퍼 클래스는 프라이머리 키를 암호화하는 것을 방지합니다. 그러나 클라이언트 도우미 클래스를 사용하지 않는 경우 파티션 키 및 정렬 키에 대해 SIGN_ONLY 작업을 설정해야 합니다. 실수로 파티션 또는 정렬 키를 암호화한 경우 전체 테이블 스캔 없이는 데이터를 복구할 수 없습니다.

이 예제에서는 `SIGN_ONLY` 작업을 가져오는 새 `link` 속성에 대한 예외를 지정합니다.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

속성 제거

DynamoDB Encryption Client로 암호화된 항목에 더 이상 속성이 필요하지 않은 경우 속성 사용을 중지할 수 있습니다. 그러나 해당 속성에 대한 작업을 삭제하거나 변경하지 마십시오. 그런 경우 해당 속성을 가진 항목이 발견되면 해당 항목에 대해 계산된 서명이 원래 서명과 일치하지 않으므로 서명 유효성 검사가 실패합니다.

코드에서 속성의 모든 추적을 제거하고 싶을 수도 있지만 항목을 삭제하는 대신 더 이상 사용되지 않는다는 설명을 추가합니다. 전체 테이블 스캔을 수행하여 속성의 모든 인스턴스를 삭제하더라도 해당 속성을 가진 암호화된 항목이 캐싱되거나 구성 어딘가에서 처리 중일 수 있습니다.

DynamoDB Encryption Client 애플리케이션의 문제 해결

> **Note**
>
> 클라이언트측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 다음 주제에서는 Java용 DynamoDB Encryption Client 버전 1.x~2.x 와 Python용

DynamoDB Encryption Client 버전 1.x~3.x에 대한 정보를 제공합니다. 자세한 내용은 AWS Database Encryption SDK for DynamoDB 버전 지원을 참조하세요.

이 섹션에서는 DynamoDB Encryption Client를 사용할 때 발생할 수 있는 문제를 설명하고 문제 해결을 위한 제안을 제공합니다.

DynamoDB Encryption Client에 대한 피드백을 제공하려면 aws-dynamodb-encryption-java 또는 aws-dynamodb-encryption-python GitHub 리포지토리에 문제를 제출하세요.

이 설명서에 대한 피드백을 제공하려면 이 페이지의 피드백 링크를 사용하십시오.

주제

- 액세스 거부됨
- 서명 확인 실패
- 이전 버전 글로벌 테이블 관련 문제
- Most Recent Provider의 성능 저하

액세스 거부됨

문제: 필요한 리소스에 대한 애플리케이션의 액세스가 거부됩니다.

제안: 필요한 권한에 대해 알아보고 이러한 권한을 애플리케이션이 실행되는 보안 컨텍스트에 추가합니다.

세부 정보

DynamoDB Encryption Client 라이브러리를 사용하는 애플리케이션을 실행하려면, 호출자에게 해당 구성 요소를 사용할 수 있는 권한이 있어야 합니다. 그렇지 않으면 필수 요소에 대한 액세스가 거부됩니다.

- DynamoDB Encryption Client에는 Amazon Web Services(AWS) 계정이 필요하지 않으며 어떤 AWS 서비스에도 의존하지 않습니다. 그러나 애플리케이션에서 를 AWS 계정 사용하는 경우 계정을 사용할 권한이 있는 및 사용자가 AWS필요합니다.
- DynamoDB Encryption Client에는 Amazon DynamoDB가 필요하지 않습니다. 그러나 클라이언트를 사용하는 애플리케이션이 DynamoDB 테이블을 생성하거나, 테이블에 항목을 넣거나, 테이블에서 항목을 가져오는 경우 호출자는 AWS 계정에 필요한 DynamoDB 작업을 사용할 수 있는 권한이 있어야 합니다. 자세한 내용은 Amazon DynamoDB 개발자 안내서의 액세스 제어 주제를 참조하세요.

- 애플리케이션이 DynamoDB Encryption Client for Python의 클라이언트 헬퍼 클래스를 사용하는 경우 호출자는 DynamoDB DescribeTable 작업을 호출할 권한이 있어야 합니다.
- DynamoDB Encryption Client에는 AWS Key Management Service ()가 필요하지 않습니다AWS KMS. 그러나 애플리케이션이 Direct KMS Materials Provider를 사용하거나가 사용하는 공급자 스토어와 함께 Most Recent Provider를 사용하는 경우 AWS KMS호출자는 AWS KMS GenerateDataKey 및 Decrypt 작업을 사용할 수 있는 권한이 있어야 합니다.

서명 확인 실패

문제: 서명 확인 실패로 인해 항목을 복호화할 수 없습니다. 또한 항목이 의도한 대로 암호화 및 서명되지 않을 수도 있습니다.

제안: 제공하는 속성 작업에서 항목의 모든 속성을 고려해야 합니다. 항목을 복호화할 때는 항목을 암호화하는 데 사용된 작업과 일치하는 속성 작업을 제공해야 합니다.

세부 정보

제공하는 속성 작업은 암호화하고 서명할 속성, 서명할(암호화하지 않음) 속성, 무시할 속성을 DynamoDB Encryption Client에 알려줍니다.

지정한 속성 작업이 항목의 모든 속성을 고려하지 않는 경우 항목이 의도한 방식으로 암호화되고 서명되지 않을 수 있습니다. 항목 복호화 시 제공하는 속성 작업과 항목 암호화 시 제공했던 속성 작업이 다른 경우 서명 확인이 실패할 수 있습니다. 이는 새 속성 동작이 모든 호스트에 전파되지 않았을 수 있는 분산 애플리케이션에서 특히 발생하는 문제입니다.

서명 유효성 검사 오류는 해결하기 어렵습니다. 이를 방지하기 위해 데이터 모델을 변경할 때 추가 예방 조치를 취하십시오. 자세한 내용은 데이터 모델 변경을 참조하세요.

이전 버전 글로벌 테이블 관련 문제

문제: 서명 확인에 실패하여 이전 버전의 Amazon DynamoDB 글로벌 테이블의 항목을 복호화할 수 없습니다.

제안: 예약된 복제 필드가 암호화되거나 서명되지 않도록 속성 작업을 설정합니다.

세부 정보

DynamoDB Encryption Client를 DynamoDB 글로벌 테이블과 함께 사용할 수 있습니다. 다중 리전 KMS 키와 함께 글로벌 테이블을 사용하고 글로벌 테이블이 복제되는 모든 AWS 리전 에 KMS 키를 복제하는 것이 좋습니다.

글로벌 테이블 버전 2019.11.21부터 DynamoDB Encryption Client에서 특별한 구성 없이 글로벌 테이블을 사용할 수 있습니다. 하지만 글로벌 테이블 버전 2017.11.29를 사용하는 경우 예약된 복제 필드가 암호화되거나 서명되지 않았는지 확인해야 합니다.

글로벌 테이블 버전 2017.11.29를 사용하는 경우 다음 속성에 대한 속성 작업을 Java의 `DO_NOTHING` 또는 Python의 `@DoNotTouch`으로 설정해야 합니다.

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

다른 버전의 글로벌 테이블을 사용하는 경우에는 별도의 조치가 필요하지 않습니다.

Most Recent Provider의 성능 저하

문제: 특히 최신 버전의 DynamoDB Encryption Client로 업데이트한 후 애플리케이션의 응답성이 떨어집니다.

제안: Time-to-Live 값과 캐시 크기를 조정합니다.

세부 정보

Most Recent Provider는 암호화 자료의 재사용을 제한적으로 허용하여 DynamoDB Encryption Client를 사용하는 애플리케이션의 성능을 개선하도록 설계되었습니다. 애플리케이션에 Most Recent Provider를 구성할 때는 성능 향상과 캐싱 및 재사용으로 인해 발생하는 보안 문제 사이에서 균형을 맞춰야 합니다.

최신 버전의 DynamoDB Encryption Client에서는 time-to-live(TTL) 값에 따라 캐시된 암호화 자료 공급자(CMP)를 사용할 수 있는 기간이 결정됩니다. 또한 TTL은 Most Recent Provider가 CMP의 새 버전을 확인하는 빈도를 결정합니다.

TTL이 너무 길면 애플리케이션이 비즈니스 규칙이나 보안 표준을 위반할 수 있습니다. TTL이 너무 짧은 경우 공급자 스토어를 자주 호출하면 공급자 스토어가 애플리케이션 및 서비스 계정을 공유하는 기타 애플리케이션의 요청을 제한할 수 있습니다. 이 문제를 해결하려면 지연 시간 및 가용성 목표를 충족하고 보안 표준을 준수하는 값으로 TTL 및 캐시 크기를 조정합니다. 자세한 내용은 time-to-live 값 설정 섹션을 참조하십시오.

Amazon DynamoDB Encryption Client 이름 변경

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

2023년 6월 9일에 클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. AWS Database Encryption SDK는 Amazon DynamoDB와 호환됩니다. 기존 DynamoDB Encryption Client에서 암호화된 항목을 복호화하고 읽을 수 있습니다. 기존 DynamoDB Encryption Client 버전에 대한 자세한 내용은 AWS DynamoDB용 Database Encryption SDK 버전 지원 섹션을 참조하세요.

AWS Database Encryption SDK는 Java용 DynamoDB Encryption Client의 주요 재작성인 DynamoDB용 Java 클라이언트 측 암호화 라이브러리 버전 3.x를 제공합니다. 여기에는 새로운 구조화된 데이터 형식, 향상된 멀티테넌시 지원, 원활한 스키마 변경, 검색 가능한 암호화 지원 등 많은 업데이트가 포함되어 있습니다.

AWS Database Encryption SDK에 도입된 새로운 기능에 대해 자세히 알아보려면 다음 주제를 참조하세요.

검색 가능한 암호화

전체 데이터베이스를 복호화하지 않고도 암호화된 레코드를 검색할 수 있는 데이터베이스를 설계할 수 있습니다. 위협 모델 및 쿼리 요구 사항에 따라 검색 가능한 암호화를 사용하여 암호화된 레코드에 대해 정확한 일치 검색 또는 보다 사용자 정의된 복잡한 쿼리를 수행할 수 있습니다.

키링

AWS Database Encryption SDK는 키링을 사용하여 봉투 암호화를 수행합니다. 키링은 레코드를 보호하는 데이터 키를 생성, 암호화 및 복호화합니다. AWS Database Encryption SDK는 대칭 암호화 또는 비대칭 RSAAWS KMS keys를 사용하여 데이터 키를 보호하는 AWS KMS 키링과 레코드를 암호화하거나 해독할 AWS KMS 때마다를 호출하지 않고도 대칭 암호화 KMS 키로 암호화 자료를 보호할 수 있는 AWS KMS 계층적 키링을 지원합니다. 원시 AES 키링 및 원시 RSA 키링을 사용하여 자체 키 자료를 지정할 수도 있습니다.

원활한 스키마 변경

AWS Database Encryption SDK를 구성할 때 암호화 및 서명할 필드, 서명할 필드(암호화하지 않음), 무시할 필드를 클라이언트에 알려주는 암호화 작업을 제공합니다. AWS Database Encryption

SDK를 사용하여 레코드를 보호한 후에도 데이터 모델을 변경할 수 있습니다. 단일 배포에서 암호화된 필드 추가 또는 제거와 같은 암호화 작업을 업데이트할 수 있습니다.

클라이언트측 암호화를 위한 기존 DynamoDB 테이블 구성

기존 버전의 DynamoDB Encryption Client는 채워지지 않은 새 테이블에 구현되도록 설계되었습니다. AWS Database Encryption SDK for DynamoDB를 사용하면 기존 Amazon DynamoDB 테이블을 DynamoDB용 Java 클라이언트 측 암호화 라이브러리 버전 3.x로 마이그레이션할 수 있습니다.

레퍼런스

클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다. 이 개발자 안내서는 여전히 DynamoDB Encryption Client에 대한 정보를 제공합니다.

다음 주제에서는 AWS Database Encryption SDK에 대한 기술 세부 정보를 제공합니다.

자료 설명 형식

자료 설명은 암호화된 레코드의 헤더 역할을 합니다. AWS Database Encryption SDK를 사용하여 필드를 암호화하고 서명하면 암호화 도구는 암호화 자료를 수집할 때 자료 설명을 기록하고 암호화 도구가 레코드에 추가하는 새 필드(aws_dbe_head)에 자료 설명을 저장합니다. 자료 설명은 암호화된 데이터 키와 레코드가 암호화되고 서명된 방법에 대한 정보를 포함하는 이식 가능한 형식의 데이터 구조입니다. 다음 테이블에서는 재료 설명을 구성하는 값을 설명합니다. 표시된 순서대로 바이트가 추가됩니다.

값	길이(바이트)
Version	1
Signatures Enabled	1
Record ID	32
Encrypt Legend	변수
Encryption Context Length	2
???	변수
Encrypted Data Key Count	1
Encrypted Data Keys	변수
Record Commitment	1

버전

이 aws_dbe_head 필드 형식의 버전입니다.

서명 활성화됨

이 레코드에 대해 ECDSA 디지털 서명이 활성화되었는지 여부를 인코딩합니다.

바이트 값	의미
0x01	ECDSA 디지털 서명 활성화됨(기본값)
0x00	ECDSA 디지털 서명 비활성화됨

레코드 ID

레코드를 식별하는 무작위로 생성된 256비트 값입니다. 레코드 ID:

- 암호화된 레코드를 고유하게 식별합니다.
- 자료 설명을 암호화된 레코드에 바인딩합니다.

범례 암호화

인증된 필드가 암호화되고 연재된 설명입니다. 암호화 범례는 복호화 메서드가 복호화를 시도해야 하는 필드를 결정하는 데 사용됩니다.

바이트 값	의미
0x65	ENCRYPT_AND_SIGN
0x73	SIGN_ONLY

암호화 범례는 다음과 같이 연재됩니다.

1. 표준 경로를 나타내는 바이트 시퀀스를 기준으로 사전순으로 표시됩니다.
2. 각 필드에 대해 위에 지정된 바이트 값 중 하나를 순서대로 추가하여 해당 필드를 암호화해야 하는지 여부를 나타냅니다.

암호화 컨텍스트 길이

암호화 컨텍스트의 길이입니다. 이 값은 부호 없는 16비트 정수로 해석되는 2바이트 값입니다. 최대 길이는 65,535바이트입니다.

암호화 컨텍스트

비밀이 아닌 임의의 추가 인증 데이터를 포함하는 이름-값 페어 세트입니다.

ECDSA 디지털 서명이 활성화되면 암호화 컨텍스트에 키-값 페어가 포함됩니다`{"aws-crypto-footer-ecdsa-key": Qtxt}`.는 SEC 1 버전 2.0에 따라 `Q` 압축된 후 base64로 인코딩된 타원 곡선 지점을 `Qtxt` 나타냅니다.

암호화된 데이터 키 수

암호화된 데이터 키의 수입니다. 암호화된 데이터 키의 수를 지정하는 무부호 8비트 정수로 해석되는 1바이트 값입니다. 각 레코드의 암호화된 데이터 키의 최대 수는 255개입니다.

암호화된 데이터 키

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
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 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 제공자가 암호화한 데이터 키입니다.

레코드 커밋

커밋 키를 사용하여 이전의 모든 자료 설명 바이트에 대해 계산된 고유한 256비트 해시 기반 메시지 인증 코드(HMAC) 해시입니다.

AWS KMS 계층적 키링 기술 세부 정보

AWS KMS 계층적 키링은 고유 데이터 키를 사용하여 각 필드를 암호화하고, 활성 브랜치 키에서 파생된 고유 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 Database Encryption SDK 개발자 안내서의 문서 기록

아래 표에 이 설명서의 주요 변경 사항이 설명되어 있습니다. Amazon은 이러한 주요 변경 사항 외에도 설명과 예제를 개선하고 고객이 제공한 피드백을 반영하도록 설명서를 자주 업데이트하고 있습니다. 중요한 변경 사항에 대해 알림을 받으려면 RSS 피드를 구독합니다.

변경 사항	설명	날짜
새로운 특성	AWS KMS ECDH 키링 및 원시 ECDH 키링에 대한 설명서가 추가되었습니다.	2024년 6월 17일
정식 출시(GA) 릴리스	DynamoDB용 .NET 클라이언트 측 암호화 라이브러리에 대한 지원을 소개합니다.	2024년 1월 17일
정식 출시(GA) 릴리스	DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x의 GA 릴리스에 대한 설명서가 업데이트되었습니다. **Warning** 개발자 시험판 릴리스 중에 생성된 브랜치 키는 더 이상 지원되지 않습니다.	2023년 7월 24일
DynamoDB Encryption Client의 브랜드 변경	클라이언트 측 암호화 라이브러리의 이름이 AWS Database Encryption SDK로 변경되었습니다.	2023년 6월 9일
미리 보기 릴리스	새로운 구조화된 데이터 형식, 향상된 멀티테넌시 지원, 원활한 스키마 변경 및 검색 가능한 암호화 지원을 포함하는	2023년 6월 9일

	DynamoDB용 Java 클라이언트측 암호화 라이브러리 버전 3.x에 대한 설명서가 추가 및 업데이트되었습니다.	
문서 변경	고객 마스터 키(CMK)라는 AWS Key Management Service 용어를 AWS KMS key 및 KMS 키로 바꿉니다.	2021년 8월 30일
새로운 기능	AWS Key Management Service (AWS KMS) 다중 리전 키에 대한 지원이 추가되었습니다. 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
새 예제	Java에서 DynamoDBMapper를 사용하는 예제를 추가했습니다.	2018년 9월 6일
Python 지원	Java뿐만 아니라 Python에 대한 지원을 추가했습니다.	2018년 5월 2일
최초 릴리스	이 설명서의 최초 릴리스입니다.	2018년 5월 2일

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.