デベロッパーガイド

AWS データベース暗号化 SDK

AWS データベース暗号化 SDK: デベロッパーガイド

Table of Contents

AWS Database Encryption SDK とは

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK は、データベース設計にクライアント側の暗号化を含めることができる一連のソフトウェアライブラリです。 AWS Database Encryption SDK は、レコードレベルの暗号化ソリューションを提供します。どのフィールドを暗号化し、データの真正性を保証する署名にどのフィールドを含めるかを指定します。伝送中および保管時の機密データを暗号化することで、AWSなどのサードパーティーがお客様のプレーンテキストデータを使用することはできません。AWS Database Encryption SDK は、Apache 2.0 ライセンスに基づいて、無償で提供されています。

このデベロッパーガイドでは、 AWS Database Encryption SDK の概念的な概要を説明します。これには、アーキテクチャの概要、データを保護する方法の詳細、サーバー側の暗号化との違い、使用開始に役立つアプリケーションの重要なコンポーネントの選択に関するガイダンスが含まれます。

AWS Database Encryption SDK は、属性レベルの暗号化で Amazon DynamoDB をサポートします。

AWS Database Encryption SDK には次の利点があります。

データベースアプリケーション向けに特別に設計

AWS Database Encryption SDK を使用するには、暗号化の専門家である必要はありません。この実装には、既存のアプリケーションで動作するように設計されたヘルパーメソッドが含まれます。

必要なコンポーネントを作成して設定すると、暗号化クライアントは、データベースへの追加時にレコードを透過的に暗号化して署名し、取得時に検証および復号します。

セキュアな暗号化と署名を含む

AWS Database Encryption SDK には、一意のデータ暗号化キーを使用して各レコードのフィールド値を暗号化し、フィールドの追加や削除、暗号化された値のスワップなどの不正な変更から保護するためにレコードに署名する安全な実装が含まれています。

ソースの暗号化マテリアルを使用する

AWS Database Encryption SDK は、キーリングを使用して、レコードを保護する一意のデータ暗号化キーを生成、暗号化、復号します。キーリングは、そのデータキーを暗号化するラッピングキーを決定します。

AWS Key Management Service (AWS KMS) や AWS CloudHSM などの暗号化サービスを含む、任意のソースからのラッピングキーを使用できます。 AWS Database Encryption SDK には、AWS アカウント や AWS のサービスは必要ありません。

暗号マテリアルのキャッシュのサポート

AWS KMS 階層キーリングは、Amazon DynamoDB テーブルに保持されている AWS KMS 保護されたブランチキーを使用して AWS KMS 呼び出しの数を減らし、暗号化および復号オペレーションで使用されるブランチキーマテリアルをローカルにキャッシュする暗号化マテリアルキャッシュソリューションです。これにより、レコードを暗号化または復号する AWS KMS たびに を呼び出すことなく、対称暗号化 KMS キーで暗号化マテリアルを保護できます。 AWS KMS 階層キーリングは、呼び出しを最小限に抑える必要があるアプリケーションに適しています AWS KMS。

検索可能な暗号化

データベース全体を復号せずに、暗号化されたレコードを検索できるデータベースを設計できます。脅威モデルとクエリ要件に応じて、検索可能な暗号化を使用して、暗号化されたデータベースに対して完全一致検索やよりカスタマイズされた複雑なクエリを実行できます。

マルチテナンシーデータベーススキーマのサポート

AWS Database Encryption SDK を使用すると、各テナントを個別の暗号化マテリアルで分離することで、共有スキーマを持つデータベースに保存されているデータを保護できます。データベース内で複数のユーザーが暗号化オペレーションを実行している場合は、いずれかの AWS KMS キーリングを使用して、暗号化オペレーションで使用する個別のキーを各ユーザーに提供します。詳細については、「マルチテナンシーデータベースの使用」を参照してください。

シームレスなスキーマ更新のサポート

AWS Database Encryption SDK を設定するときは、暗号化および署名するフィールド、署名するフィールド (暗号化しない) 、無視するフィールドをクライアントに通知する暗号化アクションを提供します。 AWS Database Encryption SDK を使用してレコードを保護した後でも、データモデルを変更できます。暗号化されたフィールドの追加や削除などの暗号化アクションを単一のデプロイで更新できます。

オープンソースリポジトリで開発

AWS Database Encryption SDK は、GitHub のオープンソースリポジトリで開発されています。これらのリポジトリを使用して、コードを表示したり、問題を読んで送信したりできるほか、実装に固有の情報を検索することもできます。

AWS Database Encryption SDK for DynamoDB

- GitHub の aws-database-encryption-sdk-dynamodb リポジトリは、Java、.NET、Rust の AWS Database Encryption SDK for DynamoDB の最新バージョンをサポートしています。

  AWS Database Encryption SDK for DynamoDB は、Dafny の製品です。Dafny は、仕様、実装するコード、およびテストするための証明を記述する検証対応言語です。その結果、機能の正確性を保証するフレームワークで AWS Database Encryption SDK for DynamoDB の機能を実装するライブラリが作成されます。

サポートとメンテナンス

AWS Database Encryption SDK は、バージョニングフェーズやライフサイクルフェーズなど、 AWS SDK とツールが使用するのと同じメンテナンスポリシーを使用します。ベストプラクティスとして、データベースの実装には AWS Database Encryption SDK の利用可能な最新バージョンを使用し、新しいバージョンがリリースされたらアップグレードすることをお勧めします。

詳細については、AWS SDKs とツールリファレンスガイド」の「SDK とツールのメンテナンスポリシー AWS SDKs 」を参照してください。

フィードバックを送る

当社では、お客様からのフィードバックをお待ちしております。質問、コメント、ご報告いただく問題がある場合は、以下のリソースをご利用ください。

AWS Database Encryption SDK で潜在的なセキュリティ脆弱性を発見した場合は、AWS セキュリティに通知してください。GitHub で公開されている問題はご報告いただく必要はありません。

このドキュメントに関するフィードバックを提供するには、任意のページのフィードバックリンクを使用します。

AWS Database Encryption SDK の概念

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

このトピックでは、 AWS Database Encryption SDK で使用される概念と用語について説明します。

AWS Database Encryption SDK のコンポーネントがどのように相互作用するかについては、「」を参照してくださいAWS Database Encryption SDK の仕組み。

AWS Database Encryption SDK の詳細については、以下のトピックを参照してください。

- AWS Database Encryption SDK がエンベロープ暗号化を使用してデータを保護する方法について説明します。
- エンベロープ暗号化の要素、レコードを保護するデータキーおよびデータキーを保護するラッピングキーについての説明。
- どのラッピングキーを使用するかを決めるキーリングについての説明。
- 暗号化プロセスの整合性を向上させる暗号化コンテキストについての説明。
- 暗号化メソッドがレコードに追加するマテリアルの説明について説明します。
- どのフィールドを暗号化して署名するかを AWS Database Encryption SDK に指示する暗号化アクションについて説明します。

トピック

- エンベロープ暗号化
- データキー
- ラッピングキー
- キーリング
- 暗号化アクション
- マテリアル記述
- 暗号化コンテキスト
- 暗号化マテリアルマネージャー
- 対称暗号化と非対称暗号化

- キーコミットメント
- デジタル署名

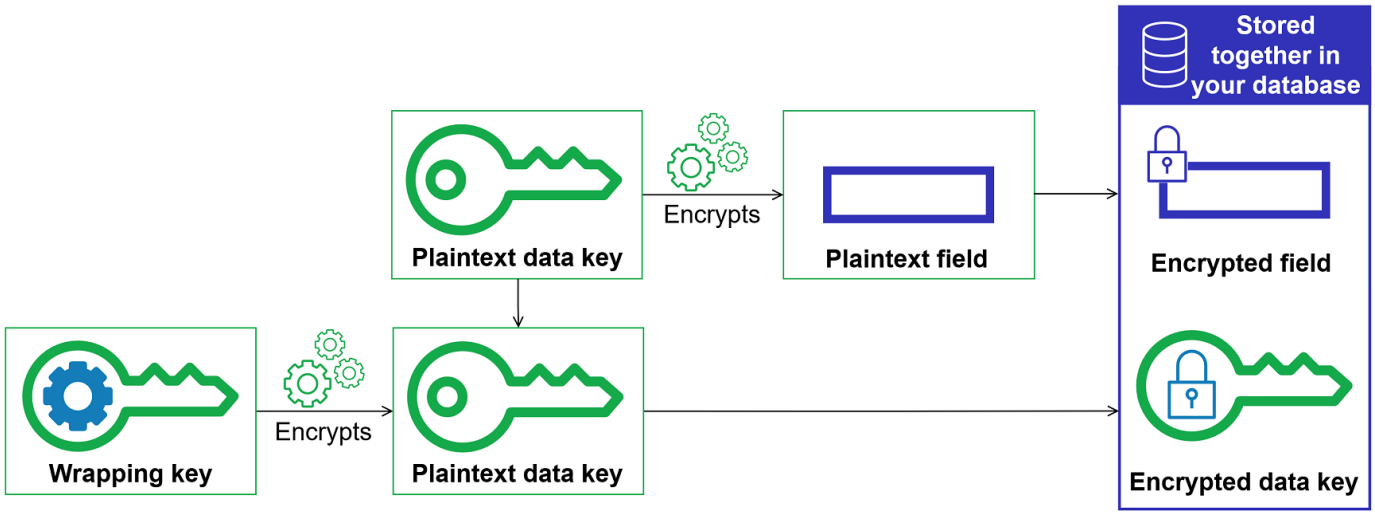
エンベロープ暗号化

暗号化されたデータのセキュリティは、復号できるデータキーを保護することによって部分的に異なります。1 つの受け入れられているデータキーを保護するベストプラクティスは暗号化することです。これを行うには、キー暗号化キーつまりラッピングキーと呼ばれる別の暗号化キーが必要です。データキーを暗号化するためにラッピングキーを使用するこの方法はエンベロープ暗号化と呼ばれています。

データキーの保護

AWS Database Encryption SDK は、各フィールドを一意のデータキーで暗号化します。その後、指定したラッピングキーで各データキーを暗号化します。暗号化されたデータキーをマテリアルの説明に格納します。

ラッピングキーを指定するには、キーリングを使用します。



複数のラッピングキーで同じデータを暗号化する

複数のラッピングキーを使用してデータキーを暗号化できます。ユーザーごとに異なるラッピングキーを指定したり、異なるタイプのラッピングキーを指定したり、場所ごとにそのように指定したい場合があります。各ラッピングキーでは、それぞれ同じデータキーを暗号化します。AWS Database Encryption SDK は、暗号化されたすべてのデータキーを、暗号化されたフィールドとともにマテリアルの説明に保存します。

データを復号するには、この暗号化されたデータキーを復号できる少なくとも 1 つのラッピングキーを指定する必要があります。

複数のアルゴリズムの強度の結合

デフォルトでは、 AWS Database Encryption SDK は、AES-GCM 対称暗号化、HMAC ベースのキー取得関数 (HKDF)、および ECDSA 署名を使用するアルゴリズムスイートを使用します。データキーを暗号化するには、ラッピングキーに適した対称または非対称の暗号化アルゴリズムを指定できます。

一般的に、対称キー暗号化アルゴリズムは迅速で、非対称またはパブリックキー暗号化よりも小さい暗号化テキストが生成されます。ただし、パブリックキーアルゴリズムはロールの本質的な分離を提供します。それぞれの長所を組み合わせるために、パブリックキー暗号化を使用してデータキーを暗号化できます。

可能な限り、いずれかの AWS KMS キーリングを使用することをお勧めします。AWS KMS キーリングを使用する場合、ラッピングキー AWS KMS key として非対称 RSA を指定することで、複数のアルゴリズムの長所を組み合わせることができます。また、対称暗号化 KMS キーを使用することもできます。

データキー

データキーは、 AWS Database Encryption SDK が暗号化アクションENCRYPT_AND_SIGNでマークされたレコード内のフィールドを暗号化するために使用する暗号化キーです。各データキーは、暗号化キーの要件に準拠したバイト配列です。 AWS Database Encryption SDK は、一意のデータキーを使用して各属性を暗号化します。

データキーを指定、生成、実装、拡張、保護、使用する必要はありません。 AWS Database Encryption SDK で暗号化オペレーションや復号オペレーションを呼び出しても、上記のアクションは行われません。

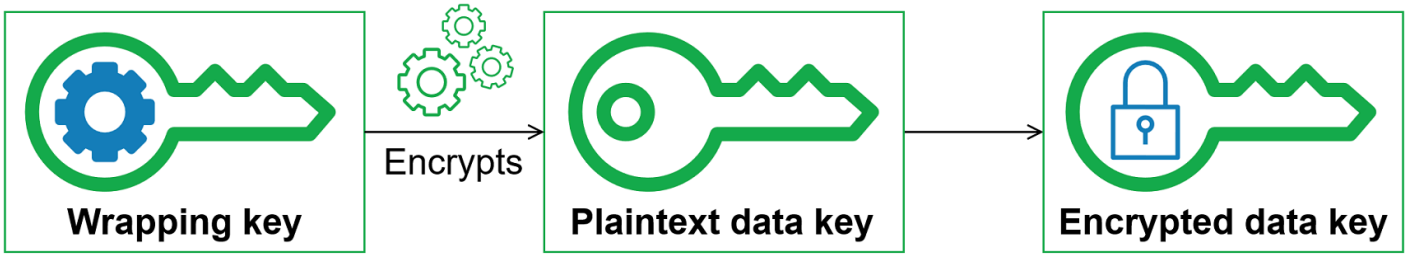
データキーを保護するために、 AWS Database Encryption SDK はラッピングキーと呼ばれる 1 つ以上のキー暗号化キーでデータキーを暗号化します。 AWS Database Encryption SDK は、プレーンテキストのデータキーを使用してデータを暗号化した後、できるだけ早くメモリから削除します。その後、暗号化されたデータキーをマテリアルの説明に格納します。詳細については、「AWS Database Encryption SDK の仕組み」を参照してください。

> **Tip**
>
> AWS Database Encryption SDK では、データキーとデータ暗号化キーを区別します。ベストプラクティスとして、サポートされているすべてのアルゴリズムスイートは鍵導出関数を使用します。鍵導出関数は、データキーを入力として受け取り、レコードの暗号化に実際に使用されたデータ暗号化キーを返します。そのため、データは、データキー「によって」暗号化されているというよりは、データキーの「下で」暗号化されていると言えます。

暗号化された各データキーには、暗号化したラッピングキーの識別子を含むメタデータが含まれます。このメタデータにより、 AWS Database Encryption SDK は復号時に有効なラッピングキーを識別できます。

ラッピングキー

ラッピングキーは、 AWS Database Encryption SDK がレコードを暗号化するデータキーを暗号化するために使用するキー暗号化キーです。各データキーは、1 つまたは複数のラッピングキーで暗号化することができます。キーリングの設定時に、データの保護に使用するラッピングキーを決定します。



AWS Database Encryption SDK は、 AWS Key Management Service (AWS KMS) 対称暗号化 KMS キー (マルチリージョンキーを含む AWS KMS) と非対称 RSA KMS キー、raw AES-GCM (Advanced Encryption Standard/Galois Counter Mode) キー、raw RSA キーなど、一般的に使用されるいくつかのラッピングキーをサポートしています。可能な場合は常に、KMS キーを使用することをお勧めします。どのラッピングキーを使用すべきかを知るには、「ラッピングキーの選択」を参照してください。

エンベロープ暗号化を使用する場合は、認可されていないアクセスからラッピングキーを保護する必要があります。これは、次のいずれかの方法で行うことができます。

- この目的のために設計された AWS Key Management Service (AWS KMS) などのサービスを使用します。

- https://en.wikipedia.org/wiki/Hardware_security_module によって提供されているようなAWS CloudHSMハードウェアセキュリティモジュール (HSM) を使用します。
- 他のキー管理ツールやサービスを使用します。

キー管理システムがない場合は、 をお勧めします AWS KMS。 AWS Database Encryption SDK は と統合され AWS KMS 、ラッピングキーの保護と使用に役立ちます。

キーリング

暗号化と復号に使用するラッピングキーを指定するには、キーリングを使用します。 AWS Database Encryption SDK が提供するキーリングを使用することも、独自の実装を設計することもできます。

キーリングは、データキーの生成、暗号化、復号を行います。また、署名内の Hash-Based Message Authentication Code (HMAC) を計算するために使用される MAC キーも生成します。キーリングを定義するとき、データキーを暗号化するラッピングキーを指定できます。ほとんどのキーリングは、少なくとも 1 つのラッピングキーを指定するか、ラッピングキーを提供および保護するサービスを指定します。暗号化時に、 AWS Database Encryption SDK はキーリングで指定されたすべてのラッピングキーを使用してデータキーを暗号化します。 AWS Database Encryption SDK が定義するキーリングの選択と使用については、「キーリングの使用」を参照してください。

暗号化アクション

暗号化アクションは、レコード内の各フィールドに対してどのアクションを実行するかを暗号化プログラムに指示します。

暗号化アクションの値は次のいずれかになります。

- [暗号化して署名] – フィールドを暗号化します。暗号化されたフィールドを署名に含めます。
- [署名のみ] – 署名にフィールドを含めます。
- 署名して暗号化コンテキストに含める – 署名と暗号化コンテキストに フィールドを含めます。

  デフォルトでは、パーティションキーとソートキーは、暗号化コンテキストに含まれる唯一の属性です。AWS KMS 階層キーリングのブランチキー ID サプライヤーが暗号化コンテキストからの復号に必要なブランチキーを特定SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTできるように、追加のフィールドを として定義することを検討してください。詳細については、「ブランチキー ID サプライヤー」を参照してください。

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 暗号化アクションを使用するには、AWS Database Encryption SDK のバージョン 3.3 以降を使用する必要があります。データモデルを更新して を含める前に、すべてのリーダーに新しいバージョンをデプロイしますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

- [何もしない] – フィールドを暗号化したり、署名に含めたりしません。

機密データを格納できるすべてのフィールドは、暗号化と署名を使用します。プライマリキー値 (DynamoDB テーブルのパーティションキーやソートキーなど) には、署名のみを使用するか、署名を使用して暗号化コンテキストに を含めます。Sign を指定して暗号化コンテキスト属性に含める場合、パーティション属性とソート属性も Sign で、暗号化コンテキストに含める必要があります。マテリアルの説明に暗号化アクションを指定する必要はありません。 AWS Database Encryption SDK は、マテリアルの説明が保存されている フィールドに自動的に署名します。

暗号化アクションは慎重に選択してください。不確かな場合は、暗号化と署名を使用します。 AWS Database Encryption SDK を使用してレコードを保護すると、既存の ENCRYPT_AND_SIGN、、SIGN_ONLYまたは SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドを に変更したりDO_NOTHING、既存のDO_NOTHINGフィールドに割り当てられた暗号化アクションを変更したりすることはできません。ただし、データモデルに他の変更を加えることはできます。例えば、単一のデプロイで暗号化フィールドを追加または削除できます。

マテリアル記述

マテリアルの説明は、暗号化されたレコードのヘッダーとして機能します。 AWS Database Encryption SDK を使用してフィールドを暗号化して署名すると、エンクリプタは暗号化マテリアルをアセンブルするときにマテリアルの説明を記録し、エンクリプタがレコードに追加する新しいフィールド (aws_dbe_head) にマテリアルの説明を保存します。

マテリアルの説明は、データキーの暗号化されたコピーと、暗号化アルゴリズム、暗号化コンテキスト、暗号化と署名の命令などの他の情報を含む、ポータブルな形式のデータ構造です。暗号化プログラムは、暗号化および署名のために暗号マテリアルをアセンブルする際に、マテリアルの説明を記録します。後で、フィールドを検証および復号するために暗号マテリアルをアセンブルする必要がある場合は、そのマテリアルの説明をガイドとして使用します。

暗号化されたデータキーを暗号化されたフィールドと一緒に格納すると、復号オペレーションが合理化され、暗号化されたデータキーを、そのキーで暗号化したデータとは別に格納および管理する必要がなくなります。

マテリアルの説明に関する技術的な情報については、「マテリアルの説明の形式」を参照してください。

暗号化コンテキスト

暗号化オペレーションのセキュリティを向上させるために、 AWS Database Encryption SDK には、レコードを暗号化して署名するすべてのリクエストに暗号化コンテキストが含まれています。

暗号化コンテキストは、任意のシークレットではない追加認証データを含む名前と値のペアのセットです。 AWS Database Encryption SDK には、データベースの論理名とプライマリキー値 (DynamoDB テーブルのパーティションキーとソートキーなど) が暗号化コンテキストに含まれます。フィールドを暗号化して、これに署名する場合、暗号化コンテキストは暗号化されたレコードに暗号化されてバインドされます。これにより、フィールドを復号するために同じ暗号化コンテキストが必要になります。

AWS KMS キーリングを使用する場合、 AWS Database Encryption SDK は暗号化コンテキストを使用して、キーリングが行う呼び出しで追加の認証データ (AAD) も提供します AWS KMS。

デフォルトのアルゴリズムスイートを使用するたびに、暗号マテリアルマネージャー (CMM) は、予約名 `aws-crypto-public-key` と、パブリック検証キーを表す値で構成される名前と値のペアを暗号化コンテキストに追加します。パブリック検証キーはマテリアルの説明に格納されます。

暗号化マテリアルマネージャー

暗号マテリアルマネージャー (CMM) は、データの暗号化、復号、署名に使用される暗号マテリアルを組み立てます。デフォルトのアルゴリズムスイートを使用する場合、暗号マテリアルには、プレーンテキストおよび暗号化されたデータキー、対称署名キー、および非対称署名キーが含まれます。CMM を直接操作することは決してありません。このためには、暗号化メソッドおよび復号メソッドを使用します。

CMM は AWS Database Encryption SDK とキーリングの間の連絡係として機能するため、ポリシーの適用のサポートなど、カスタマイズと拡張の理想的なポイントです。CMM を明示的に指定することはできますが、必須ではありません。キーリングを指定すると、 AWS Database Encryption SDK はデフォルトの CMM を作成します。デフォルトの CMM は、指定したキーリングから暗号化マテリアルまたは復号マテリアルを取得します。これには、AWS Key Management Service (AWS KMS) などの暗号化サービスの呼び出しが含まれる場合があります。

対称暗号化と非対称暗号化

対称暗号化では、データの暗号化と復号化に同じキーが使用されます。

非対称暗号化では、数学的に関連するデータキーペアが使用されます。ペアの 1 つのキーでデータが暗号化され、ペアの他のキーだけでデータが復号されます。

AWS Database Encryption SDK はエンベロープ暗号化を使用します。データは対称データキーで暗号化されます。対称データキーを 1 つ以上の対称または非対称のラッピングキーで暗号化します。データキーの暗号化されたコピーを少なくとも 1 つ含むマテリアルの説明をレコードに追加します。

データの暗号化 (対称暗号化)

データを暗号化するために、 AWS Database Encryption SDK は対称データキーと、対称暗号化アルゴリズムを含むアルゴリズムスイートを使用します。データを復号するために、 AWS Database Encryption SDK は同じデータキーと同じアルゴリズムスイートを使用します。

データキーの暗号化 (対称暗号化または非対称暗号化)

暗号化および復号のオペレーションに指定するキーリングにより、対称データキーの暗号化および復号方法が決まります。対称暗号化 KMS キーを持つ AWS KMS キーリングなどの対称暗号化を使用するキーリング、または非対称 RSA KMS キーを持つキーリングなどの AWS KMS 非対称暗号化を使用するキーリングを選択できます。

キーコミットメント

AWS Database Encryption SDK は、キーコミットメント (堅牢性とも呼ばれます) をサポートしています。これは、各暗号文を 1 つのプレーンテキストにのみ復号できるセキュリティプロパティです。これを実行するために、キーコミットメントを使用することで、レコードを暗号化したデータキーのみが復号に使用されるようになります。 AWS Database Encryption SDK には、すべての暗号化および復号オペレーションに対するキーコミットメントが含まれています。

最新の対称暗号 (AES を含む) のほとんどは、 AWS Database Encryption SDK がレコードENCRYPT_AND_SIGNでマークされた各プレーンテキストフィールドを暗号化するために使用する一意のデータキーなど、1 つのシークレットキーでプレーンテキストを暗号化します。同じデータキーでこのレコードを復号すると、元のデータと同じプレーンテキストが返されます。別のキーで復号化すると、通常は失敗します。2 つの異なるキーを使用して暗号文を復号することは難しいですが、技術的には可能です。まれに、数バイトの暗号化テキストを別の理解可能なプレーンテキストに部分的に複号できるキーを見つけることは可能です。

AWS Database Encryption SDK は、常に 1 つの一意のデータキーで各属性を暗号化します。複数のラッピングキーでそのデータキーを暗号化する場合がありますが、ラッピングキーは常に同じデータキーを暗号化します。ただし、手動で作成した高度な暗号化されたレコードには、実際には異なるデータキーが含まれて、それぞれ異なるラッピングキーによって暗号化されることがあります。例えば、あるユーザーが暗号化されたレコードを復号すると 0x0 (false) を返し、同じ暗号化されたレコードを別のユーザーが復号すると 0x1 (true) となることがあります。

このシナリオを防ぐために、 AWS Database Encryption SDK には、暗号化および復号時にキーコミットメントが含まれています。暗号化メソッドは、暗号文を生成した一意のデータキーを、データキーの導出を使用してマテリアルの説明に基づいて計算された Hash-based Message Authentication Code (HMAC) であるキーコミットメントに暗号的にバインドします。その後、キーコミットメントをマテリアルの説明に格納します。キーコミットメントを使用してレコードを復号すると、 AWS Database Encryption SDK はその暗号化されたレコードの唯一のキーがデータキーであることを確認します。データキーの検証が失敗すると、復号オペレーションは失敗します。

デジタル署名

AWS Database Encryption SDK は、認証された暗号化アルゴリズム、AES-GCM、および復号プロセスを使用してデータを暗号化し、デジタル署名を使用せずに暗号化されたメッセージの整合性と信頼性を検証します。しかし、AES-GCM は対称キーを使用するため、暗号化テキストの復号化に使用されるデータキーを復号できる人は誰でも、新しい暗号化された暗号化テキストを手動で作成できるようになり、セキュリティ上の懸念が生じる可能性があります。たとえば、 をラッピングキー AWS KMS key として使用すると、 アクセス`kms:Decrypt`許可を持つユーザーは、 を呼び出すことなく暗号化された暗号文を作成できます`kms:Encrypt`。

この問題を回避するために、デフォルトのアルゴリズムスイートは、暗号化されたレコードに Elliptic Curve Digital Signature Algorithm (ECDSA) 署名を追加します。デフォルトのアルゴリズムスイートは、認証された暗号化アルゴリズムである AES-GCM を使用して `ENCRYPT_AND_SIGN` とマークされたレコード内のフィールドを暗号化します。次に、、、および とマークされたレコードのフィールドで`SIGN_ONLY`、ハッシュベースのメッセージ認証コード (HMACs) `ENCRYPT_AND_SIGN`と非対称 ECDSA 署名の両方を計算します`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。復号プロセスでは、署名を使用して、認可されたユーザーがレコードを暗号化したことを検証します。

デフォルトのアルゴリズムスイートを使用すると、 AWS Database Encryption SDK は暗号化されたレコードごとに一時的なプライベートキーとパブリックキーペアを生成します。 AWS Database Encryption SDK は、パブリックキーをマテリアルの説明に保存し、プライベートキーを破棄します。これにより、パブリックキーで検証する別の署名を誰も作成できなくなります。このアルゴリズ

ムは、マテリアルの説明で追加の認証済みデータとして暗号化されたデータキーにパブリックキーをバインドし、フィールドのみを復号できるユーザーがパブリックキーを変更したり、署名の検証に影響を与えたりするのを防ぎます。

AWS Database Encryption SDK には、常に HMAC 検証が含まれています。ECDSA デジタル署名はデフォルトで有効になっていますが、必須ではありません。データを暗号化するユーザーとデータを復号するユーザーが同等に信頼されている場合は、パフォーマンスを改善するためにデジタル署名を含まないアルゴリズムスイートの使用を検討することをお勧めします。代替アルゴリズムスイートの選択の詳細については、「アルゴリズムスイートの選択」を参照してください。

> **Note**
>
> キーリングがエンクリプタと復号器を区別しない場合、デジタル署名は暗号化値を提供しません。

非対称 RSA AWS KMS キーリングを含む AWS KMS キーリングは、 AWS KMS キーポリシーと IAM ポリシーに基づいてエンクリプタと復号器を区別できます。

暗号化の性質上、次のキーリングはエンクリプタと復号器を区別できません。

- AWS KMS 階層キーリング
- AWS KMS ECDH キーリング
- Raw AES キーリング
- Raw RSA キーリング
- Raw ECDH キーリング

AWS Database Encryption SDK の仕組み

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK は、データベースに保存するデータを保護するために特別に設計されたクライアント側の暗号化ライブラリを提供します。ライブラリには、拡張が可能でまた変更な

しで使用できる安全な実装が含まれています。カスタムコンポーネントの定義と使用の詳細については、データベース実装の GitHub リポジトリを参照してください。

このセクションのワークフローでは、 AWS Database Encryption SDK がデータベース内のデータを暗号化、署名、復号、検証する方法について説明します。これらのワークフローは、抽象的な要素とデフォルト機能を使用した基本的なプロセスを表します。 AWS Database Encryption SDK がデータベース実装と連携する方法の詳細については、「データベースの暗号化された内容」トピックを参照してください。

AWS Database Encryption SDK は、エンベロープ暗号化を使用してデータを保護します。各レコードは一意のデータキーで暗号化されます。データキーは、暗号化アクションで ENCRYPT_AND_SIGN とマークされた各フィールドの一意のデータ暗号化キーを導出するために使用されます。その後、データキーのコピーが、指定したラッピングキーによって暗号化されます。暗号化されたレコードを復号するために、 AWS Database Encryption SDK は、指定したラッピングキーを使用して、少なくとも 1 つの暗号化されたデータキーを復号します。その後、暗号文を復号し、プレーンテキストのエントリを返すことができます。

AWS Database Encryption SDK で使用される用語の詳細については、「」を参照してくださいAWS Database Encryption SDK の概念。

暗号化および署名

AWS Database Encryption SDK は、データベース内のレコードを暗号化、署名、検証、復号するレコードエンクリプタです。レコードに関する情報と、暗号化して署名するフィールドに関する指示が取り込まれます。指定したラッピングキーから設定された暗号マテリアルマネージャーから、暗号マテリアルとその使用方法に関する指示を取得します。

次のチュートリアルでは、 AWS Database Encryption SDK がデータエントリを暗号化して署名する方法について説明します。

1. 暗号化マテリアルマネージャーは、 AWS Database Encryption SDK に 1 つのプレーンテキストデータキー、指定されたラッピングキーで暗号化されたデータキーのコピー、MAC キーという一意のデータ暗号化キーを提供します。 ???

   > **Note**
   >
   > 複数のラッピングキーでデータキーを暗号化できます。各ラッピングキーは、データキーの個別のコピーを暗号化します。 AWS Database Encryption SDK は、暗号化されたすべてのデータキーをマテリアルの説明に保存します。 AWS Database Encryption

SDK は、マテリアルの説明を格納するレコードに新しいフィールド (aws_dbe_head) を追加します。
MAC キーは、データキーの暗号化された各コピーについて導出されます。MAC キーは、マテリアルの説明には格納されません。代わりに、復号メソッドは、ラッピングキーを使用して MAC キーを再度導出します。

2. 暗号化メソッドは、指定した暗号化アクションで ENCRYPT_AND_SIGN とマークされた各フィールドを暗号化します。
3. 暗号化メソッドは、データキーから commitKey を導出し、それを使用してキーコミットメントの値を生成して、その後にデータキーを破棄します。
4. 暗号化メソッドは、マテリアルの説明をレコードに追加します。マテリアルの説明には、暗号化されたデータキーと、暗号化されたレコードに関する他の情報が含まれます。マテリアルの説明に含まれる情報の詳細なリストについては、「マテリアルの説明の形式」を参照してください。
5. 暗号化メソッドは、ステップ 1 で返された MAC キーを使用して、マテリアルの説明、暗号化コンテキスト、および暗号化アクションSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTで ENCRYPT_AND_SIGN、SIGN_ONLY、または とマークされた各フィールドの正規化におけるハッシュベースのメッセージ認証コード (HMAC) 値を計算します。HMAC の値は、暗号化メソッドがレコードに追加する新しいフィールド (aws_dbe_foot) に格納されます。
6. 暗号化メソッドは、マテリアルの説明、暗号化コンテキスト、および ENCRYPT_AND_SIGN、、または とマークされた各フィールドの正規化にわたって ECDSA 署名を計算SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTしSIGN_ONLY、ECDSA 署名を aws_dbe_footフィールドに保存します。

> **Note**
>
> ECDSA 署名はデフォルトで有効になっていますが、必須ではありません。

7. 暗号化メソッドは、暗号化および署名されたレコードをデータベースに格納します。

復号および検証

1. 暗号マテリアルマネージャー (CMM) は、プレーンテキストのデータキーおよび関連付けられた MAC キーを含む、マテリアルの説明に格納されている復号マテリアルを復号メソッドに提供します。

- CMM は、指定されたキーリング内のラッピングキーを使用して暗号化されたデータキーを復号し、プレーンテキストのデータキーを返します。

2. 復号メソッドは、マテリアルの説明内のキーコミットメントの値を比較および検証します。
3. 復号メソッドは、署名フィールド内の署名を検証します。

   これはENCRYPT_AND_SIGN、定義した許可された認証されていないフィールドのSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTリストから、、SIGN_ONLY、または とマークされているフィールドを識別します。 ???復号メソッドは、ステップ 1 で返された MAC キーを使用して、ENCRYPT_AND_SIGN、、SIGN_ONLYまたは とマークされたフィールドの HMAC 値を再計算して比較しますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。その後、暗号化コンテキストに格納されているパブリックキーを使用して ECDSA 署名を検証します。
4. 復号メソッドは、プレーンテキストデータキーを使用して、ENCRYPT_AND_SIGN とマークされた各値を復号します。 AWS Database Encryption SDK は、プレーンテキストのデータキーを破棄します。
5. 復号方法は、プレーンテキストレコードを返します。

AWS Database Encryption SDK でサポートされているアルゴリズムスイート

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

アルゴリズムスイートは、暗号化アルゴリズムと関連する値の集合です。暗号化システムはアルゴリズム実装を使用して暗号文を生成します。

AWS Database Encryption SDK は、アルゴリズムスイートを使用してデータベース内のフィールドを暗号化して署名します。サポートされているすべてのアルゴリズムスイートは、AES-GCM と呼ばれる Galois/Counter Mode (GCM) を使用した Advanced Encryption Standard (AES) アルゴリズムを使用して raw データを暗号化します。 AWS Database Encryption SDK は 256 ビットの暗号化キーをサポートしています。認証タグの長さは常に 16 バイトです。

AWS データベース暗号化 SDK アルゴリズムスイート

アルゴリズム	暗号化アルゴリズム	データキーの長さ (ビット)	キー導出アルゴリズム	対称署名アルゴリズム	対称署名アルゴリズム	キーコミットメント
デフォルト	AES-GCM	256	SHA-512 を使用する HKDF	HMAC-SHA-384	P-384 および SHA-384 を使用する ECDSA	SHA-512 を使用する HKDF
ECDSA デジタル署名を使用しない AES-GCM	AES-GCM	256	SHA-512 を使用する HKDF	HMAC-SHA-384	なし	SHA-512 を使用する HKDF

暗号化アルゴリズム

使用する暗号化アルゴリズムの名前とモード。 AWS Database Encryption SDK のアルゴリズムスイートは、Galois/Counter Mode (GCM) で Advanced Encryption Standard (AES) アルゴリズムを使用します。

データキーの長さ

データキーの長さ (ビット単位)。 AWS Database Encryption SDK は 256 ビットのデータキーをサポートしています。データキーは、HMAC extract-and-expandキー取得関数 (HKDF) への入力として使用されます。HKDF の出力は、暗号化アルゴリズムのデータ暗号化キーとして使用されます。

キー導出アルゴリズム

データ暗号化キーを取得するために使用される、HMAC ベースの抽出および展開キー取得関数 (HKDF)。 AWS Database Encryption SDK は、RFC 5869 で定義された HKDF を使用します。

- 使用されるハッシュ関数は SHA-512 です
- 抽出ステップの場合
  - ソルトは使用されません。RFC の場合、ソルトはゼロの文字列に設定されます。

    - 入力キーマテリアルは、キーリングのデータキーです。
- 展開ステップの場合
    - 入力疑似ランダムキーは抽出ステップからの出力です。
    - キーラベルは、ビッグエンディアンバイト順序の DERIVEKEY 文字列を UTF-8 でエンコードしたバイトです。
    - 入力情報は、アルゴリズム ID とキー ラベルの連結です (この順序)。
    - 出力キーマテリアルの長さはデータキーの長さです。この出力は、暗号化アルゴリズムのデータ暗号化キーとして使用されます。

対称署名アルゴリズム

対称署名の生成に使用されるハッシュベースのメッセージ認証コード (HMAC) アルゴリズム。サポートされているすべてのアルゴリズムスイートには、HMAC 検証が含まれています。

AWS Database Encryption SDK は、マテリアルの説明と、ENCRYPT_AND_SIGN、、SIGN_ONLYまたは とマークされたすべてのフィールドをシリアル化しますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。次に、暗号化ハッシュ関数アルゴリズム (SHA-384) で HMAC を使用して正規化に署名します。

対称 HMAC 署名は、 AWS Database Encryption SDK がレコードに追加する新しいフィールド (aws_dbe_foot) に保存されます。

対称署名アルゴリズム

非対称デジタル署名を生成するために使用される署名アルゴリズム。

AWS Database Encryption SDK は、マテリアルの説明と、ENCRYPT_AND_SIGN、、SIGN_ONLYまたは とマークされたすべてのフィールドをシリアル化しますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。次に、楕円曲線デジタル署名アルゴリズム (ECDSA) を以下の詳細とともに使用して正規化に署名します。

- 使用される楕円曲線は、デジタル署名標準 (DSS) (FIPS PUB 186-4) で定義されている P-384 です。 http://doi.org/10.6028/NIST.FIPS.186-4
- 使用されるハッシュ関数は SHA-384 です。

非対称 ECDSA 署名は、 aws_dbe_footフィールドに対称 HMAC 署名とともに保存されます。

ECDSA デジタル署名はデフォルトで含まれていますが、必須ではありません。

キーコミットメント

コミットキーの取得に使用される HMAC extract-and-expandキー取得関数 (HKDF)。

- 使用されるハッシュ関数は SHA-512 です
- 抽出ステップの場合
  - ソルトは使用されません。RFC の場合、ソルトはゼロの文字列に設定されます。
  - 入力キーマテリアルは、キーリングのデータキーです。
- 展開ステップの場合
  - 入力疑似ランダムキーは抽出ステップからの出力です。
  - 入力情報は、`COMMITKEY`文字列の UTF-8-encodedされたバイトをビッグエンディアンバイト順で表したものです。
  - 出力キーマテリアルの長さは 256 ビットです。この出力はコミットキーとして使用されます。

コミットキーは、マテリアルの説明に対するレコードコミットメント、つまり個別の 256 ビット Hash-Based Message Authentication Code (HMAC) ハッシュを計算します。アルゴリズムスイートへのキーコミットメントの追加に関する技術的な説明については、Cryptology ePrint Archiveの「Key Committing AEADs」を参照してください。

デフォルトのアルゴリズムスイート

デフォルトでは、 AWS Database Encryption SDK は、AES-GCM、HMAC extract-and-expandキー取得関数 (HKDF)、HMAC 検証、ECDSA デジタル署名、キーコミットメント、および 256 ビット暗号化キーを備えたアルゴリズムスイートを使用します。

デフォルトのアルゴリズムスイートには、HMAC 検証 (対称署名) と ECDSA デジタル署名 (非対称署名) が含まれます。これらの署名は、 AWS Database Encryption SDK がレコードに追加する新しいフィールド (`aws_dbe_foot`) に保存されます。ECDSA デジタル署名は、認可ポリシーで 1 つのユーザーのセットにデータの暗号化を許可し、別のユーザーのセットにデータの復号を許可する場合に特に便利です。

デフォルトのアルゴリズムスイートでは、データキーをレコードに結び付ける HMAC ハッシュであるキーコミットメントも取得されます。キーコミットメント値は、マテリアルの説明とコミットキーから計算された HMAC です。その後、キーコミットメントの値は、マテリアルの説明に格納されます。キーのコミットメントにより、各暗号文は 1 つのプレーンテキストのみに確実に復号されます。これは、暗号化アルゴリズムへの入力として使用されるデータキーを検証することによって行います。暗号化時に、アルゴリズムスイートはキーコミットメント HMAC を取得します。復号する前に、データキーが同じキーコミットメント HMAC を生成することを検証します。一致しない場合、復号呼び出しは失敗します。

ECDSA デジタル署名を使用しない AES-GCM

デフォルトのアルゴリズムスイートはほとんどのアプリケーションに適していますが、代替アルゴリズムスイートを選択できます。たとえば、一部の信頼モデルは、ECDSA デジタル署名のないアルゴリズムスイートによって満たされます。このスイートは、データを暗号化するユーザーとデータを復号するユーザーが等しく信頼されている場合にのみ使用します。

すべての AWS Database Encryption SDK アルゴリズムスイートには、HMAC 検証 (対称署名) が含まれています。唯一の違いは、ECDSA デジタル署名のない AES-GCM アルゴリズムスイートには、信頼性と否認のない追加のレイヤーを提供する非対称署名がないことです。

たとえば、キーリング 、、`wrappingKeyA`および `wrappingKeyB` に複数のラッピングキーがあり`wrappingKeyC`、 を使用してレコードを復号する場合`wrappingKeyA`、HMAC 対称署名は、レコードが `wrappingKeyA` にアクセスできるユーザーによって暗号化されたことを確認しますデフォルトのアルゴリズムスイートを使用した場合、HMACs の同じ検証を提供し`wrappingKeyA`、さらに ECDSA デジタル署名を使用して、レコードが `wrappingKeyA` の暗号化アクセス許可を持つユーザーによって暗号化されたことを確認します。

デジタル署名のない AES-GCM アルゴリズムスイートを選択するには、暗号化設定に次のスニペットを含めます。

Java

次のスニペットは、ECDSA デジタル署名のない AES-GCM アルゴリズムスイートを指定します。詳細については、「the section called “暗号化設定”」を参照してください。

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

C# / .NET

次のスニペットは、ECDSA デジタル署名のない AES-GCM アルゴリズムスイートを指定します。詳細については、「the section called “暗号化設定”」を参照してください。

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Rust

次のスニペットは、ECDSA デジタル署名のない AES-GCM アルゴリズムスイートを指定します。詳細については、「the section called “暗号化設定”」を参照してください。

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

での AWS Database Encryption SDK の使用 AWS KMS

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK を使用するには、キーリングを設定し、1 つ以上のラッピングキーを指定する必要があります。キーのインフラストラクチャがない場合は、AWS Key Management Service (AWS KMS) を使用することをお勧めします。

AWS Database Encryption SDK は、2 種類の AWS KMS キーリングをサポートしています。従来の AWS KMS キーリングは、データキーを生成、暗号化、復号するために AWS KMS keys を使用します。対称暗号化 (SYMMETRIC_DEFAULT) または非対称 RSA KMS キーのいずれかを使用できます。 AWS Database Encryption SDK は一意のデータキーを使用してすべてのレコードを暗号化および署名するため、 AWS KMS キーリングは暗号化および復号オペレーション AWS KMS ごとに を呼び出す必要があります。への呼び出し数を最小限に抑える必要があるアプリケーションの場合 AWS KMS、 AWS Database Encryption SDK はAWS KMS 階層キーリングもサポートします。階層キーリングは、Amazon DynamoDB テーブルに保持されている AWS KMS 保護されたブランチキーを使用して AWS KMS 呼び出しの数を減らし、暗号化および復号オペレーションで使用されるブランチキーマテリアルをローカルにキャッシュする暗号化マテリアルキャッシュソリューションです。可能な限り、 AWS KMS キーリングを使用することをお勧めします。

Database Encryption SDK AWS とやり取りするには AWS KMS、 の AWS KMS モジュールが必要です AWS SDK for Java。

で AWS Database Encryption SDK を使用する準備をするには AWS KMS

1. を作成します AWS アカウント。方法については、 AWS ナレッジセンターの「新しい Amazon Web Services アカウントを作成してアクティブ化する方法」を参照してください。
2. 対称暗号化を作成します AWS KMS key。ヘルプについては、「AWS Key Management Service デベロッパーガイド」の「キーの作成」を参照してください。

> **Tip**
>
> AWS KMS key プログラムで を使用するには、 の Amazon リソースネーム (ARN) が必要です AWS KMS key。 AWS KMS keyの ARN を見つけるには、「AWS Key

Management Service デベロッパーガイド」の「キー ID と ARN を検索する」を参照してください。

3. アクセスキー ID とセキュリティアクセスキーを生成します。IAM ユーザーのアクセスキー ID とシークレットアクセスキーを使用するか、 を使用して AWS Security Token Service 、アクセスキー ID、シークレットアクセスキー、セッショントークンを含む一時的なセキュリティ認証情報を使用して新しいセッションを作成できます。セキュリティのベストプラクティスとして、IAM ユーザーまたは AWS (ルート) ユーザーアカウントに関連付けられた長期的な認証情報の代わりに、一時的な認証情報を使用することをお勧めします。

   アクセスキーを使用して IAM ユーザーを作成するには、「IAM ユーザーガイド」の「IAM ユーザーの作成」を参照してください。

   一時的なセキュリティ認証情報を生成するには、「IAM ユーザーガイド」の「一時的なセキュリティ認証情報のリクエスト」を参照してください。
4. の手順AWS SDK for Javaと、ステップ 3 で生成したアクセスキー ID とシークレットアクセスキーを使用して AWS 認証情報を設定します。一時的な認証情報を生成した場合は、セッショントークンも指定する必要があります。

   この手順により、 AWS SDKs へのリクエストに署名 AWS できます。とやり取りする AWS Database Encryption SDK のコードサンプルは、このステップを完了したことを AWS KMS 前提としています。

AWS Database Encryption SDK の設定

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK は、使いやすいように設計されています。 AWS Database Encryption SDK にはいくつかの設定オプションがありますが、デフォルト値はほとんどのアプリケーションで実用的で安全になるように慎重に選択されています。ただし、パフォーマンスを改善するために構成を調整したり、設計にカスタム機能を追加したりしたい場合があります。

トピック

プログラミング言語の選択

AWS Database Encryption SDK for DynamoDB は、複数のプログラミング言語で利用できます。言語の実装は、完全に相互運用可能で、同じ機能を提供するように設計されていますが、異なる方法で実装される可能性があります。通常は、アプリケーションと互換性のあるライブラリを使用します。

ラッピングキーの選択

AWS Database Encryption SDK は、各フィールドを暗号化するための一意の対称データキーを生成します。データキーを設定、管理、または使用する必要はありません。 AWS Database Encryption SDK がこれを行います。

ただし、各データキーを暗号化するには、1 つ以上のラッピングキーを選択する必要があります。AWS Database Encryption SDK は、AWS Key Management Service (AWS KMS) 対称暗号化 KMS キーと非対称 RSA KMS キーをサポートします。また、さまざまなサイズで提供する AES 対称キーと RSA 非対称キーもサポートします。ラッピングキーの安全性と耐久性はお客様の責任となりま

す。そのため、ハードウェアセキュリティモジュールや などのキーインフラストラクチャサービス で暗号化キーを使用することをお勧めします AWS KMS。

暗号化と復号のためにラッピングキーを指定するには、キーリングを使用します。使用するキーリングのタイプに応じて、1 つのラッピングキー、または同じタイプもしくは異なるタイプの複数のラッピングキーを指定できます。複数のラッピングキーを使用してデータキーをラップする場合、各ラッピングキーは同じデータキーのコピーを暗号化します。暗号化されたデータキー (ラッピングキーごとに 1 つ) は、暗号化されたフィールドと一緒に格納されるマテリアルの説明に格納されます。データを復号するには、 AWS Database Encryption SDK はまずラッピングキーのいずれかを使用して暗号化されたデータキーを復号する必要があります。

可能な限り、いずれかの AWS KMS キーリングを使用することをお勧めします。 AWS Database Encryption SDK は、 AWS KMS キーリングと AWS KMS 階層キーリングを提供し、 への呼び出しの数を減らします AWS KMS。キーリング AWS KMS key で を指定するには、サポートされている AWS KMS キー識別子を使用します。 AWS KMS 階層キーリングを使用する場合は、キー ARN を指定する必要があります。キーのキー識別子の詳細については AWS KMS 、「 AWS Key Management Service デベロッパーガイド」の「キー識別子」を参照してください。

- AWS KMS キーリングで暗号化する場合、対称暗号化 KMS キーに任意の有効なキー識別子 (キー ARN、エイリアス名、エイリアス ARN、またはキー ID) を指定できます。非対称 RSA KMS キーを使用する場合は、キー ARN を指定する必要があります。

  暗号化時に KMS キーのエイリアス名またはエイリアス ARN を指定する場合、 AWS Database Encryption SDK は、そのエイリアスに現在関連付けられているキー ARN を保存します。エイリアスは保存されません。エイリアスの変更は、データキーの復号に使用される KMS キーには影響しません。
- デフォルトでは、 AWS KMS キーリングは strict モード (特定の KMS キーを指定する) でレコードを復号します。復号のために AWS KMS keys を識別するにはキー ARN を使用する必要があります。

  AWS KMS キーリングで暗号化すると、 AWS Database Encryption SDK は暗号化されたデータキーを使用して のキー ARN をマテリアルの説明 AWS KMS key に保存します。Strict モードで復号する場合、 AWS Database Encryption SDK は、ラッピングキーを使用して暗号化されたデータキーを復号しようとする前に、キーリングに同じキー ARN が表示されることを確認します。別のキー識別子を使用する場合、識別子が同じキーを参照している場合でも AWS KMS key、 AWS Database Encryption SDK は を認識または使用しません。
- 検出モードで復号する場合は、ラッピングキーを指定しません。まず、 AWS Database Encryption SDK は、マテリアルの説明に保存されたキー ARN を使用してレコードの復号を試みま

す。これが機能しない場合、 AWS Database Encryption SDK は、その KMS キーを所有またはアクセスできるユーザーに関係なく、暗号化された KMS キーを使用してレコードを復号 AWS KMS するよう に求めます。

raw AES キーまたは raw RSA キーペアをキーリング内のラッピングキーとして指定するには、名前空間と名前を指定する必要があります。復号する際には、暗号化の際に使用した各 raw ラッピングキーとまったく同じ名前空間と名前を使用する必要があります。別の名前空間または名前を使用する場合、 AWS Database Encryption SDK は、キーマテリアルが同じであっても、ラッピングキーを認識または使用しません。

検出フィルターの作成

KMS キーを使用して暗号化されたデータを復号する場合は、厳格モードで復号する、つまり、使用するラッピングキーを、指定したもののみに制限するのがベストプラクティスです。ただし、必要に応じて、ラッピングキーを指定しない検出モードで復号することもできます。このモードでは、その KMS キーを所有またはアクセスできるユーザーに関係なく、暗号化されたデータキーを暗号化した KMS キーを使用して復号 AWS KMS できます。

検出モードで復号する必要がある場合は、常に検出フィルターを使用することをお勧めします。これにより、使用できる KMS キーが、指定された AWS アカウント および パーティション内のキーに制限されます。検出フィルターはオプションですが、ベストプラクティスです。

次の表を使用して、検出フィルターのパーティションの値を決定します。

リージョン	パーティション
AWS リージョン	`aws`
中国リージョン	`aws-cn`
AWS GovCloud (US) Regions	`aws-us-gov`

次の例は、検出フィルターを作成する方法を示しています。コードを使用する前に、サンプル値を AWS アカウント および パーティションの有効な値に置き換えます。

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()?;
```

マルチテナンシーデータベースの使用

AWS Database Encryption SDK を使用すると、各テナントを個別の暗号化マテリアルで分離することで、共有スキーマを持つデータベースのクライアント側の暗号化を設定できます。マルチテナンシーデータベースを検討する場合は、セキュリティ要件と、マルチテナンシーがそれらのセキュリティ要件にどのように影響し得るかを確認してください。例えば、マルチテナントデータベースを使用すると、 AWS Database Encryption SDK を別のサーバー側の暗号化ソリューションと組み合わせる能力に影響する可能性があります。

データベース内で複数のユーザーが暗号化オペレーションを実行している場合は、いずれかの AWS KMS キーリングを使用して、暗号化オペレーションで使用する個別のキーを各ユーザーに提供できます。マルチテナンシーのクライアント側の暗号化ソリューション用のデータキーの管理は複雑になる場合があります。可能な場合は常に、データをテナンシーごとに整理することをお勧めします。テナンシーがプライマリキーの値 (Amazon DynamoDB テーブルのパーティションキーなど) によって識別される場合、キーの管理は簡単になります。

AWS KMS キーリングを使用して、各テナントを個別の AWS KMS キーリング および で分離できます AWS KMS keys。テナントごとに行われた呼び出しの量 AWS KMS に基づいて、 AWS KMS 階層

キーリングを使用して呼び出しを最小限に抑えることができます AWS KMS。AWS KMS 階層キーリングは、Amazon DynamoDB テーブルに保持されている AWS KMS 保護されたブランチキーを使用し、暗号化および復号オペレーションで使用されるブランチキーマテリアルをローカルにキャッシュすることで、 AWS KMS 呼び出しの数を減らす暗号化マテリアルキャッシュソリューションです。データベースに検索可能な暗号化を実装するには、 AWS KMS 階層キーリングを使用する必要があります。

署名付きビーコンの作成

AWS Database Encryption SDK は、標準ビーコンと複合ビーコンを使用して、クエリされたデータベース全体を復号することなく、暗号化されたレコードを検索できる検索可能な暗号化ソリューションを提供します。ただし、 AWS Database Encryption SDK は、プレーンテキストの署名付きフィールドから完全に設定できる署名付きビーコンもサポートしています。署名付きビーコンは、SIGN_ONLYフィールドと SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドに対してインデックスを作成し、複雑なクエリを実行する複合ビーコンの一種です。

例えば、マルチテナンシーデータベースがある場合、特定のテナンシーのキーによって暗号化されたレコードがあるかどうかを確認するために、データベースをクエリできるようにする署名付きビーコンを作成することをお勧めします。詳細については、「マルチテナンシーデータベース内のビーコンのクエリ」を参照してください。

署名付きビーコンを作成するには、 AWS KMS 階層キーリングを使用する必要があります。

署名付きビーコンを設定するには、次の値を指定します。

Java

複合ビーコン設定

次の例では、署名付きビーコン設定内でローカルで署名付きパートリストを定義します。

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

ビーコンバージョン定義

次の例では、ビーコンバージョンで署名付きパートリストをグローバルに定義します。ビーコンバージョンの定義の詳細については、「ビーコンの使用」を参照してください。

```
 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

完全なコードサンプルを参照: BeaconConfig.cs

署名付きビーコン設定

次の例では、署名付きビーコン設定内でローカルで署名付きパートリストを定義します。

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

ビーコンバージョン定義

次の例では、ビーコンバージョンで署名付きパートリストをグローバルに定義します。ビーコンバージョンの定義の詳細については、「ビーコンの使用」を参照してください。

```
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
            }
        }
    }
};
```

署名付きパートは、ローカルまたはグローバルに定義されたリストで定義できます。可能な限り、ビーコンバージョンのグローバルリストで署名付きパートを定義することをお勧めします。署名付きパートをグローバルに定義することで、各パートを 1 回定義し、そのパートを複数の複合ビーコン設定で再利用できます。署名付きパートを 1 回だけ使用する場合は、署名付きビーコン設定のローカルリストで定義できます。コンストラクタリストでは、ローカルパートとグローバルパートの両方を参照できます。

署名付きパートリストをグローバルに定義する場合は、署名付きビーコンがビーコン設定のフィールドをアセンブルできるすべての方法を識別するコンストラクタパートのリストを指定する必要があります。

> **Note**
>
> 署名付きパートリストをグローバルに定義するには、 AWS Database Encryption SDK のバージョン 3.2 以降を使用する必要があります。新しいパートをグローバルに定義する前に、すべてのリーダーに新しいバージョンをデプロイします。
> 既存のビーコン設定を更新して、署名付きパートリストをグローバルに定義することはできません。

ビーコン名

ビーコンをクエリする際に使用する名前。

署名付きビーコンの名前は、暗号化されていないフィールドと同じ名前にすることはできません。2 つのビーコンを同じ名前にすることはできません。

分割文字

署名付きビーコンを設定する部分を分離するために使用される文字。

分割文字は、署名付きビーコンの構築元となるフィールドのプレーンテキストの値に出現することはできません。

署名付きの部分のリスト

署名付きビーコンに含まれる署名付きフィールドを識別します。

各部分には、名前、ソース、プレフィックスが含まれている必要があります。ソースは、パートが識別する SIGN_ONLYまたは SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドです。ソースは、フィールド名、またはネストされたフィールドの値を参照するインデックスである必要があります。パーツ名がソースを識別する場合、ソースを省略すると、 AWS Database Encryption SDK は自動的にその名前をソースとして使用します。可能な場合は常に、部分名としてソースを指定することをお勧めします。プレフィックスには任意の文字列を指定できますが、一意である必要があります。署名付きビーコン内の 2 つの署名付きの部分に同じプレフィックスを付けることはできません。複合ビーコンによって提供される部分と他の部分を区別する短い値を使用することをお勧めします。

可能な限り、署名付きパートをグローバルに定義することをお勧めします。署名付きパートを 1 つの複合ビーコンでのみ使用する場合は、ローカルで定義することを検討してください。ローカルに定義されたパートは、グローバルに定義されたパートと同じプレフィックスまたは名前を持つことはできません。

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-" }
};
```

コンストラクタリスト (オプション)

署名付きの部分を署名付きビーコンによってアセンブルするさまざまな方法を定義するコンストラクターを識別します。

コンストラクタリストを指定しない場合、 AWS Database Encryption SDK は次のデフォルトのコンストラクタを使用して署名付きビーコンをアセンブルします。

- すべての署名付きの部分 (署名付きの部分のリストに追加された順)
- すべての部分は必須です

コンストラクタ

各コンストラクターは、署名付きビーコンをアセンブルする 1 つの方法を定義するコンストラクター部分の順序付きリストです。コンストラクター部分はリストに追加された順序で結合され、各部分は指定された分割文字で区切られます。

各コンストラクター部分は、署名付きの部分に名前を付け、その部分がコンストラクター内で必須であるか、またはオプションであるかを定義します。例えば、`Field1`、`Field1.Field2`、および `Field1.Field2.Field3` で署名付きビーコンをクエリする場合は、`Field2` および `Field3` をオプションとしてマークし、コンストラクターを 1 つ作成します。

各コンストラクターには、少なくとも 1 つの必須部分が必要です。クエリで `BEGINS_WITH` 演算子を使用できるように、各コンストラクターの最初の部分を必須にすることをお勧めします。

コンストラクターは、必要な部分がすべてレコード内に存在する場合に成功します。新しいレコードを書き込む際に、署名付きビーコンはコンストラクターのリストを使用して、指定された値からビーコンをアセンブルできるかどうかを判断します。コンストラクターがコンストラクターのリストに追加された順序でビーコンのアセンブルを試み、成功した最初のコンストラクターを使用します。コンストラクターが成功しない場合、ビーコンはレコードに書き込まれません。

すべてのリーダーとライターは、クエリの結果が確実に正しくなるようにコンストラクターの同じ順序を指定する必要があります。

独自のコンストラクターのリストを指定するには、次の手順を使用します。

1. 署名付きの部分ごとにコンストラクター部分を作成し、その部分が必須かどうかを定義します。

   コンストラクターの部分の名前は、署名されたフィールドの名前である必要があります。

   次の例は、1 つの署名付きフィールドのコンストラクターの部分を作成する方法を示しています。

   Java

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

   C# / .NET

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

2. ステップ 1 で作成したコンストラクター部分を使用して、署名付きビーコンをアセンブルする可能な方法ごとにコンストラクターを作成します。

   例えば、Field1.Field2.Field3 と Field4.Field2.Field3 をクエリする場合は、2 つのコンストラクターを作成する必要があります。Field1 と Field4 は、2 つの別個のコンストラクターで定義されているため、両方とも必須にすることができます。

   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. ステップ 2 で作成したすべてのコンストラクターを含むコンストラクターのリストを作成します。

   Java

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

   C# / .NET

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

4. 署名付きビーコンを作成する際に `constructorList` を指定します。

AWS Database Encryption SDK のキーストア

AWS Database Encryption SDK では、キーストアは、階層AWS KMS キーリングで使用される階層データを保持する Amazon DynamoDB テーブルです。キーストアは、階層キーリングを使用して暗号化オペレーションを実行する AWS KMS ために に対して行う必要がある呼び出しの数を減らすのに役立ちます。

キーストアは、階層キーリングがエンベロープ暗号化を実行し、データ暗号化キーを保護するために使用するブランチキーを保持および管理します。キーストアは、アクティブなブランチキーと以前のすべてのバージョンのブランチキーを保存します。アクティブなブランチキーは、ブランチキーの最新バージョンです。階層キーリングは、暗号化リクエストごとに一意のデータ暗号化キーを使用し、アクティブなブランチキーから派生した一意のラッピングキーを使用して各データ暗号化キーを暗号化します。階層キーリングは、アクティブなブランチキーと、その導出ラッピングキーの間に確立された階層に依拠します。

キーストアの用語と概念

キーストア

ブランチキーやビーコンキーなどの階層データを保持する DynamoDB テーブル。

ルートキー

キーストア内のブランチキーとビーコンキーを生成して保護する対称暗号化 KMS キー。

ブランチキー

エンベロープ暗号化用の一意のラッピングキーを取得するために再利用されるデータキー。1 つのキーストアに複数のブランチキーを作成できますが、各ブランチキーは一度に 1 つのアクティブなブランチキーバージョンのみを持つことができます。アクティブなブランチキーは、ブランチキーの最新バージョンです。

ブランチキーは、kms:GenerateDataKeyWithoutPlaintext オペレーション AWS KMS keys を使用して算出されます。

ラッピングキー

暗号化オペレーションで使用されるデータ暗号化キーを暗号化するために使用される一意のデータキー。

ラップキーはブランチキーから派生します。キー取得プロセスの詳細については、AWS KMS「階層キーリングの技術的な詳細」を参照してください。

データ暗号化キー

暗号化オペレーションで使用されるデータキー。階層キーリングは、暗号化リクエストごとに一意のデータ暗号化キーを使用します。

ビーコンキー

検索可能な暗号化用のビーコンを生成するために使用されるデータキー。詳細については、「検索可能な暗号化」を参照してください。

最小特権のアクセス許可の実装

キーストアと AWS KMS 階層キーリングを使用する場合は、次のロールを定義して最小特権の原則に従うことをお勧めします。

キーストア管理者

キーストア管理者は、キーストアと、キーストアが保持および保護するブランチキーを作成および管理します。キーストア管理者は、キーストアとして機能する Amazon DynamoDB テーブルへの書き込み権限を持つ唯一のユーザーである必要があります。これらは、`CreateKey`やなどの特権的な管理者オペレーションにアクセスできる唯一のユーザーである必要がありますVersionKey。これらのオペレーションは、キーストアアクションを静的に設定する場合にのみ実行できます。

`CreateKey` は、キーストア許可リストに新しい KMS キー ARN を追加できる特権オペレーションです。この KMS キーは、新しいアクティブなブランチキーを作成できます。KMS キーがブランチキーストアに追加されると、削除できないため、このオペレーションへのアクセスを制限することをお勧めします。

キーストアユーザー

ほとんどの場合、キーストアユーザーは、データを暗号化、復号、署名、検証する際に、階層キーリングを介してのみキーストアとやり取りします。そのため、キーストアとして機能する Amazon DynamoDB テーブルへの読み取りアクセス許可のみが必要です。キーストアユーザーは、、、 などの暗号化オペレーションを可能にする使用オペレーションにのみアクセスする必要がありますGetActiveBranchKeyGetBranchKeyVersionGetBeaconKey。使用するブランチキーを作成または管理するためのアクセス許可は必要ありません。

キーストアアクションが静的に設定されている場合、または検出用に設定されている場合、使用操作を実行できます。キーストアアクションが検出用に設定されている場合、管理者オペレーション (`CreateKey` および `VersionKey`) を実行することはできません。

ブランチキーストア管理者がブランチキーストアに複数の KMS キーを許可リストに登録した場合は、階層キーリングが複数の KMS キーを使用できるように、キーストアユーザーがキーストアアクションを検出用に設定することをお勧めします。

キーストアを作成する

ブランチキーを作成したりAWS KMS 、階層キーリングを使用する前に、ブランチキーを管理および保護する Amazon DynamoDB テーブルであるキーストアを作成する必要があります。

> **Important**
>
> ブランチキーを保持する DynamoDB テーブルを削除しないでください。このテーブルを削除すると、階層キーリングを使用して暗号化されたデータを復号できなくなります。

パーティションキーとソートキーに必要な次の文字列値を使用して、Amazon DynamoDB デベロッパーガイドの「テーブルの作成」の手順に従います。

	パーティションキー	ソートキー
ベーステーブル	`branch-key-id`	`type`

論理キーストア名

キーストアとして機能する DynamoDB テーブルに名前を付けるときは、キーストアアクションを設定するときに指定する論理キーストア名を慎重に検討することが重要です。論理キーストア名はキーストアの識別子として機能し、最初のユーザーが最初に定義した後は変更できません。キーストアアクションでは、常に同じ論理キーストア名を指定する必要があります。

DynamoDB テーブル名と論理キーストア名の間には one-to-one のマッピングが必要です。論理キーストア名は、DynamoDB の復元オペレーションを簡素化するために、テーブルに格納されているすべてのデータに暗号的にバインドされます。論理キーストア名は DynamoDB テーブル名とは異な

る場合がありますが、DynamoDB テーブル名を論理キーストア名として指定することを強くお勧めします。バックアップから DynamoDB テーブルを復元した後にテーブル名が変更された場合、論理キーストア名を新しい DynamoDB テーブル名にマッピングして、階層キーリングが引き続きキーストアにアクセスできるようにすることができます。

論理キーストア名に機密情報や機密情報を含めないでください。論理キーストア名は、 AWS KMS CloudTrail イベントでプレーンテキストで として表示されますtablename。

次の手順

1. the section called “キーストアアクションを設定する”
2. the section called “ブランチキーを作成する”
3. AWS KMS 階層キーリングを作成する

キーストアアクションを設定する

キーストアアクションは、ユーザーが実行できるオペレーションと、 AWS KMS その階層キーリングがキーストアに許可リストされている KMS キーをどのように使用するかを決定します。 AWS Database Encryption SDK は、次のキーストアアクション設定をサポートしています。

静的

キーストアを静的に設定すると、キーストアは、キーストアアクションを設定するkmsConfigurationときに で指定した KMS キー ARN に関連付けられた KMS キーのみを使用できます。ブランチキーの作成、バージョニング、または取得時に別の KMS キー ARN が発生した場合、例外がスローされます。

でマルチリージョン KMS キーを指定できますがkmsConfiguration、リージョンを含むキーの ARN 全体が KMS キーから派生したブランチキーに保持されます。別のリージョンでキーを指定することはできません。値を一致させるには、まったく同じマルチリージョンキーを指定する必要があります。

キーストアアクションを静的に設定すると、使用オペレーション (GetActiveBranchKey、GetBranchKeyVersion、GetBeaconKey) と管理オペレーション (CreateKey および) を実行できますVersionKey。 CreateKeyは、キーストア許可リストに新しい KMS キー ARN を追加できる特権オペレーションです。この KMS キーは、新しいアクティブなブランチキーを作成できます。KMS キーがキーストアに追加されると、削除できないため、このオペレーションへのアクセスを制限することをお勧めします。

発見

検出用にキーストアアクションを設定すると、キーストアはキーストアで許可リストに登録されている任意の AWS KMS key ARN を使用できます。ただし、マルチリージョン KMS キーが発生し、キーの ARN のリージョンが使用されている AWS KMS クライアントのリージョンと一致しない場合、例外がスローされます。

検出用にキーストアを設定する場合、 `CreateKey`や などの管理オペレーションを実行することはできません`VersionKey`。暗号化、復号、署名、検証オペレーションを有効にする使用オペレーションのみを実行できます。詳細については、「the section called “最小特権のアクセス許可の実装”」を参照してください。

キーストアアクションを設定する

キーストアアクションを設定する前に、次の前提条件を満たしていることを確認してください。

- 実行する必要があるオペレーションを決定します。詳細については、「the section called “最小特権のアクセス許可の実装”」を参照してください。
- 論理キーストア名を選択する

  DynamoDB テーブル名と論理キーストア名の間には one-to-one のマッピングが必要です。論理キーストア名は、DynamoDB 復元オペレーションを簡素化するために、テーブルに保存されているすべてのデータに暗号的にバインドされます。最初のユーザーが最初に定義した後は変更できません。キーストアアクションでは、常に同じ論理キーストア名を指定する必要があります。詳細については、「logical key store name」を参照してください。

静的設定

次の例では、キーストアアクションを静的に設定します。キーストアとして機能する DynamoDB テーブルの名前、キーストアの論理名、対称暗号化 KMS キーを識別する KMS キー ARN を指定する必要があります。

> **Note**
>
> キーストアサービスを静的に設定するときは、指定した KMS キー ARN を慎重に検討してください。`CreateKey` オペレーションは、KMS キー ARN をブランチキーストアの許可リストに追加します。KMS キーがブランチキーストアに追加されると、削除することはできません。

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)?;
```

検出の設定

次の の例では、検出用のキーストアアクションを設定します。キーストアとして機能する DynamoDB テーブルの名前と論理キーストア名を指定する必要があります。

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()?;
```

アクティブなブランチキーを作成する

ブランチキーは、 AWS KMS 階層キーリング AWS KMS key が呼び出しの数を減らすために使用する から派生したデータキーです AWS KMS。アクティブなブランチキーは、ブランチキーの最新バージョンです。階層キーリングは、暗号化リクエストごとに一意のデータキーを生成し、アクティブなブランチキーから派生した一意のラッピングキーを使用して各データキーを暗号化します。

新しいアクティブなブランチキーを作成するには、キーストアアクションを静的に設定する必要があります。 `CreateKey`は、キーストアアクション設定で指定された KMS キー ARN をキーストア許可リストに追加する特権オペレーションです。次に、KMS キーを使用して新しいアクティブなブランチキーを生成します。KMS キーがキーストアに追加されると、削除できないため、このオペレーションへのアクセスを制限することをお勧めします。

アプリケーションのコントロールプレーンの KeyStore Admin インターフェイスを介して `CreateKey`オペレーションを使用することをお勧めします。このアプローチは、キー管理のベストプラクティスと一致しています。

データプレーンにブランチキーを作成しないでください。この方法により、次のような結果になる可能性があります。

- への不要な呼び出し AWS KMS
- 同時実行性の高い環境で AWS KMS の への複数の同時呼び出し
- バッキング DynamoDB テーブルへの複数の TransactWriteItems 呼び出し。

`CreateKey` オペレーションには、既存のブランチキーが上書きされないように、`TransactWriteItems`呼び出しに条件チェックが含まれます。ただし、データプレーンでキーを作成すると、リソースの非効率的な使用やパフォーマンスの問題が発生する可能性があります。

キーストアで 1 つの KMS キーを許可リストに登録することも、キーストアアクション設定で指定した KMS キー ARN を更新して`CreateKey`再度 を呼び出すことで、複数の KMS キーを許可リストに登録することもできます。複数の KMS キーを許可リストに登録する場合、キーストアユーザーは、アクセスできるキーストアで許可リストに登録された任意のキーを使用できるように、キーストアアクションを検出用に設定する必要があります。詳細については、「the section called “キーストアアクションを設定する”」を参照してください。

必要な アクセス許可

ブランチキーを作成するには、キーストアアクションで指定された KMS キーに対する kms:GenerateDataKeyWithoutPlaintext および kms:ReEncrypt アクセス許可が必要です。

ブランチキーを作成する

次のオペレーションでは、キーストアアクション設定で指定した KMS キーを使用して新しいアクティブなブランチキーを作成し、キーストアとして機能する DynamoDB テーブルにアクティブなブランチキーを追加します。

CreateKey を呼び出す際に、次のオプションの値を指定することを選択できます。

- branchKeyIdentifier: カスタム branch-key-id を定義します。

  カスタム branch-key-id を作成するには、encryptionContext パラメータに追加の暗号化コンテキストを含める必要もあります。
- encryptionContext: は、kms:GenerateDataKeyWithoutPlaintext 呼び出しに含まれる暗号化コンテキストで追加の認証データ (AAD) を提供する、シークレット以外のキーと値のペアのオプションセットを定義します。

  この追加の暗号化コンテキストは aws-crypto-ec: プレフィックスとともに表示されます。

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();
```

まず、CreateKey オペレーションにより次の値が生成されます。

- branch-key-id のバージョン 4 Universally Unique Identifier (UUID) (カスタム branch-key-id を指定した場合を除く)。
- ブランチキーバージョンのバージョン 4 UUID
- ISO 8601 の日時形式の timestamp (協定世界時 (UTC))。

その後、CreateKey オペレーションは次のリクエストを使用して kms:GenerateDataKeyWithoutPlaintext を呼び出します。

```
{
     "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**
>
> 検索可能な暗号化のためにデータベースを設定していない場合でも、CreateKey オペレーションはアクティブなブランチキーとビーコンキーを作成します。両方のキーはキーストアに保存されます。詳細については、「検索可能な暗号化のための階層キーリングの使用」を参照してください。

次に、CreateKey オペレーションは kms:ReEncrypt を呼び出し、暗号化コンテキストを更新してブランチキーのアクティブレコードを作成します。

最後に、CreateKey オペレーションは ddb:TransactWriteItems を呼び出して、ステップ 2 で作成したテーブルにブランチキーを永続化する新しい項目を書き込みます。項目には次の属性があります。

```
{
      "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"
 }
```

アクティブなブランチキーをローテーションする

各ブランチキーのために一度に存在できるアクティブなバージョンは 1 つだけです。通常、アクティブな各ブランチキーバージョンは、複数のリクエストを満たすために使用されます。ただし、ユーザーがアクティブなブランチキーを再利用する範囲を制御し、アクティブなブランチキーをローテーションする頻度を決定します。

ブランチキーは、プレーンテキストデータキーの暗号化には使用されません。これらは、プレーンテキストデータキーを暗号化する一意のラッピングキーを導出するために使用されます。ラッピングキー導出プロセスでは、28 バイトのランダム性を備えた一意の 32 バイトのラッピングキーが生成されます。これは、暗号の摩耗が発生する前に、ブランチキーが 7 穣 9 秭、つまり 2^{96} を超える

一意のラッピングキーを導出できることを意味します。このように枯渇するリスクは極めて低いものの、ビジネスルールや契約、政府の規制により、アクティブなブランチキーのローテーションが必要になる場合があります。

ブランチキーのアクティブなバージョンは、ローテーションされるまでアクティブなままとなります。以前のバージョンのアクティブなブランチキーは、暗号化オペレーションの実行には使用されず、新しいラッピングキーの取得には使用できませんが、引き続きクエリを実行し、アクティブ中に暗号化したデータキーを復号するためのラッピングキーを提供できます。

> **Warning**
>
> テスト環境でのブランチキーの削除は元に戻せません。削除されたブランチキーは復元できません。テスト環境で同じ ID のブランチキーを削除して再作成すると、次の問題が発生する可能性があります。
>
> - 以前のテスト実行のマテリアルはキャッシュに残る可能性があります
> - 一部のテストホストまたはスレッドは、削除されたブランチキーを使用してデータを暗号化する場合があります
> - 削除されたブランチで暗号化されたデータは復号できません
>
> 統合テストで暗号化の失敗を防ぐには:
>
> - 新しいブランチキーを作成する前に階層キーリングリファレンスをリセットする、または
> - テストごとに一意のブランチキー IDs

必要なアクセス許可

ブランチキーをローテーションするには、キーストアアクションで指定された KMS キーに対する kms:GenerateDataKeyWithoutPlaintext および kms:ReEncrypt アクセス許可が必要です。

アクティブなブランチキーをローテーションする

`VersionKey` オペレーションを使用して、アクティブなブランチキーをローテーションします。アクティブなブランチキーをローテーションすると、以前のバージョンを置き換えるために新しいブランチキーが作成されます。アクティブなブランチキーをローテーションしても、`branch-key-id` は変わりません。`VersionKey` を呼び出す際に、現在アクティブなブランチキーを識別する `branch-key-id` を指定する必要があります。

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?;
```

キーリング

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK は、キーリングを使用してエンベロープ暗号化を実行します。データキーの生成、暗号化、復号は、キーリングによって行われます。キーリングは、暗号化された各レコードを保護する一意のデータキーのソースと、そのデータキーを暗号化するラッピングキーを決定します。キーリングは暗号化時に指定し、復号時には同じキーリングか別のキーリングを指定します。

各キーリングを個別に使用するか、キーリングを組み合わせてマルチキーリングにすることができます。ほとんどのキーリングではデータキーを生成、暗号化、および復号することができますが、特定のオペレーションを 1 つのみ実行するキーリング (例: データキーのみを生成するキーリング) を作成し、他のキーリングと組み合わせて使用することができます。

ラッピングキーを保護し、 AWS Key Management Service (AWS KMS) を暗号化しないままに AWS KMS keys しない を使用するキーリングなど、安全な境界内で暗号化オペレーションを実行する AWS KMS キーリングを使用することをお勧めします。また、ハードウェアセキュリティモジュール (HSM) に保存されているラッピングキーや他のマスターキーサービスによって保護されているラッピングキーを使用するキーリングを作成することもできます。

キーリングは、データキー、そして最終的にはデータを保護するラッピングキーを決定します。タスクに実用的で、最も安全なラッピングキーを使用してください。可能な場合は常に、ハードウェアセキュリティモジュール (HSM) またはキー管理インフラストラクチャ (AWS Key Management Service (AWS KMS) の KMS キーや AWS CloudHSM の暗号化キーなど) によって保護されたラッピングキーを使用してください。

AWS Database Encryption SDK には複数のキーリングとキーリング設定が用意されており、独自のカスタムキーリングを作成できます。同じタイプまたは異なるタイプの 1 つ以上のキーリングを含むマルチキーリングを作成することもできます。

トピック

- キーリングのしくみ
- AWS KMS キーリング

- AWS KMS 階層キーリング
- AWS KMS ECDH キーリング
- Raw AES キーリング
- Raw RSA キーリング
- Raw ECDH キーリング
- マルチキーリング

キーリングのしくみ

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

データベース内のフィールドを暗号化して署名すると、 AWS Database Encryption SDK はキーリングに暗号化マテリアルを要求します。キーリングは、プレーンテキストのデータキー、キーリング内の各ラッピングキーによって暗号化されたデータキーのコピー、およびデータキーに関連付けられた MAC キーを返します。 AWS Database Encryption SDK は、プレーンテキストキーを使用してデータを暗号化し、できるだけ早くプレーンテキストのデータキーをメモリから削除します。その後、 AWS Database Encryption SDK は、暗号化されたデータキーと、暗号化や署名の指示などの他の情報を含むマテリアルの説明を追加します。 AWS Database Encryption SDK は MAC キーを使用して、マテリアルの説明と `ENCRYPT_AND_SIGN`または とマークされたすべてのフィールドの正規化に関するハッシュベースのメッセージ認証コード (HMACs) を計算します`SIGN_ONLY`。

データを復号する際には、データの暗号化に使用したのと同じキーリングを使用することも、別のキーリングを使用することもできます。データを復号するには、復号キーリングが暗号化キーリング内の少なくとも 1 つのラッピングキーにアクセスできる必要があります。

AWS Database Encryption SDK は、暗号化されたデータキーをマテリアルの説明からキーリングに渡し、キーリングのいずれかを復号するよう求めます。キーリングは、ラッピングキーを使用して暗号化されたデータキーのいずれかを復号し、プレーンテキストのデータキーを返します。 AWS Database Encryption SDK は、プレーンテキストデータキーを使用してデータを復号します。キーリングのラッピングキーのいずれも暗号化されたデータキーを復号できない場合は、復号は失敗します。

単一のキーリングを使用するか、同じタイプまたは異なるタイプのキーリングを組み合わせてマルチキーリングにすることもできます。データを暗号化すると、マルチキーリングは、マルチキーリングを構成するすべてのキーリング内のすべてのラッピングキーによって暗号化されたデータキーのコピーと、そのデータキーに関連付けられた MAC キーを返します。データは、マルチキーリングのラッピングキーのいずれかを持つキーリングを使用して復号できます。

AWS KMS キーリング

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS KMS キーリングは、対称暗号化または非対称 RSA AWS KMS keysを使用してデータキーを生成、暗号化、復号します。 AWS Key Management Service (AWS KMS) は KMS キーを保護し、FIPS 境界内で暗号化オペレーションを実行します。可能な限り、 AWS KMS キーリング、または同様のセキュリティプロパティを持つキーリングを使用することをお勧めします。

キーリングで対称マルチリージョン KMS AWS KMS キーを使用することもできます。マルチリージョンを使用した詳細と例については AWS KMS keys、「」を参照してくださいマルチリージョンの使用 AWS KMS keys。マルチリージョンキーの詳細については、「AWS Key Management Service デベロッパーガイド」の「マルチリージョンキーを使用する」を参照してください。

AWS KMS キーリングには、次の 2 種類のラッピングキーを含めることができます。

- ジェネレーターキー: プレーンテキストのデータキーを生成し、暗号化します。データを暗号化するキーリングには、ジェネレーターキーが 1 つ必要です。
- 追加キー: ジェネレーターキーが生成したプレーンテキストのデータキーを暗号化します。 AWS KMS キーリングには 0 個以上の追加キーを含めることができます。

レコードを暗号化するにはジェネレーターキーが必要です。 AWS KMS キーリングに AWS KMS キーが 1 つしかない場合、そのキーはデータキーの生成と暗号化に使用されます。

すべてのキーリングと同様に、 AWS KMS キーリングは個別に使用することも、同じタイプまたは異なるタイプの他のキーリングを持つマルチキーリングでも使用できます。

トピック

- AWS KMS キーリングに必要なアクセス許可

- AWS KMS キーリング AWS KMS keys での の識別
- AWS KMS キーリングの作成
- マルチリージョンの使用 AWS KMS keys
- AWS KMS 検出キーリングの使用
- AWS KMS リージョン検出キーリングの使用

AWS KMS キーリングに必要なアクセス許可

AWS Database Encryption SDK は を必要とせず AWS アカウント 、 に依存しません AWS のサービス。ただし、 AWS KMS キーリングを使用するには、 AWS アカウント と、キーリング AWS KMS keys の に対する以下の最小限のアクセス許可が必要です。

- AWS KMS キーリングで暗号化するには、ジェネレーターキーに対する kms:GenerateDataKey アクセス許可が必要です。 AWS KMS キーリングのすべての追加キーに対する kms:Encrypt アクセス許可が必要です。
- AWS KMS キーリングで復号するには、 AWS KMS キーリング内の少なくとも 1 つのキーに対する kms:Decrypt アクセス許可が必要です。
- AWS KMS キーリングで構成されるマルチキーリングで暗号化するには、ジェネレーターキーリングのジェネレーターキーに対する kms:GenerateDataKey アクセス許可が必要です。他のすべてのキーリングの他のすべての AWS KMS キーに対する kms:Encrypt アクセス許可が必要です。
- 非対称 RSA AWS KMS キーリングで暗号化するには、キーリングの作成時に暗号化に使用するパブリックキーマテリアルを指定する必要があるため、kms:GenerateDataKey または kms:Encrypt は必要ありません。このキーリングで暗号化する場合、 AWS KMS 呼び出しは行われません。非対称 RSA AWS KMS キーリングで復号するには、kms:Decrypt アクセス許可が必要です。

のアクセス許可の詳細については AWS KMS keys、「 AWS Key Management Service デベロッパーガイド」の「認証とアクセスコントロール」を参照してください。

AWS KMS キーリング AWS KMS keys での の識別

AWS KMS キーリングには 1 つ以上の を含めることができます AWS KMS keys。 AWS KMS キーリングで を指定する AWS KMS key には、サポートされている AWS KMS キー識別子を使用します。キーリング AWS KMS key 内の を識別するために使用できるキー識別子は、 オペレーションと言語の実装によって異なります。 AWS KMS keyのキー識別子の詳細については、AWS Key Management Service デベロッパーガイドの「キー識別子」を参照してください。

ベストプラクティスとして、自らのタスクにとって実用的である最も具体的なキー識別子を使用します。

- AWS KMS キーリングで暗号化するには、キー ID、キー ARN、エイリアス名、またはエイリアス ARN を使用してデータを暗号化できます。

  > **Note**
  >
  > 暗号化キーリングで KMS キーのエイリアス名またはエイリアス ARN を指定すると、暗号化オペレーションによって、現在エイリアスに関連付けられているキー ARN が、暗号化されたデータキーのメタデータに保存されます。エイリアスは保存されません。エイリアスの変更は、暗号化されたデータキーの復号に使用される KMS キーには影響しません。

- AWS KMS キーリングで復号するには、キー ARN を使用して を識別する必要があります AWS KMS keys。詳細については、「ラッピングキーの選択」を参照してください。
- 暗号化および復号に使用するキーリングでは、キー ARN を使用して AWS KMS keysを指定する必要があります。

復号時に、 AWS Database Encryption SDK は、暗号化されたデータ AWS KMS キーの 1 つを復号 AWS KMS key できる のキーリングを検索します。具体的には、 AWS Database Encryption SDK は、マテリアルの説明で暗号化されたデータキーごとに次のパターンを使用します。

- AWS Database Encryption SDK は、マテリアル説明のメタデータからデータキーを暗号化 AWS KMS key した のキー ARN を取得します。
- AWS Database Encryption SDK は、キー ARN AWS KMS key が一致する の復号キーリングを検索します。
- キーリングで一致するキー ARN AWS KMS key を持つ が見つかった場合、 AWS Database Encryption SDK は KMS キーを使用して暗号化されたデータキーを復号するように AWS KMS に要求します。
- それ以外の場合は、暗号化された次のデータキーに進みます (ある場合)。

AWS KMS キーリングの作成

各 AWS KMS キーリングは、同じ AWS KMS key または異なる AWS アカウント および AWS KMS keys の 1 つまたは複数の で設定できます AWS リージョン。 AWS KMS key は、対称暗号化キー (SYMMETRIC_DEFAULT) または非対称 RSA KMS キーである必要があります。対称暗号化マルチ

リージョン KMS キーを使用することもできます。マルチ AWS KMS キーリングでは、1 つ以上のキーリングを使用できます。 ???

データを暗号化および復号する AWS KMS キーリングを作成することも、暗号化または復号専用の AWS KMS キーリングを作成することもできます。データを暗号化する AWS KMS キーリングを作成するときは、ジェネレーターキーを指定する必要があります。ジェネレーターキー AWS KMS key は、プレーンテキストのデータキーを生成して暗号化するために使用される です。データキーは数学的には KMS キーとは無関係です。次に、選択した場合は、同じプレーンテキストのデータキーを暗号化 AWS KMS keys する追加の を指定できます。このキーリングで保護された暗号化されたフィールドを復号するには、使用する復号キーリングに、キーリングで AWS KMS keys 定義されたの少なくとも 1 つが含まれているか、含まれていない必要があります AWS KMS keys。(を使用しない AWS KMS キーリング AWS KMS keys は、AWS KMS 検出キーリングと呼ばれます)。

暗号化キーリングまたはマルチキーリング内のすべてのラッピングキーは、データキーを暗号化できる必要があります。いずれかのラッピングキーが暗号化に失敗すると、暗号化メソッドは失敗します。そのため、呼び出し元は、キーリング内のすべてのキーについて必要な許可を持っている必要があります。検出キーリングを使用して、単独またはマルチキーリングでデータを暗号化すると、暗号化操作は失敗します。

次の例では、 `CreateAwsKmsMrkMultiKeyring`メソッドを使用して、対称暗号化 KMS AWS KMS キーを持つ キーリングを作成します。`CreateAwsKmsMrkMultiKeyring` メソッドは自動的に AWS KMS クライアントを作成し、キーリングが単一リージョンキーとマルチリージョンキーの両方を正しく処理するようにします。これらの例では、キー ARNs を使用して KMS キーを識別します。詳細については、「AWS KMS キーリング AWS KMS keys での の識別」を参照してください。

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?;
```

次の例では、 CreateAwsKmsRsaKeyringメソッドを使用して、非対称 RSA KMS AWS KMS キーを持つ キーリングを作成します。非対称 RSA AWS KMS キーリングを作成するには、次の値を指定します。

- kmsClient: 新しい AWS KMS クライアントを作成する
- kmsKeyID: 非対称 RSA KMS キーを識別するキー ARN
- publicKey: 渡したキーのパブリックキーを表す UTF-8 エンコードされた PEM ファイルの ByteBuffer kmsKeyID
- encryptionAlgorithm: 暗号化アルゴリズムは RSAES_OAEP_SHA_256または である必要があります 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?;
```

マルチリージョンの使用 AWS KMS keys

マルチリージョンを AWS Database Encryption SDK のラッピングキー AWS KMS keys として使用できます。1 つの でマルチリージョンキーを使用して暗号化する場合 AWS リージョン、別の で関連するマルチリージョンキーを使用して復号できます AWS リージョン。

マルチリージョン KMS キーは、同じキーマテリアルとキー ID AWS リージョン を持つ異なる AWS KMS keys の のセットです。これらの関連キーは、さまざまなリージョンで同じキーであるかのように使用できます。マルチリージョンキーは、クロスリージョン呼び出しを行うことなく、あるリー

ジョンで暗号化し、別のリージョンで復号する必要がある一般的なディザスタリカバリおよびバックアップシナリオをサポートします AWS KMS。マルチリージョンキーの詳細については、「AWS Key Management Service デベロッパーガイド」の「マルチリージョンキーを使用する」を参照してください。

マルチリージョンキーをサポートするために、 AWS Database Encryption SDK には AWS KMS multi-Region-awareキーリングが含まれています。`CreateAwsKmsMrkMultiKeyring` メソッドは、単一リージョンキーとマルチリージョンキーの両方をサポートします。

- 単一リージョンキーの場合、マルチリージョン対応シンボルは、単一リージョン AWS KMS キーリングのように動作します。データを暗号化した単一リージョンキーを使用してのみ、暗号化テキストの復号が試されます。 AWS KMS キーリングエクスペリエンスを簡素化するには、対称暗号化 KMS キーを使用するたびに `CreateAwsKmsMrkMultiKeyring`メソッドを使用することをお勧めします。
- マルチリージョンキーの場合、マルチリージョン対応シンボルは、データを暗号化したのと同じマルチリージョンキー、または指定したリージョン内の関連するマルチリージョンキーを使用して暗号文の復号を試みます。

複数の KMS キーを使用するマルチリージョン対応キーリングでは、複数の単一リージョンキーとマルチリージョンキーを指定できます。ただし、関連するマルチリージョンキーのセットごとに 1 つのキーしか指定できません。同じキー ID で複数のキー識別子を指定すると、コンストラクタの呼び出しは失敗します。

次の例では、マルチリージョン KMS AWS KMS キーを使用して キーリングを作成します。この例では、ジェネレーターキーとしてマルチリージョンキーを指定し、子キーとして単一リージョンキーを指定します。

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?;
```

マルチリージョン AWS KMS キーリングを使用すると、暗号文を strict モードまたは discover モードで復号できます。厳格モードで暗号文を復号するには、暗号文を復号するリージョン内の関連するマルチリージョンキーのキー ARN を使用してマルチリージョン対応シンボルをインスタンス化します。別のリージョン (例: レコードが暗号化されたリージョン) で関連するマルチリージョンキーのキー ARN を指定した場合、マルチリージョン対応シンボルは、その AWS KMS keyのクロスリージョン呼び出しを実行します。

Strict モードで復号する場合、マルチリージョン対応シンボルにはキー ARN が必要です。関連するマルチリージョンキーの各セットからキー ARN を 1 つだけ受け付けます。

AWS KMS マルチリージョンのキーを使用して、検出モードで復号することもできます。検出モードで復号する場合は、 AWS KMS keysを指定しません。(単一リージョン AWS KMS の検出キーリングの詳細については、「」を参照してくださいAWS KMS 検出キーリングの使用。)

マルチリージョンキーで暗号化した場合、検出モードのマルチリージョン対応シンボルは、ローカルリージョン内の関連するマルチリージョンキーを使用して復号しようとします。何も存在しない場

合、呼び出しは失敗します。検出モードでは、 AWS Database Encryption SDK は暗号化に使用されるマルチリージョンキーのクロスリージョン呼び出しを試みません。

AWS KMS 検出キーリングの使用

復号するときは、 AWS Database Encryption SDK が使用できるラッピングキーを指定することがベストプラクティスです。このベストプラクティスに従うには、ラ AWS KMS ッピングキーを AWS KMS 指定したキーに制限する復号キーリングを使用します。ただし、AWS KMS 検出キーリング、つまりラッピングキーを指定しない AWS KMS キーリングを作成することもできます。

AWS Database Encryption SDK は、標準の AWS KMS 検出キーリングと AWS KMS マルチリージョンキー用の検出キーリングを提供します。 AWS Database Encryption SDK でのマルチリージョンキーの使用については、「マルチリージョンの使用 AWS KMS keys」を参照してください。

ラッピングキーが指定されていないため、検出キーリングはデータを暗号化できません。検出キーリングを使用して、単独またはマルチキーリングでデータを暗号化すると、暗号化操作は失敗します。

復号時に、検出キーリングを使用すると、 AWS Database Encryption SDK は、暗号化されたデータキーを所有またはアクセスできるユーザーに関係なく、暗号化されたデータキーを暗号化 AWS KMS key された を使用して復号するように AWS KMS に要求できます AWS KMS key。呼び出しは、呼び出し元にその AWS KMS keyに対する `kms:Decrypt` 許可がある場合にのみ成功します。

> **Important**
>
> 復号マルチキーリングに AWS KMS 検出キーリングを含めると、検出キーリングはマルチキーリングの他のキーリングで指定されたすべての KMS キー制限を上書きします。マルチキーリングは、最も制限の少ないキーリングのように動作します。検出キーリングを単独または複数のキーリングで使用してデータを暗号化すると、暗号化オペレーションは失敗します

AWS Database Encryption SDK は、便利な AWS KMS 検出キーリングを提供します。ただし、次の理由から、可能な限り制限されたキーリングを使用することをお勧めします。

- 真正性 – AWS KMS 検出キーリングは、呼び出し元 AWS KMS key に復号化に使用するアクセス許可がある限り、マテリアル説明のデータキーの暗号化に使用された任意の AWS KMS key を使用できます。これは、発信者 AWS KMS key が使用する ではない場合があります。たとえば、暗号化されたデータキーの 1 つが、誰でも使用できる安全性 AWS KMS key の低い で暗号化されている可能性があります。

- レイテンシーとパフォーマンス – AWS Database AWS KMS Encryption SDK は、他の AWS アカウント およびリージョン AWS KMS keys の によって暗号化されたデータキーを含む、暗号化されたすべてのデータキーを復号しようとし、呼び出し元 AWS KMS keys に復号に使用するアクセス許可がないため、検出キーリングは他のキーリングよりもかなり遅くなる可能性があります。

検出キーリングを使用する場合は、検出フィルターを使用して、指定 AWS アカウント およびパーティションで使用できる KMS キーを制限することをお勧めします。アカウント ID とパーティションの検索については、のAWS アカウント 「識別子と ARN 形式」を参照してくださいAWS 全般のリファレンス。 https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html#arns-syntax

次のコード例では、 AWS Database Encryption SDK が使用できる KMS AWS KMS キーをawsパーティションと111122223333サンプルアカウントのキーに制限する検出フィルターを使用して、検出キーリングをインスタンス化します。

このコードを使用する前に、例 AWS アカウント とパーティションの値を AWS アカウント およびパーティションの有効な値に置き換えます。KMS キーが中国リージョンにある場合は、aws-cn のパーティションの値を使用します。KMS キーが AWS GovCloud (US) Regionsにある場合は、aws-us-gov のパーティションの値を使用します。他のすべての AWS リージョンについては、aws のパーティションの値を使用します。

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 リージョン検出キーリングの使用

AWS KMS リージョンレベルの検出キーリングは、KMS キーの ARN を指定しないキーリングです。代わりに、 AWS Database Encryption SDK は、特に KMS キーのみを使用して復号化できます AWS リージョン。

AWS KMS リージョン検出キーリングを使用して復号する場合、 AWS Database Encryption SDK は、指定された AWS KMS key の で暗号化された暗号化されたデータキーを復号します AWS リージョン。成功するには、発信者がデータキーを暗号化 AWS リージョン した指定された AWS KMS keys 内の少なくとも 1 つの に対する`kms:Decrypt`アクセス許可を持っている必要があります。

他の検出キーリングと同様、リージョンレベルの検出キーリングは暗号化には影響しません。暗号化されたフィールドを復号する場合にのみ機能します。暗号化と復号に使用されるマルチキーリングで

リージョンレベルの検出キーリングを使用する場合、それは復号時にのみ有効です。マルチリージョン検出キーリングを単独または複数のキーリングで使用してデータを暗号化すると、暗号化オペレーションは失敗します。

> **Important**
>
> 復号マルチキーリングに AWS KMS リージョン検出キーリングを含めると、リージョン検出キーリングは、マルチキーリングの他のキーリングで指定されたすべての KMS キー制限を上書きします。マルチキーリングは、最も制限の少ないキーリングのように動作します。AWS KMS 検出キーリングは、単独で使用する場合も、マルチキーリングで使用する場合も、暗号化には影響しません。

AWS Database Encryption SDK のリージョン検出キーリングは、指定されたリージョンの KMS キーでのみ復号を試みます。検出キーリングを使用する場合は、 AWS KMS クライアントでリージョンを設定します。これらの AWS Database Encryption SDK 実装では、リージョンごとに KMS キーをフィルタリングしませんが、指定されたリージョン外の KMS キーの復号リクエストは失敗 AWS KMS します。

検出キーリングを使用する場合は、検出フィルターを使用して、復号に使用される KMS キーを、指定された AWS アカウント および パーティション内のキーに制限することをお勧めします。

たとえば、次のコードは、検出フィルターを使用して AWS KMS リージョン検出キーリングを作成します。このキーリングは、 AWS Database Encryption SDK を米国西部 (オレゴン) リージョン (us-west-2) のアカウント 111122223333 の KMS キーに制限します。

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 階層キーリング

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

> **Note**
>
> 2023 年 7 月 24 日の時点では、デベロッパープレビュー中に作成されたブランチキーはサポートされていません。新しいブランチキーを作成して、デベロッパープレビュー中に作成したキーストアを引き続き使用します。

AWS KMS 階層キーリングを使用すると、レコードを暗号化または復号する AWS KMS たびに を呼び出すことなく、対称暗号化 KMS キーで暗号化マテリアルを保護できます。これは、 への呼び出しを最小限に抑える必要があるアプリケーションや AWS KMS、セキュリティ要件に違反することなく一部の暗号化マテリアルを再利用できるアプリケーションに適しています。

階層キーリングは、Amazon DynamoDB テーブルに保持されている AWS KMS 保護されたブランチキーを使用し、暗号化および復号オペレーションで使用されるブランチキーマテリアルをローカルにキャッシュすることで AWS KMS 、呼び出しの数を減らす暗号化マテリアルキャッシュソリューションです。DynamoDB テーブルは、ブランチキーを管理および保護するキーストアとして機能します。アクティブなブランチキーと、ブランチキーの以前のすべてのバージョンが格納されます。アクティブなブランチキーは、ブランチキーの最新バージョンです。階層キーリングは、暗号化リクエストごとに一意のデータ暗号化キーを使用し、アクティブなブランチキーから派生した一意のラッピングキーを使用して各データ暗号化キーを暗号化します。階層キーリングは、アクティブなブランチキーと、その導出ラッピングキーの間に確立された階層に依拠します。

階層キーリングは通常、複数のリクエストを満たすために各ブランチキーバージョンを使用します。ただし、ユーザーがアクティブなブランチキーを再利用する範囲を制御し、アクティブなブランチキーをローテーションする頻度を決定します。ブランチキーのアクティブなバージョンは、ローテーションされるまでアクティブなままとなります。アクティブなブランチキーの以前のバージョンは暗号化オペレーションの実行には使用されませんが、引き続きクエリを実行して復号オペレーションに使用できます。

階層キーリングをインスタンス化すると、ローカルキャッシュが作成されます。ブランチキーマテリアルがローカルキャッシュ内に格納される最大時間 (ブランチキーマテリアルが期限切れになって

キャッシュから削除されるまでの時間) を定義するキャッシュ制限を指定します。階層キーリングは 1 回の AWS KMS 呼び出しでブランチキーを復号し、 オペレーションで branch-key-idが初めて指定されたときにブランチキーマテリアルをアセンブルします。その後、ブランチキーマテリアルはローカルキャッシュに格納され、キャッシュ制限が期限切れになるまで、その branch-key-id を指定するすべての暗号化および復号オペレーションのために再利用されます。ブランチキーマテリアルをローカルキャッシュに保存すると、 AWS KMS 呼び出しが減ります。例えば、キャッシュ制限が 15 分である場合を考えてみましょう。そのキャッシュ制限内で 10,000 回の暗号化オペレーションを実行する場合、従来の AWS KMS キーリングは 10,000 回の暗号化オペレーションを満たすために 10,000 回の AWS KMS 呼び出しを行う必要があります。アクティブな が 1 つある場合branch-key-id、階層キーリングは 10,000 回の暗号化オペレーションを満たすために 1 回の AWS KMS 呼び出しを行うだけで済みます。

ローカルキャッシュは、暗号化マテリアルと復号マテリアルを分離します。暗号化マテリアルはアクティブなブランチキーからアセンブルされ、キャッシュ制限の有効期限が切れるまですべての暗号化オペレーションに再利用されます。復号マテリアルは、暗号化されたフィールドのメタデータで識別されるブランチキー ID とバージョンからアセンブルされ、キャッシュ制限の有効期限が切れるまで、ブランチキー ID とバージョンに関連するすべての復号オペレーションに再利用されます。ローカルキャッシュは、同じブランチキーの複数のバージョンを一度に保存できます。ローカルキャッシュが を使用するように設定されている場合branch key ID supplier、一度に複数のアクティブなブランチキーからのブランチキーマテリアルを保存することもできます。

> **Note**
>
> AWS Database Encryption SDK の階層キーリングに関するすべての言及は、 AWS KMS 階層キーリングを参照しています。

トピック

- 仕組み
- 前提条件
- 必要なアクセス許可
- キャッシュを選択する
- 階層キーリングを作成する
- 検索可能な暗号化のための階層キーリングの使用

仕組み

次のチュートリアルでは、階層キーリングが暗号化および復号マテリアルをアセンブルする方法と、暗号化および復号オペレーションのためにキーリングが実行するさまざまな呼び出しについて説明します。ラッピングキーの導出とプレーンテキストデータキーの暗号化プロセスの技術的な詳細については、「AWS KMS 階層キーリングの技術的な詳細」を参照してください。

暗号化および署名

次のチュートリアルでは、階層キーリングが暗号化マテリアルをアセンブルし、一意のラッピングキーを導出する方法について説明します。

1. 暗号化メソッドは、階層キーリングに暗号化マテリアルを要求します。キーリングはプレーンテキストのデータキーを生成し、ローカルキャッシュにラッピングキーを生成する有効なブランチキーマテリアルがあるかどうかをチェックします。有効なブランチキーマテリアルがある場合、キーリングはステップ 4 に進みます。
2. 有効なブランチキーマテリアルがない場合、階層キーリングはキーストアにアクティブなブランチキーをクエリします。

    a. キーストアは AWS KMS を呼び出してアクティブなブランチキーを復号し、プレーンテキストのアクティブなブランチキーを返します。アクティブなブランチキーを識別するデータは、 AWS KMSに対する復号呼び出しで追加認証データ (AAD) を提供するためにシリアル化されます。

    b. キーストアは、プレーンテキストのブランチキーと、ブランチキーのバージョンなど、それを識別するデータを返します。
3. 階層キーリングはブランチキーマテリアル (プレーンテキストブランチキーとブランチキーバージョン) をアセンブルし、それらのコピーをローカルキャッシュに格納します。
4. 階層キーリングは、プレーンテキストブランチキーと 16 バイトのランダムソルトから一意のラッピングキーを導出します。プレーンテキストデータキーのコピーを暗号化するために、導出されたラッピングキーを使用します。

暗号化メソッドは、暗号化マテリアルを使用してレコードを暗号化して署名します。 AWS Database Encryption SDK でレコードがどのように暗号化および署名されるのかに関する詳細については、「暗号化して署名」を参照してください。

復号および検証

次のチュートリアルでは、階層キーリングが復号マテリアルをアセンブルし、暗号化されたデータキーを復号する方法について説明します。

1. 復号メソッドは、暗号化されたレコードのマテリアルの説明フィールドから暗号化されたデータキーを識別し、それを階層キーリングに渡します。
2. 階層キーリングは、ブランチキーのバージョン、16 バイトのソルト、およびデータキーの暗号化方法を説明する他の情報を含む、暗号化されたデータキーを識別するデータを逆シリアル化します。

   詳細については、「AWS KMS 階層キーリングの技術的な詳細」を参照してください。
3. 階層キーリングは、ステップ 2 で特定されたブランチキーのバージョンと一致する有効なブランチキーマテリアルがローカルキャッシュ内に存在するかどうかをチェックします。有効なブランチキーマテリアルがある場合、キーリングはステップ 6 に進みます。
4. 有効なブランチキーマテリアルがない場合、階層キーリングは、ステップ 2 で識別されたブランチキーバージョンに一致するブランチキーについてキーストアをクエリします。

   a. キーストアは AWS KMS を呼び出してブランチキーを復号し、プレーンテキストのアクティブなブランチキーを返します。アクティブなブランチキーを識別するデータは、 AWS KMSに対する復号呼び出しで追加認証データ (AAD) を提供するためにシリアル化されます。

   b. キーストアは、プレーンテキストのブランチキーと、ブランチキーのバージョンなど、それを識別するデータを返します。
5. 階層キーリングはブランチキーマテリアル (プレーンテキストブランチキーとブランチキーバージョン) をアセンブルし、それらのコピーをローカルキャッシュに格納します。
6. 階層キーリングは、アセンブルされたブランチキーマテリアルと、ステップ 2 で識別された 16 バイトのソルトを使用して、データキーを暗号化した一意のラッピングキーを複製します。
7. 階層キーリングは、複製されたラッピングキーを使用してデータキーを復号し、プレーンテキストのデータキーを返します。

復号メソッドは、復号マテリアルとプレーンテキストデータキーを使用し、レコードを復号して検証します。 AWS Database Encryption SDK でのレコードの復号化と検証の詳細については、「復号化と検証」を参照してください。

前提条件

階層キーリングを作成して使用する前に、次の前提条件を満たしていることを確認してください。

- ユーザーまたはキーストア管理者がキーストアを作成し、少なくとも 1 つのアクティブなブランチキーを作成しました。
- キーストアアクションを設定しました。

> **Note**
>
> キーストアアクションの設定方法によって、実行できるオペレーションと、階層キーリングで使用できる KMS キーが決まります。詳細については、「キーストアアクション」を参照してください。

- キーストアキーとブランチキーにアクセスして使用するために必要な AWS KMS アクセス許可があります。詳細については、「the section called “必要なアクセス許可”」を参照してください。
- サポートされているキャッシュタイプを確認し、ニーズに最適なキャッシュタイプを設定しました。詳細については、the section called “キャッシュを選択する”を参照してください。

必要なアクセス許可

AWS Database Encryption SDK は を必要とせず AWS アカウント 、 に依存しません AWS のサービス。ただし、階層キーリングを使用するには、キーストアの対称暗号化 AWS KMS key(複数可) に対する AWS アカウント と以下の最小限のアクセス許可が必要です。

- 階層キーリングを使用してデータを暗号化および復号するには、kms:Decrypt が必要です。
- ブランチキーを作成してローテーションするには、kms:GenerateDataKeyWithoutPlaintext と kms:ReEncrypt が必要です。

ブランチキーとキーストアへのアクセスの制御の詳細については、「」を参照してくださいthe section called “最小特権のアクセス許可の実装”。

キャッシュを選択する

階層キーリングは、暗号化および復号オペレーションで使用されるブランチキーマテリアルをローカルにキャッシュ AWS KMS することで、 に対する呼び出しの数を減らします。階層キーリングを作成する前に、使用するキャッシュのタイプを決定する必要があります。デフォルトのキャッシュを使用するか、ニーズに合わせてキャッシュをカスタマイズできます。

階層キーリングは、次のキャッシュタイプをサポートしています。

- the section called “デフォルトキャッシュ”

- the section called “MultiThreadedキャッシュ”
- the section called “StormTracking キャッシュ”
- the section called “共有キャッシュ”

デフォルトキャッシュ

ほとんどのユーザーにとって、Default キャッシュはスレッド要件を満たします。Default キャッシュは、高度にマルチスレッド化されている環境をサポートするように設計されています。ブランチキーマテリアルエントリの有効期限が切れると、デフォルトキャッシュは、ブランチキーマテリアルエントリが 10 秒前に期限切れになることを 1 つのスレッドに通知 AWS KMS することで、複数のスレッドが呼び出されるのを防ぎます。これにより、1 つのスレッドのみが にリクエストを送信 AWS KMS してキャッシュを更新します。

デフォルトキャッシュと StormTracking キャッシュは同じスレッドモデルをサポートしますが、デフォルトキャッシュを使用するにはエントリ容量を指定するだけです。より詳細なキャッシュのカスタマイズを行うには、 を使用しますthe section called “StormTracking キャッシュ”。

ローカルキャッシュに保存できるブランチキーマテリアルエントリの数をカスタマイズする場合を除き、階層キーリングを作成するときにキャッシュタイプを指定する必要はありません。キャッシュタイプを指定しない場合、階層キーリングはデフォルトのキャッシュタイプを使用し、エントリ容量を 1000 に設定します。

デフォルトキャッシュをカスタマイズするには、次の値を指定します。

- エントリキャパシティ: ローカルキャッシュに格納できるブランチキーマテリアルのエントリの数を制限します。

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キャッシュ

MultiThreadedキャッシュはマルチスレッド環境で安全に使用できますが、 AWS KMS または Amazon DynamoDB 呼び出しを最小限に抑える機能はありません。その結果、ブランチキーマテリアルのエントリの期限が切れると、すべてのスレッドに同時に通知されます。これにより、キャッシュを更新するための複数の AWS KMS 呼び出しが発生する可能性があります。

MultiThreadedキャッシュを使用するには、次の値を指定します。

- エントリキャパシティ: ローカルキャッシュに格納できるブランチキーマテリアルのエントリの数を制限します。
- エントリのプルーニングテールのサイズ: エントリキャパシティに達した場合にプルーニングするエントリの数を定義します。

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 キャッシュ

StormTracking キャッシュは、高度にマルチスレッド化されている環境をサポートするように設計されています。ブランチキーマテリアルエントリの有効期限が切れると、StormTracking キャッシュは、ブランチキーマテリアルエントリの有効期限が切れることを 1 つのスレッドに通知 AWS KMS することで、複数のスレッドが呼び出されるのを防ぎます。これにより、1 つのスレッドのみが にリクエストを送信 AWS KMS してキャッシュを更新します。

StormTracking キャッシュを使用するには、次の値を指定します。

- エントリキャパシティ: ローカルキャッシュに格納できるブランチキーマテリアルのエントリの数を制限します。

  デフォルト値: 1000 エントリ
- エントリのプルーニングテールのサイズ: 一度にプルーニングするブランチキーマテリアルのエントリの数を定義します。

  デフォルトの値: 1 個のエントリ
- 猶予期間: 期限が切れる前にブランチキーマテリアルの更新を試行する秒数を定義します。

  デフォルト値: 10 秒
- 猶予間隔: ブランチキーマテリアルの更新が試行される間隔の秒数を定義します。

  デフォルト値: 1 秒
- ファンアウト: ブランチキーマテリアルの更新の同時試行が可能な回数を定義します。

デフォルトの値: 20 回の試行

- 処理中の Time To Live (TTL): ブランチキーマテリアルの更新の試行がタイムアウトするまでの秒数を定義します。キャッシュが GetCacheEntry に応答して NoSuchEntry を返すたびに、同じキーが PutCache エントリを使用して書き込まれるまで、そのブランチキーは処理中であるとみなされます。

  デフォルト値: 10 秒

- スリープ: fanOut を超えた場合にスレッドがスリープする秒数を定義します。

  デフォルトの値: 20 ミリ秒

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()?)
```

共有キャッシュ

デフォルトでは、階層キーリングは、キーリングをインスタンス化するたびに新しいローカルキャッシュを作成します。ただし、共有キャッシュを使用すると、複数の階層キーリング間でキャッシュを共有できるため、メモリを節約できます。インスタンス化する階層キーリングごとに新しい暗号化マテリアルキャッシュを作成するのではなく、共有キャッシュは 1 つのキャッシュのみをメモリに保存します。このキャッシュは、それを参照するすべての階層キーリングで使用できます。共有キャッシュは、キーリング間での暗号化マテリアルの重複を回避することで、メモリ使用量を最適化するのに役立ちます。代わりに、階層キーリングは同じ基盤となるキャッシュにアクセスし、全体的なメモリフットプリントを削減できます。

共有キャッシュを作成する場合でも、キャッシュタイプを定義します。キャッシュタイプthe section called “StormTracking キャッシュ”として the section called “デフォルトキャッシュ”、the section called “MultiThreadedキャッシュ”、または を指定することも、互換性のあるカスタムキャッシュを置き換えることもできます。

パーティション

複数の階層キーリングで 1 つの共有キャッシュを使用できます。共有キャッシュを使用して階層キーリングを作成するときは、オプションのパーティション ID を定義できます。パーティション ID は、キャッシュに書き込む階層キーリングを区別します。2 つの階層キーリングが同じパーティション ID、logical key store name、ブランチキー ID を参照する場合、2 つのキーリングはキャッシュ内で同じキャッシュエントリを共有します。同じ共有キャッシュで異なるパーティション IDs を持つ 2 つの階層キーリングを作成すると、各キーリングは共有キャッシュ内の独自の指定されたパーティ

ションからのみキャッシュエントリにアクセスします。パーティションは共有キャッシュ内の論理分割として機能し、各階層キーリングは、他のパーティションに保存されているデータを妨害することなく、独自の指定されたパーティションで独立して動作できます。

パーティション内のキャッシュエントリを再利用または共有する場合は、独自のパーティション ID を定義する必要があります。パーティション ID を階層キーリングに渡すと、キーリングは、ブランチキーマテリアルを再度取得して再承認するのではなく、共有キャッシュに既に存在するキャッシュエントリを再利用できます。パーティション ID を指定しない場合、階層キーリングをインスタンス化するたびに、一意のパーティション ID がキーリングに自動的に割り当てられます。

次の手順は、デフォルトのキャッシュタイプで共有キャッシュを作成し、階層キーリングに渡す方法を示しています。

1. マテリアルプロバイダーライブラリ CryptographicMaterialsCache (MPL) を使用して (CMC) を作成します。 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. 共有キャッシュのCacheTypeオブジェクトを作成します。

   ステップ 1 でsharedCryptographicMaterialsCache作成した を新しいCacheTypeオブジェクトに渡します。

   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. ステップ 2 の sharedCache オブジェクトを階層キーリングに渡します。

共有キャッシュを使用して階層キーリングを作成する場合、オプションで を定義partitionIDして、複数の階層キーリング間でキャッシュエントリを共有できます。パーティション ID を指定しない場合、階層キーリングはキーリングに一意のパーティション ID を自動的に割り当てます。

> **Note**
>
> 同じパーティション ID、、logical key store nameブランチキー ID を参照する 2 つ以上のキーリングを作成すると、階層キーリングは共有キャッシュ内で同じキャッシュエントリを共有します。複数のキーリングで同じキャッシュエントリを共有しない場合は、階層キーリングごとに一意のパーティション ID を使用する必要があります。

次の例では、 でbranch key ID supplierキャッシュ制限が 600 秒の階層キーリングを作成します。次の階層キーリング設定で定義されている値の詳細については、「」を参照してくださいthe section called “階層キーリングを作成する”。

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?;
```

階層キーリングを作成する

階層キーリングを作成するには、次の値を指定する必要があります。

- キーストア名

  キーストアとして機能するために作成した DynamoDB テーブルの名前、またはキーストア管理者。

-

  キャッシュ制限 Time to Live (TTL)

  ローカルキャッシュ内のブランチキーマテリアルエントリを使用できる時間 (期限切れになるまでの時間) (秒)。キャッシュ制限 TTL は、クライアントがブランチキーの使用を許可 AWS KMS するために を呼び出す頻度を指定します。この値はゼロより大きくなければなりません。キャッシュ制限 TTL の有効期限が切れると、エントリは提供されず、ローカルキャッシュから削除されます。

- ブランチキーの識別子

  キーストア内の 1 つのアクティブなブランチキー`branch-key-id`を識別する を静的に設定するか、ブランチキー ID サプライヤーを指定できます。

  ブランチキー ID サプライヤは、暗号化コンテキストに保存されているフィールドを使用して、レコードの復号に必要なブランチキーを決定します。デフォルトでは、パーティションキーとソートキーのみが暗号化コンテキストに含まれます。ただし、`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`暗号化アクションを使用して、暗号化コンテキストに追加のフィールドを含めることができます。

  各テナントに独自のブランチキーがあるマルチテナントデータベースには、ブランチキー ID サプライヤーを使用することを強くお勧めします。ブランチキー ID サプライヤーを使用してブランチキー IDs のわかりやすい名前を作成し、特定のテナントの正しいブランチキー ID を簡単に認識できます。例えば、フレンドリ名を使用すると、ブランチキーを `b3f61619-4d35-48ad-a275-050f87e15122` の代わりに `tenant1` として参照できます。

  復号オペレーションの場合、単一の階層キーリングを静的に設定して復号を単一のテナンシーに制限することも、ブランチキー ID サプライヤーを使用してレコードの復号を担当するテナンシーを識別することもできます。

- (オプション) キャッシュ

  キャッシュタイプまたはローカルキャッシュに格納できるブランチキーマテリアルエントリの数をカスタマイズする場合は、キーリングを初期化する際にキャッシュタイプとエントリキャパシティを指定します。

  階層キーリングは、デフォルト、MultiThreaded、StormTracking、共有のキャッシュタイプをサポートします。各キャッシュタイプを定義する方法の詳細と例については、「」を参照してくださいthe section called “キャッシュを選択する”。

  キャッシュを指定しない場合、階層キーリングは、自動的に Default キャッシュタイプを使用し、エントリキャパシティを 1,000 に設定します。

- (オプション) パーティション ID

  を指定する場合はthe section called “共有キャッシュ”、オプションでパーティション ID を定義できます。パーティション ID は、キャッシュに書き込む階層キーリングを区別します。パーティション内のキャッシュエントリを再利用または共有する場合は、独自のパーティション ID を定義する必要があります。パーティション ID には任意の文字列を指定できます。パーティション ID を指定しない場合、作成時に一意のパーティション ID がキーリングに自動的に割り当てられます。

  詳細については、「Partitions」を参照してください。

  > **Note**
  >
  > 同じパーティション ID、、logical key store nameブランチキー ID を参照する 2 つ以上のキーリングを作成すると、階層キーリングは共有キャッシュ内で同じキャッシュエントリを共有します。複数のキーリングで同じキャッシュエントリを共有しない場合は、階層キーリングごとに一意のパーティション ID を使用する必要があります。

- (オプション) 許可トークンのリスト

  階層キーリング内の KMS キーへのアクセスを許可によって制御する場合は、キーリングを初期化する際に必要なすべての許可トークンを指定する必要があります。

静的ブランチキー ID を使用して階層キーリングを作成する

次の例は、静的ブランチキー ID、、the section called “デフォルトキャッシュ”キャッシュ制限 TTL が 600 秒の階層キーリングを作成する方法を示しています。

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?;
```

ブランチキー ID サプライヤーを使用して階層キーリングを作成する

次の手順は、ブランチキー ID サプライヤーを使用して階層キーリングを作成する方法を示しています。

1. ブランチキー ID サプライヤーを作成する

   次の例では、ステップ 1 で作成した 2 つのブランチキーのフレンドリ名を作成し、`CreateDynamoDbEncryptionBranchKeyIdSupplier`を呼び出して AWS Database Encryption SDK for DynamoDB クライアントを使用してブランチキー ID サプライヤーを作成します。

   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. 階層キーリングを作成する

次の例では、ステップ 1 で作成したブランチキー ID サプライヤー、キャッシュ制限 TLL が 600 秒、最大キャッシュサイズが 1000 の階層キーリングを初期化します。

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?;
```

検索可能な暗号化のための階層キーリングの使用

検索可能な暗号化を使用すると、データベース全体を復号することなく、暗号化されたレコードを検索できます。これは、ビーコンを使用して暗号化されたフィールドのプレーンテキストの値にインデックスを付けることで実現されます。検索可能な暗号化を実装するには、階層キーリングを使用する必要があります。

キーストア CreateKey オペレーションは、ブランチキーとビーコンキーの両方を生成します。ブランチキーは、レコードの暗号化および復号オペレーションで使用されます。ビーコンキーは、ビーコンを生成するために使用されます。

ブランチキーとビーコンキーは、キーストアサービスの作成時に指定した AWS KMS key ものと同じ によって保護されます。CreateKey オペレーションが AWS KMS を呼び出してブランチキーを生成すると、kms:GenerateDataKeyWithoutPlaintext を 2 回呼び出して、次のリクエストを使用してビーコンキーを生成します。

```
{
   "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"
}
```

両方のキーを生成した後、CreateKey オペレーションは ddb:TransactWriteItems を呼び出して、ブランチキーとビーコンキーを永続化する 2 つの新しい項目をブランチキーストアに書き込みます。

標準ビーコンを設定すると、 AWS Database Encryption SDK はキーストアにビーコンキーをクエリします。その後、HMAC ベースの extract-and-expand 鍵導出関数 (HKDF) を使用して、ビーコンキーと標準ビーコンの名前を組み合わせ、特定のビーコンの HMAC キーを作成します。

ブランチキーとは異なり、キーストアbranch-key-idには ごとに 1 つのビーコンキーバージョンしかありません。ビーコンキーがローテーションされることはありません。

ビーコンキーソースの定義

標準ビーコンおよび複合ビーコンのビーコンバージョンを定義する際には、ビーコンキーを識別し、ビーコンキーマテリアルのキャッシュ制限 Time To Live (TTL) を定義する必要があります。ビーコンキーマテリアルは、ブランチキーとは別のローカルキャッシュに格納されます。次のスニペットは、シングルテナンシーデータベースの keySource を定義する方法を示しています。関連付けられている branch-key-id によってビーコンキーを識別します。

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()?,
))
```

マルチテナンシーデータベースでのビーコンソースの定義

マルチテナンシーデータベースがある場合は、keySource を設定する際に次の値を指定する必要があります。

- keyFieldName

  特定のテナンシーについて生成されたビーコンに使用されるビーコンキーに関連付けられた branch-key-id を格納するフィールドの名前を定義します。keyFieldName には任意の文字列を指定できますが、データベース内の他のすべてのフィールドで一意である必要があります。新しいレコードをデータベースに書き込むと、そのレコードについてのビーコンを生成するために使用されるビーコンキーを識別する branch-key-id がこのフィールドに格納されます。このフィールドをビーコンクエリに含めて、ビーコンの再計算に必要となる適切なビーコンキーマテリアルを特定する必要があります。詳細については、「マルチテナンシーデータベース内のビーコンのクエリ」を参照してください。

- cacheTTL

  ローカルビーコンキャッシュ内のビーコンキーマテリアルエントリを使用できる時間 (期限切れになるまでの時間) (秒)。この値はゼロより大きくなければなりません。キャッシュ制限 TTL の期限が切れると、エントリはローカルキャッシュから削除されます。

- (オプション) キャッシュ

  キャッシュタイプまたはローカルキャッシュに格納できるブランチキーマテリアルエントリの数をカスタマイズする場合は、キーリングを初期化する際にキャッシュタイプとエントリキャパシティを指定します。

  階層キーリングは、デフォルト、MultiThreaded、StormTracking、共有のキャッシュタイプをサポートします。各キャッシュタイプを定義する方法の詳細と例については、「」を参照してくださいthe section called “キャッシュを選択する”。

キャッシュを指定しない場合、階層キーリングは、自動的に Default キャッシュタイプを使用し、エントリキャパシティを 1,000 に設定します。

次の例では、ブランチキー ID サプライヤー、キャッシュ制限 TLL が 600 秒、エントリ容量が 1000 の階層キーリングを作成します。

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 キーリング

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

> **Important**
>
> AWS KMS ECDH キーリングは、 マテリアルプロバイダーライブラリのバージョン 1.5.0 以降でのみ使用できます。

AWS KMS ECDH キーリングは、非対称キーアグリーメントAWS KMS keysを使用して、2 つの当事者間で共有対称ラッピングキーを取得します。まず、キーリングは楕円曲線 Diffie-Hellman (ECDH) キーアグリーメントアルゴリズムを使用して、送信者の KMS キーペアと受信者のパブリックキーのプライベートキーから共有シークレットを取得します。次に、キーリングは共有シークレットを使用して、データ暗号化キーを保護する共有ラッピングキーを取得します。 AWS Database Encryption SDK が (KDF_CTR_HMAC_SHA384) を使用して共有ラッピングキーを取得するキー取得関数は、キー取得に関する NIST レコメンデーションに準拠しています。

キー取得関数は、64 バイトのキーマテリアルを返します。両当事者が正しいキーマテリアルを使用していることを確認するために、 AWS Database Encryption SDK は最初の 32 バイトをコミットメントキーとして使用し、最後の 32 バイトを共有ラッピングキーとして使用します。復号時に、キーリングが暗号化されたレコードのマテリアル説明フィールドに保存されているのと同じコミットメントキーと共有ラッピングキーを再現できない場合、オペレーションは失敗します。たとえば、Alice のプライベートキーと Bob のパブリックキーで設定されたキーリングを使用してレコードを暗号化する場合、Bob のプライベートキーと Alice のパブリックキーで設定されたキーリングは、同じコ

ミットメントキーと共有ラッピングキーを再現し、レコードを復号化できます。Bob のパブリックキーが KMS キーペアからでない場合、Bob は Raw ECDH キーリングを作成してレコードを復号できます。

AWS KMS ECDH キーリングは、AES-GCM を使用して対称キーでレコードを暗号化します。次に、データキーは、AES-GCM を使用して派生した共有ラッピングキーでエンベロープ暗号化されます。各 AWS KMS ECDH キーリングには 1 つの共有ラッピングキーのみを含めることができますが、複数の AWS KMS ECDH キーリングを単独で、または他のキーリングと共にマルチキーリングに含めることができます。

トピック

- AWS KMS ECDH キーリングに必要なアクセス許可
- AWS KMS ECDH キーリングの作成
- AWS KMS ECDH 検出キーリングの作成

AWS KMS ECDH キーリングに必要なアクセス許可

AWS Database Encryption SDK は AWS アカウントを必要とせず、どの AWS サービスにも依存しません。ただし、 AWS KMS ECDH キーリングを使用するには、 AWS アカウントと、キーリング AWS KMS keys の に対する以下の最小限のアクセス許可が必要です。アクセス許可は、使用するキーアグリーメントスキーマによって異なります。

- `KmsPrivateKeyToStaticPublicKey` キーアグリーメントスキーマを使用してレコードを暗号化および復号するには、送信者の非対称 KMS キーペアに kms:GetPublicKey と kms:DeriveSharedSecret が必要です。キーリングをインスタンス化するときに送信者の DER エンコードされたパブリックキーを直接指定する場合、送信者の非対称 KMS キーペアに対する kms:DeriveSharedSecret アクセス許可のみが必要です。
- `KmsPublicKeyDiscovery` キーアグリーメントスキーマを使用してレコードを復号するには、指定された非対称 KMS キーペアに対する kms:DeriveSharedSecret および kms:GetPublicKey アクセス許可が必要です。

AWS KMS ECDH キーリングの作成

データを暗号化および復号する AWS KMS ECDH キーリングを作成するには、`KmsPrivateKeyToStaticPublicKey`キーアグリーメントスキーマを使用

する必要があります。キーアグリーメントスキーマを使用して AWS KMS ECDH `KmsPrivateKeyToStaticPublicKey` キーリングを初期化するには、次の値を指定します。

- 送信者の AWS KMS key ID

  `KeyUsage` 値が の非対称 NIST 推奨楕円曲線 (ECC)KMS キーペアを識別する必要がありますKEY_AGREEMENT。送信者のプライベートキーは、共有シークレットを取得するために使用されます。

- (オプション) 送信者のパブリックキー

  RFC 5280 で定義されているように、 `SubjectPublicKeyInfo` (SPKI) とも呼ばれる DER エンコードされた X.509 パブリックキーである必要があります。 https://tools.ietf.org/html/rfc5280

  AWS KMS GetPublicKey オペレーションは、非対称 KMS キーペアのパブリックキーを必要な DER エンコード形式で返します。

  キーリングが行う AWS KMS 呼び出しの数を減らすには、送信者のパブリックキーを直接指定できます。送信者のパブリックキーに値が指定されていない場合、キーリングは AWS KMS を呼び出して送信者のパブリックキーを取得します。

- 受信者のパブリックキー

  RFC 5280 で定義されているように、(`SubjectPublicKeyInfo`SPKI) とも呼ばれる受信者の DER エンコードされた X.509 パブリックキーを指定する必要があります。 https://tools.ietf.org/html/rfc5280

  AWS KMS GetPublicKey オペレーションは、非対称 KMS キーペアのパブリックキーを必要な DER エンコード形式で返します。

- 曲線仕様

  指定されたキーペアの楕円曲線仕様を識別します。送信者と受信者の両方のキーペアは、同じ曲線仕様である必要があります。

  有効な値: `ECC_NIST_P256`、`ECC_NIS_P384`、`ECC_NIST_P512`

- (オプション) 許可トークンのリスト

  AWS KMS ECDH キーリングの KMS キーへのアクセスをグラントで制御する場合は、キーリングを初期化するときに必要なすべてのグラントトークンを指定する必要があります。

C# / .NET

次の例では、送信者の KMS キー、送信者のパブリックキー、受信者のパブリックキーを使用して、 を使用して AWS KMS ECDH キーリングを作成します。この例では、オプションの senderPublicKeyパラメータを使用して、送信者のパブリックキーを指定します。送信者のパブリックキーを指定しない場合、キーリングは AWS KMS を呼び出して送信者のパブリックキーを取得します。送信者と受信者の両方のキーペアがECC_NIST_P256曲線上にあります。

```
// 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

次の例では、送信者の KMS キー、送信者のパブリックキー、受信者のパブリックキーを使用して、 を使用して AWS KMS ECDH キーリングを作成します。この例では、オプションの senderPublicKeyパラメータを使用して、送信者のパブリックキーを指定します。送信者のパ

ブリックキーを指定しない場合、キーリングは AWS KMS を呼び出して送信者のパブリックキーを取得します。送信者と受信者の両方のキーペアがECC_NIST_P256曲線上にあります。

```
// 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

次の例では、送信者の KMS キー、送信者のパブリックキー、受信者のパブリックキーを使用して、 を使用して AWS KMS ECDH キーリングを作成します。この例では、オプションの sender_public_keyパラメータを使用して、送信者のパブリックキーを指定します。送信者のパブリックキーを指定しない場合、キーリングは AWS KMS を呼び出して送信者のパブリックキーを取得します。

```
// 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 検出キーリングの作成

復号するときは、 AWS Database Encryption SDK が使用できるキーを指定するのがベストプラクティスです。このベストプラクティスに従うには、KmsPrivateKeyToStaticPublicKeyキーアグリーメントスキーマで AWS KMS ECDH キーリングを使用します。ただし、 AWS KMS ECDH 検出キーリング、つまり、指定された KMS キーペアのパブリックキーが、暗号化されたレコードのマテリアル説明フィールドに保存されている受信者のパブリックキーと一致するレコードを復号できる AWS KMS ECDH キーリングを作成することもできます。

> **Important**
>
> KmsPublicKeyDiscovery キーアグリーメントスキーマを使用してレコードを復号する場合、所有者に関係なく、すべてのパブリックキーを受け入れます。

キーアグリーメントスキーマを使用して AWS KMS ECDH KmsPublicKeyDiscovery キーリングを初期化するには、次の値を指定します。

- 受信者の AWS KMS key ID

  KeyUsage 値が の非対称 NIST 推奨楕円曲線 (ECC)KMS キーペアを識別する必要がありますKEY_AGREEMENT。
- 曲線仕様

  受信者の KMS キーペアの楕円曲線仕様を識別します。

  有効な値: ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512
- (オプション) 許可トークンのリスト

  AWS KMS ECDH キーリングの KMS キーへのアクセスをグラントで制御する場合は、キーリングを初期化するときに必要なすべてのグラントトークンを指定する必要があります。

C# / .NET

次の例では、ECC_NIST_P256曲線に KMS キーペアを持つ AWS KMS ECDH 検出キーリングを作成します。指定された KMS キーペアに対する kms:GetPublicKey および kms:DeriveSharedSecret アクセス許可が必要です。このキーリングは、指定された KMS キーペアのパブリックキーが、暗号化されたレコードのマテリアル説明フィールドに保存されている受信者のパブリックキーと一致するレコードを復号できます。

```
// 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

次の例では、ECC_NIST_P256曲線に KMS キーペアを持つ AWS KMS ECDH 検出キーリングを作成します。指定された KMS キーペアに対する kms:GetPublicKey および kms:DeriveSharedSecret アクセス許可が必要です。このキーリングは、指定された KMS キーペアのパブリックキーが、暗号化されたレコードのマテリアル説明フィールドに保存されている受信者のパブリックキーと一致するレコードを復号できます。

```
// 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 キーリング

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK では、データキーを保護するラッピングキーとして指定した AES 対称キーを使用できます。キーマテリアルを生成、格納、保護する必要があります (ハードウェアセキュリティモジュール (HSM) またはキー管理システムで行うのが好ましいです)。ラッピングキーを指定し、ローカルまたはオフラインでデータキーを暗号化する必要がある場合は、Raw AES キーリングを使用します。

Raw AES キーリングは、AES-GCM アルゴリズムと、バイト配列として指定したラッピングキーを使用することによってデータを暗号化します。各 Raw AES キーリングで指定できるラッピングキーは 1 つだけですが、複数の Raw AES キーリングを単独で、または他のキーリングとともにマルチキーリングに含めることができます。

主要な名前空間と名前

キーリング内の AES キーを識別するために、Raw AES キーリングは、指定したキーの名前空間とキー名を使用します。これらの値はシークレットではありません。Database AWS Encryption SDK

がレコードに追加するマテリアルの説明にプレーンテキストで表示されます。HSM またはキー管理システムのキーの名前空間と、そのシステムで AES キーを識別するキー名を使用することをお勧めします。

> **Note**
>
> キーの名前空間とキー名は、`JceMasterKey` の [プロバイダー ID] (または [プロバイダー]) フィールドと [キー ID] フィールドに相当します。

特定のフィールドを暗号化および復号するために異なるキーリングを構築する場合、名前空間と名前の値が重要です。復号キーリング内のキーの名前空間とキー名が、暗号化キーリング内のキーの名前空間とキー名の大文字と小文字の区別に正確に一致しない場合、キーマテリアルのバイトが同一であっても、復号キーリングは使用されません。

例えば、キーの名前空間 `HSM_01` とキー名 `AES_256_012` を使用して Raw AES キーリングを定義するとします。その後、そのキーリングを使用して一部のデータを暗号化します。そのデータを復号するには、同じキー名前空間、キー名、およびキーマテリアルを使用して Raw AES キーリングを作成します。

次の例は、Raw AES キーリングの作成方法を示しています。`AESWrappingKey` 変数は、指定したキーマテリアルを表します。

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 キーリング

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

Raw RSA キーリングは、指定した RSA パブリックキーとプライベートキーを使用して、ローカルメモリでデータキーの非対称の暗号化と復号を実行します。プライベートキーを生成、格納、保護す

る必要があります (ハードウェアセキュリティモジュール (HSM) またはキー管理システムで行うのが好ましいです)。暗号化関数を使用して、RSA パブリックキーのデータキーを暗号化します。復号関数でプライベートキーを使用して、データキーを復号します。複数の RSA パディングモードから選択できます。

暗号化と復号を行う Raw RSA キーリングには、非対称のパブリックキーとプライベートキーのペアを含める必要があります。ただし、データの暗号化は、パブリックキーのみを持つ Raw RSA キーリングを使用して行うことができます。また、データの復号は、プライベートキーのみを持つ Raw RSA キーリングを使用して行うことができます。Raw RSA キーリングは、マルチキーリングに含めることができます。Raw RSA キーリングをパブリックキーおよびプライベートキーを使用して設定する場合は、それらが同じキーペアの一部であることを確認してください。

Raw RSA キーリングは、RSA 非対称暗号化キーで使用される AWS Encryption SDK for Java 場合、の JceMasterKey と同等であり、相互運用されます。

> **Note**
>
> Raw RSA キーリングは、非対称 KMS キーをサポートしません。非対称 RSA KMS キーを使用するには、AWS KMS キーリングを構築します。

名前空間と名前

キーリング内の RSA キーマテリアルを識別するために、Raw RSA キーリングは、指定したキーの名前空間とキー名を使用します。これらの値はシークレットではありません。Database AWS Encryption SDK がレコードに追加するマテリアルの説明にプレーンテキストで表示されます。HSM またはキー管理システムで RSA キーペア (またはそのプライベートキー) を識別するキーの名前空間とキー名を使用することをお勧めします。

> **Note**
>
> キーの名前空間とキー名は、`JceMasterKey` の [プロバイダー ID] (または [プロバイダー]) フィールドと [キー ID] フィールドに相当します。

特定のレコードを暗号化および復号するために異なるキーリングを構築する場合、名前空間と名前の値が重要です。復号キーリング内のキーの名前空間とキー名が、暗号化キーリング内のキーの名前空間とキー名の大文字と小文字の区別に正確に一致しない場合、そのキーが同じキーペアからのものであっても、復号キーリングは使用されません。

暗号化および復号キーリング内のキーマテリアルのキーの名前空間とキー名は、キーリングのキーペアに RSA パブリックキー、RSA プライベートキー、または両方のキーが含まれているかどうかにかかわらず、同じである必要があります。例えば、キーの名前空間 `HSM_01` とキー名 `RSA_2048_06` を持つ RSA パブリックキーの Raw RSA キーリングを使用してデータを暗号化するとします。そのデータを復号するには、プライベートキー (またはキーペア)、および同じキーの名前空間と名前を使用して Raw RSA キーリングを構築します。

パディングモード

暗号化と復号に使用される Raw RSA キーリングのためにパディングモードを指定するか、またはそれを指定する言語実装の機能を使用する必要があります。

は、各言語の制約に従い、次のパディングモード AWS Encryption SDK をサポートしています。OAEP パディングモード、特に SHA-256 を使用する OAEP および SHA-256 パディングを使用する MGF1 をお勧めします。PKCS1 パディングモードは、下位互換性のためのみサポートされています。

- SHA-1 を使用する OAEP および SHA-1 パディングを使用する MGF1
- SHA-256 を使用する OAEP および SHA-256 パディングを使用する MGF1
- SHA-384 を使用する OAEP および SHA-384 パディングを使用する MGF1
- SHA-512 を使用する OAEP および SHA-512 パディングを使用する MGF1
- PKCS1 v1.5 パディング

次の Java の例は、RSA キーペアのパブリックキーとプライベートキーを使用し、SHA-256 を使用する OAEP および SHA-256 パディングモードを使用する MGF1 を採用する Raw RSA キーリングを作成する方法を示しています。`RSAPublicKey` および `RSAPrivateKey` 変数は、指定するキーマテリアルを表します。

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?;
```

Raw ECDH キーリング

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

> **Important**
>
> Raw ECDH キーリングは、マテリアルプロバイダーライブラリのバージョン 1.5.0 でのみ使用できます。

Raw ECDH キーリングは、指定した楕円曲線のパブリック/プライベートキーペアを使用して、2 つの当事者間で共有ラッピングキーを取得します。まず、キーリングは、送信者のプライベートキー、受信者のパブリックキー、楕円曲線 Diffie-Hellman (ECDH) キーアグリーメントアルゴリズムを使用して共有シークレットを取得します。次に、キーリングは共有シークレットを使用して、データ暗号化キーを保護する共有ラッピングキーを取得します。 AWS Database Encryption SDK が (KDF_CTR_HMAC_SHA384) を使用して共有ラッピングキーを取得するキー取得関数は、キー取得に関する NIST レコメンデーションに準拠しています。

キー取得関数は、64 バイトのキーマテリアルを返します。両当事者が正しいキーマテリアルを使用するように、 AWS Database Encryption SDK は最初の 32 バイトをコミットメントキーとして使用し、最後の 32 バイトを共有ラッピングキーとして使用します。復号時に、キーリングが暗号化されたレコードのマテリアル説明フィールドに保存されているのと同じコミットメントキーと共有ラッピングキーを再現できない場合、オペレーションは失敗します。たとえば、Alice のプライベートキーと Bob のパブリックキーで設定されたキーリングを使用してレコードを暗号化する場合、Bob のプライベートキーと Alice のパブリックキーで設定されたキーリングは、同じコミットメントキーと共有ラッピングキーを再現し、レコードを復号化できます。Bob のパブリックキーが AWS KMS key ペアからのものである場合、Bob は AWS KMS ECDH キーリングを作成してレコードを復号できます。

Raw ECDH キーリングは、AES-GCM を使用して対称キーでレコードを暗号化します。次に、データキーは、AES-GCM を使用して派生した共有ラッピングキーでエンベロープ暗号化されます。各 Raw ECDH キーリングには 1 つの共有ラッピングキーのみを含めることができますが、複数の Raw ECDH キーリングを単独で、または他のキーリングと共にマルチキーリングに含めることができます。

プライベートキーの生成、保存、保護は、ハードウェアセキュリティモジュール (HSM) またはキー管理システムで行うことをお勧めします。送信者と受信者のキーペアは、ほぼ同じ楕円曲線上にあります。 AWS Database Encryption SDK は、次の楕円曲線仕様をサポートしています。

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

Raw ECDH キーリングの作成

Raw ECDH キーリングは、`RawPrivateKeyToStaticPublicKey`、、`EphemeralPrivateKeyToStaticPublicKey`の 3 つのキーアグリーメントスキーマをサポートしていますPublicKeyDiscovery。選択したキーアグリーメントスキーマによって、実行できる暗号化オペレーションとキーマテリアルの組み立て方法が決まります。

トピック

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

RawPrivateKeyToStaticPublicKey

`RawPrivateKeyToStaticPublicKey` キーアグリーメントスキーマを使用して、キーリングで送信者のプライベートキーと受信者のパブリックキーを静的に設定します。このキーアグリーメントスキーマは、レコードを暗号化および復号化できます。

`RawPrivateKeyToStaticPublicKey` キーアグリーメントスキーマを使用して Raw ECDH キーリングを初期化するには、次の値を指定します。

- 送信者のプライベートキー

  RFC 5958 で定義されているように、送信者の PEM エンコードされたプライベートキー (PKCS #8 PrivateKeyInfo 構造) を指定する必要があります。
- 受信者のパブリックキー

RFC 5280 で定義されているように、(SubjectPublicKeyInfoSPKI) とも呼ばれる受信者の DER エンコードされた X.509 パブリックキーを指定する必要があります。 https://tools.ietf.org/html/rfc5280

非対称キーアグリーメント KMS キーペアのパブリックキー、または の外部で生成されたキーペアのパブリックキーを指定できます AWS。

- 曲線仕様

  指定されたキーペアの楕円曲線仕様を識別します。送信者と受信者の両方のキーペアには、同じ曲線仕様が必要です。

  有効な値: 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

次の Java の例では、RawPrivateKeyToStaticPublicKeyキーアグリーメントスキーマを使用して、送信者のプライベートキーと受信者のパブリックキーを静的に設定します。両方のキーペアがECC_NIST_P256曲線上にあります。

```
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

次の Python の例では、raw_ecdh_static_configurationキーアグリーメントスキーマを使用して、送信者のプライベートキーと受信者のパブリックキーを静的に設定します。両方のキーペアが同じ曲線上にある必要があります。

```
// 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

キーアグリーメントスキーマで設定されたEphemeralPrivateKeyToStaticPublicKeyキーリングは、新しいキーペアをローカルに作成し、暗号化呼び出しごとに一意の共有ラッピングキーを取得します。

このキーアグリーメントスキーマはレコードのみを暗号化できます。EphemeralPrivateKeyToStaticPublicKey キーアグリーメントスキーマで暗号化されたレコードを復号するには、同じ受信者のパブリックキーで設定された検出キーアグリーメントスキーマを使用する必要があります。復号するには、PublicKeyDiscoveryキーアグリーメントアルゴリズムで Raw ECDH キーリングを使用するか、受信者のパブリックキーが非対称キーアグリーメン

ト KMS キーペアからのものである場合は、KmsPublicKeyDiscovery キーアグリーメントスキーマで AWS KMS ECDH キーリングを使用できます。

EphemeralPrivateKeyToStaticPublicKey キーアグリーメントスキーマを使用して Raw ECDH キーリングを初期化するには、次の値を指定します。

- 受信者のパブリックキー

  RFC 5280 で定義されているように、(SubjectPublicKeyInfoSPKI) とも呼ばれる受信者の DER エンコードされた X.509 パブリックキーを指定する必要があります。 https://tools.ietf.org/html/rfc5280

  非対称キーアグリーメント KMS キーペアのパブリックキー、または の外部で生成されたキーペアのパブリックキーを指定できます AWS。
- 曲線仕様

  指定されたパブリックキーの楕円曲線仕様を識別します。

  暗号化時に、キーリングは指定された曲線に新しいキーペアを作成し、新しいプライベートキーと指定されたパブリックキーを使用して共有ラッピングキーを取得します。

  有効な値: ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

C# / .NET

次の例では、キーアグリーメントスキーマを使用して Raw ECDH EphemeralPrivateKeyToStaticPublicKeyキーリングを作成します。暗号化時に、キーリングは指定されたECC_NIST_P256曲線に新しいキーペアをローカルに作成します。

```
// 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

次の例では、キーアグリーメントスキーマを使用して Raw ECDH EphemeralPrivateKeyToStaticPublicKeyキーリングを作成します。暗号化時に、キーリングは指定されたECC_NIST_P256曲線に新しいキーペアをローカルに作成します。

```
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

次の例では、キーアグリーメントスキーマを使用して Raw ECDH ephemeral_raw_ecdh_static_configurationキーリングを作成します。暗号化時に、キーリングは指定された曲線に新しいキーペアをローカルに作成します。

```
// 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

復号するときは、 AWS Database Encryption SDK が使用できるラッピングキーを指定するのがベストプラクティスです。このベストプラクティスに従うには、送信者のプライベートキーと受信者のパブリックキーの両方を指定する ECDH キーリングを使用します。ただし、Raw ECDH 検出キーリング、つまり、指定されたキーのパブリックキーが暗号化されたレコードのマテリアル説明フィールドに保存されている受信者のパブリックキーと一致するレコードを復号できる Raw ECDH キーリングを作成することもできます。このキーアグリーメントスキーマはレコードのみを復号できます。

> **Important**
>
> PublicKeyDiscovery キーアグリーメントスキーマを使用してレコードを復号する場合、所有者に関係なく、すべてのパブリックキーを受け入れます。

PublicKeyDiscovery キーアグリーメントスキーマを使用して Raw ECDH キーリングを初期化するには、次の値を指定します。

- 受信者の静的プライベートキー

  RFC 5958 で定義されているように、受信者の PEM エンコードされたプライベートキー (PKCS #8 PrivateKeyInfo 構造) を指定する必要があります。

- 曲線仕様

  指定されたプライベートキーの楕円曲線仕様を識別します。送信者と受信者の両方のキーペアには、同じ曲線仕様が必要です。

  有効な値: ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

C# / .NET

次の例では、キーアグリーメントスキーマを使用して Raw ECDH PublicKeyDiscoveryキーリングを作成します。このキーリングは、指定されたプライベートキーのパブリックキーが、暗号化されたレコードのマテリアル説明フィールドに保存されている受信者のパブリックキーと一致するレコードを復号できます。

```
// 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

次の例では、キーアグリーメントスキーマを使用して Raw ECDH PublicKeyDiscoveryキーリングを作成します。このキーリングは、指定されたプライベートキーのパブリックキーが、暗号化されたレコードのマテリアル説明フィールドに保存されている受信者のパブリックキーと一致するレコードを復号できます。

```
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

次の の例では、キーアグリーメントスキーマを使用して Raw ECDH discovery_raw_ecdh_static_configurationキーリングを作成します。このキーリングは、指定されたプライベートキーのパブリックキーがメッセージ暗号文に保存されている受信者のパブリックキーと一致するメッセージを復号できます。

```
// 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?;
```

マルチキーリング

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

キーリングは組み合わせてマルチキーリングにすることができます。マルチキーリングは、種類に関係なく、1 つ以上の個別のキーリングで構成されるキーリングです。一連のキーリングを複数使用した場合のように動作します。マルチキーリングを使用してデータを暗号化する場合は、そのキーリングに含まれる任意のラッピングキーを使用してそのデータを復号できます。

マルチキーリングを作成してデータを暗号化する場合は、いずれかのキーリングをジェネレーターキーリングに指定します。他のすべてのキーリングは、子キーリングと呼ばれます。ジェネレーターキーリングは、プレーンテキストのデータキーを生成して暗号化します。その後、すべての子キーリングのすべてのラッピングキーによって、そのプレーンテキストデータキーが暗号化されます。マルチキーリングは、プレーンテキストのキーと、マルチキーリングのラッピングキーごとに 1 つの暗号化されたデータキーを返します。ジェネレーターキーリングが KMS キーリングの場合、 AWS KMS キーリングのジェネレーターキーはプレーンテキストキーを生成して暗号化します。次に、 AWS KMS キーリング AWS KMS keys のすべての追加キーと、マルチキーリングのすべての子キーリングのすべてのラッピングキーは、同じプレーンテキストキーを暗号化します。

復号時に、 AWS Database Encryption SDK はキーリングを使用して、暗号化されたデータキーの 1 つを復号しようとします。キーリングは、マルチキーリングで指定された順番で呼び出されます。暗号化されたデータキーがキーリングの任意のキーによって復号されると、処理は停止されます。

マルチキーリングを作成するにはまず、子キーリングをインスタンス化します。この例では、 AWS KMS キーリングと Raw AES キーリングを使用しますが、サポートされている任意のキーリングをマルチキーリングに結合できます。

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?;
```

次に、マルチキーリングを作成し、ジェネレーターキーリングがある場合はそれを指定します。この例では、キーリングが AWS KMS ジェネレーターキーリング、AES キーリングが子キーリングであるマルチキーリングを作成します。

Java

Java コンCreateMultiKeyringInputストラクタを使用すると、ジェネレーターキーリングと子キーリングを定義できます。結果 createMultiKeyringInput のオブジェクトは不変です。

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

C# / .NET

.NET CreateMultiKeyringInput コンストラクターでは、ジェネレータキーリングと子キーリングを定義できます。結果 CreateMultiKeyringInput のオブジェクトは不変です。

```
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?;
```

これで、データの暗号化と復号にマルチキーリングを使用できます。

検索可能な暗号化

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

検索可能な暗号化を使用すると、データベース全体を復号することなく、暗号化されたレコードを検索できます。これはビーコンを使用して実現されます。ビーコンは、フィールドに書き込まれるプレーンテキストの値と、実際にデータベースに格納される暗号化された値との間のマップを作成します。 AWS Database Encryption SDK は、レコードに追加する新しいフィールドにビーコンを保存します。使用するビーコンのタイプに応じて、暗号化されたデータに対して、完全一致検索や、より力スタマイズされた複雑なクエリを実行できます。

> **Note**
>
> AWS Database Encryption SDK の検索可能な暗号化は、検索可能な対称暗号化など、学術研究で定義された検索可能な対称暗号化とは異なります。

ビーコンは、フィールドのプレーンテキストの値と暗号化された値の間のマップを作成する、切り詰められた Hash-Based Message Authentication Code (HMAC) タグです。検索可能な暗号化用に設定された暗号化されたフィールドに新しい値を書き込むと、 AWS Database Encryption SDK はプレーンテキスト値で HMAC を計算します。この HMAC 出力は、そのフィールドのプレーンテキストの値と 1 対 1 (1:1) で一致します。HMAC 出力は切り詰められ、複数の個別のプレーンテキストの値が、切り詰められた同じ HMAC タグにマッピングされます。これらの誤検知により、不正ユーザーは、プレーンテキストの値に関する特徴的な情報を識別しにくくなります。ビーコンをクエリすると、AWS Database Encryption SDK はこれらの誤検知を自動的に除外し、クエリのプレーンテキスト結果を返します。

各ビーコンについて生成される誤検知の平均数は、切り詰めた後に残っているビーコンの長さによって決まります。実装に適切なビーコンの長さを決定する方法については、「ビーコンの長さの決定」を参照してください。

> **Note**
>
> 検索可能な暗号化は、データが入力されていない新しいデータベースで実装されるように設計されています。既存のデータベースで設定されたビーコンは、データベースにアップロードされた新しいレコードのみをマッピングします。ビーコンは既存のデータをマッピングできなくなります。

トピック

ビーコンが適しているデータセット

ビーコンを使用して、暗号化されたデータをクエリすると、クライアント側の暗号化されたデータベースに関連するパフォーマンスコストが削減されます。ビーコンを使用する場合、クエリの効率性と、データの分布に関して明らかになる情報の量との間には、固有のトレードオフが存在します。ビーコンはフィールドの暗号化状態を変更しません。 AWS Database Encryption SDK を使用してフィールドを暗号化して署名すると、フィールドのプレーンテキスト値がデータベースに公開されることはありません。データベースには、フィールドのランダム化および暗号化された値が格納されます。

ビーコンは、計算元の暗号化されたフィールドと一緒に格納されます。これは、不正ユーザーが暗号化されたフィールドのプレーンテキストの値を表示できない場合でも、ビーコンに対して統計分析を実行してデータセットの分布の詳細を知ることができ、極端な場合には、ビーコンがマッピングするプレーンテキストの値を識別できる場合があることを意味します。ビーコンを設定する方法によって、これらのリスクを軽減できます。特に、適切なビーコンの長さを選択することは、データセットの機密性を維持するのに役立ちます。

セキュリティとパフォーマンス

- ビーコンが短いほど、セキュリティはより強くなります。
- ビーコンが長いほど、パフォーマンスはより高くなります。

検索可能な暗号化では、すべてのデータセットについて、必要なレベルのパフォーマンスとセキュリティの両方を提供できない場合があります。ビーコンを設定する前に、脅威モデル、セキュリティ要件、パフォーマンスのニーズを確認してください。

検索可能な暗号化がデータセットに適しているかどうかを判断する際には、データセットの一意性に関する次の要件を考慮してください。

ディストリビューション

ビーコンによって維持されるセキュリティの強度は、データセットの分布によって異なります。検索可能な暗号化用に暗号化されたフィールドを設定すると、 AWS Database Encryption SDK はそのフィールドに書き込まれたプレーンテキスト値で HMAC を計算します。特定のフィールドについて計算されるすべてのビーコンは、テナンシーごとに個別のキーを使用するマルチテナンシーデータベースを除き、同じキーを使用して計算されます。これは、同じプレーンテキストの値がフィールドに複数回書き込まれる場合、そのプレーンテキストの値のすべてについて同じ HMAC タグが作成されることを意味します。

非常に一般的な値を含むフィールドからビーコンを構築しないようにしてください。例えば、イリノイ州のすべての居住者の住所を格納するデータベースを考えてみましょう。暗号化された `City` フィールドからビーコンを構築する場合、シカゴに居住しているイリノイ州の母集団の割合が大きいため、「シカゴ」について計算されたビーコンは過剰に出現します。不正ユーザーが暗号化された値とビーコンの値を読み取ることしかできない場合でも、ビーコンがこの分布を保持していれば、どのレコードにシカゴの居住者のデータが含まれているかを特定できる可能性があります。分布に関して明らかになる特徴的な情報の量を最小限にするには、ビーコンを十分に切り詰める必要があります。この不均一な分布をわからなくするために必要な長さにビーコンを設定すると、パフォーマンスに大きな悪影響が及び、アプリケーションのニーズを満たせない可能性があります。

データセットの分布を注意深く分析して、ビーコンをどの程度切り詰める必要があるかを判断する必要があります。切り詰めた後に残るビーコンの長さは、分布に関して特定できる統計情報の量に直接相関します。データセットに関して明らかになる特徴的な情報の量を十分かつ最小限に抑えるために、ビーコンをより短くすることを選択する必要がある場合があります。

極端な場合には、不均一に分布したデータセットについて、パフォーマンスとセキュリティのバランスを効果的に実現できるビーコンの長さを計算することができません。例えば、希少疾患の医学的検査の結果を格納するフィールドからビーコンを構築しないでください。`NEGATIVE` の結果はデータセット内で大幅に増えることが想定されるため、`POSITIVE` の結果は、それがどれだけ稀であるかによって簡単に識別できます。フィールドで可能な値が 2 つしかない場合、分布をわからなくするのは非常に困難です。分布をわからなくするのに十分な程度にまでビーコンを短

くすると、すべてのプレーンテキストの値が同じ HMAC タグにマッピングされます。ビーコンをより長くすると、どのビーコンがプレーンテキストの `POSITIVE` の値にマッピングされているのかが明らかになります。

相関関係

相関する値を持つフィールドから個別のビーコンを構築しないことを強くお勧めします。相関するフィールドから構築されたビーコンでは、各データセットの分布に関して、不正ユーザーに対して明らかになる情報の量を十分かつ最小限に抑えるために、ビーコンをより短くする必要があります。ビーコンをどの程度切り詰める必要があるかを判断するには、エントロピーや相関する値の結合分布などのデータセットを注意深く分析する必要があります。結果として得られるビーコンの長さがパフォーマンスのニーズを満たさない場合、ビーコンはデータセットに適していない可能性があります。

例えば、郵便番号は 1 つの都市にのみ関連付けられている可能性が高いため、`City` フィールドと `ZIPCode` フィールドから 2 つの別個のビーコンを構築すべきではありません。通常、ビーコンによって生成される誤検知により、不正ユーザーは、データセットに関する特徴的な情報を識別しにくくなります。ただし、`City` および `ZIPCode` フィールド間の相関関係を知ることで、不正ユーザーは、どの結果が誤検知であるかを簡単に特定し、異なる郵便番号を区別できます。

また、同じプレーンテキストの値を含むフィールドからビーコンを構築することも避けてください。例えば、`mobilePhone` および `preferredPhone` フィールドは同じ値を保持する可能性が高いため、これらのフィールドからビーコンを構築すべきではありません。両方のフィールドから異なるビーコンを構築すると、 AWS Database Encryption SDK は異なるキーで各フィールドのビーコンを作成します。これにより、同じプレーンテキストの値について 2 つの異なる HMAC タグが作成されます。2 つの異なるビーコンに同じ誤検知が発生する可能性は低く、不正ユーザーは異なる電話番号を区別できる可能性があります。

相関するフィールドがデータセットに含まれている場合や、分布が不均一である場合でも、ビーコンをより短くすることで、データセットの機密性を維持するビーコンを構築できる場合があります。ただし、ビーコンの長さは、データセット内のすべての一意の値が多数の誤検知を生成し、データセットに関して明らかになる特徴的な情報の量を効果的かつ最小限に抑えることを保証するものではありません。ビーコンの長さによって推定されるのは、生成される誤検知の平均数のみです。データセットが不均一に分布しているほど、生成される誤検知の平均数を決定する際のビーコンの長さの有効性は低くなります。

ビーコンを構築するフィールドの分布を慎重に検討し、セキュリティ要件を満たすためにビーコンの長さをどの程度切り詰める必要があるのかを検討してください。この章の次のトピックは、ビーコンが統一的に分布しており、相関データが含まれていないことを前提としています。

検索可能な暗号化のシナリオ

次の例は、検索可能な暗号化のシンプルなソリューションを示しています。アプリケーションでは、この例で使用されているフィールド例は、ビーコンの分布および相関の一意性に関する推奨事項を満たしていない可能性があります。この章の検索可能な暗号化の概念を読む際に、この例を参考として使用できます。

会社の従業員データを追跡する Employees という名前のデータベースについて考えてみましょう。データベース内の各レコードには、EmployeeID、LastName、FirstName、および Address と呼ばれるフィールドが含まれています。Employees データベース内の各フィールドは、プライマリキー EmployeeID によって識別されます。

データベース内のプレーンテキストレコードの例を次に示します。

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

暗号化アクションで LastName フィールドと FirstName フィールドを ENCRYPT_AND_SIGN とマークした場合、これらのフィールドの値は、データベースにアップロードされる前にローカルで暗号化されます。アップロードされる暗号化データは完全にランダム化されており、データベースはこのデータが保護されているとは認識しません。典型的なデータエントリを検出するだけです。つまり、実際にデータベースに格納されるレコードは次のようになります。

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

```
    }
}
```

LastName フィールド内の完全一致を検索するために、データベースをクエリする必要がある場合は、LastName フィールドに書き込まれるプレーンテキストの値を、データベースに格納される暗号化された値にマッピングするように、LastName という名前の標準ビーコンを設定します。

このビーコンは、LastName フィールド内のプレーンテキストの値から HMAC を計算します。各 HMAC 出力は切り詰められるため、プレーンテキストの値と完全に一致しなくなります。例えば、Jones の完全なハッシュと切り詰められたハッシュは次のようになります。

完全なハッシュ

2aa4e9b404c68182562b6ec761fcca5306de527826a69468885e59dc36d0c3f824bdd44cab45526f

切り詰められたハッシュ

b35099d408c833

標準ビーコンを設定した後、LastName フィールド上で一致検索を実行できます。例えば、Jones を検索する場合は、LastName ビーコンを使用して次のクエリを実行します。

```
LastName = Jones
```

AWS Database Encryption SDK は誤検出を自動的に除外し、クエリのプレーンテキストの結果を返します。

ビーコン

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

ビーコンは、フィールドに書き込まれるプレーンテキストの値と、実際にデータベースに格納される暗号化された値の間のマップを作成する、切り詰められた Hash-Based Message Authentication Code (HMAC) タグです。ビーコンはフィールドの暗号化状態を変更しません。ビーコンは、フィールドのプレーンテキストの値について HMAC を計算し、それを暗号化された値と一緒に格納します。この HMAC 出力は、そのフィールドのプレーンテキストの値と 1 対 1 (1:1) で一致しま

す。HMAC 出力は切り詰められ、複数の個別のプレーンテキストの値が、切り詰められた同じ HMAC タグにマッピングされます。これらの誤検知により、不正ユーザーは、プレーンテキストの値に関する特徴的な情報を識別しにくくなります。

ビーコンは、暗号化アクションSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTで ENCRYPT_AND_SIGN、SIGN_ONLY、または とマークされたフィールドからのみ構築できます。ビーコン自体は署名も暗号化もされません。DO_NOTHING とマークされているフィールドを使用してビーコンを構築することはできません。

設定するビーコンのタイプによって、実行できるクエリのタイプが決まります。検索可能な暗号化をサポートするビーコンには 2 つのタイプがあります。標準ビーコンは、一致検索を実行します。複合ビーコンは、リテラルプレーンテキスト文字列と標準ビーコンを組み合わせて、複雑なデータベースオペレーションを実行します。ビーコンを設定した後、暗号化されたフィールドを検索する前に、各ビーコンについてセカンダリインデックスを設定する必要があります。詳細については、「ビーコンを使用したセカンダリインデックスの設定」を参照してください。

トピック

- 標準ビーコン
- 複合ビーコン

標準ビーコン

標準ビーコンは、データベースで検索可能な暗号化を実装する最も簡単な方法です。単一の暗号化されたフィールドまたは仮想フィールドについてのみ一致検索を実行できます。標準ビーコンの設定方法については、「標準ビーコンの設定」を参照してください。

標準ビーコンが構築されるフィールドは、ビーコンソースと呼ばれます。これは、ビーコンがマッピングする必要があるデータの場所を識別します。ビーコンソースは、暗号化されたフィールドまたは仮想フィールドのいずれかです。各標準ビーコンのビーコンソースは一意である必要があります。同じビーコンソースで 2 つのビーコンを設定することはできません。

標準ビーコンを使用して、暗号化されたフィールドまたは仮想フィールドの等価検索を実行できます。または、複合ビーコンを構築して、より複雑なデータベースオペレーションを実行することもできます。標準ビーコンの整理と管理に役立つように、 AWS Database Encryption SDK には、標準ビーコンの用途を定義する以下のオプションビーコンスタイルが用意されています。詳細については、「ビーコンスタイルの定義」を参照してください。

単一の暗号化されたフィールドに対して等価検索を実行する標準ビーコンを作成することも、仮想フィールドを作成して複数の ENCRYPT_AND_SIGN、、SIGN_ONLYおよび SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドの連結に対して等価検索を実行する標準ビーコンを作成することもできます。

仮想フィールド

仮想フィールドは、1 つ以上のソースフィールドから構築された概念的なフィールドです。仮想フィールドを作成しても、レコードに新しいフィールドは書き込まれません。仮想フィールドは、データベースに明示的に格納されません。これは、フィールドの特定のセグメントを識別する方法、またはレコード内の複数のフィールドを連結して特定のクエリを実行する方法についてビーコンに指示を与えるために、標準ビーコン設定で使用されます。仮想フィールドには少なくとも 1 つの暗号化されたフィールドが必要です。

> **Note**
>
> 次の例は、仮想フィールドを使用して実行できる変換とクエリのタイプを示しています。アプリケーションでは、この例で使用されているフィールド例は、ビーコンの分布および相関の一意性に関する推奨事項を満たしていない可能性があります。

例えば、FirstName および LastName フィールドの連結に対して一致検索を実行する場合は、次のいずれかの仮想フィールドを作成することが考えられます。

- FirstName フィールドの最初の文字と、それに続く LastName フィールドから構築される仮想 NameTag フィールド (すべて小文字)。この仮想フィールドを使用すると、NameTag=mjones をクエリできます。
- LastName フィールドと、それに続く FirstName フィールドから構築される仮想 LastFirst フィールド。この仮想フィールドを使用すると、LastFirst=JonesMary をクエリできます。

または、暗号化されたフィールドの特定のセグメントに対して一致検索を実行する場合は、クエリを実行するセグメントを識別する仮想フィールドを作成します。

例えば、IP アドレスの最初の 3 つのセグメントを使用して暗号化された IPAddress フィールドをクエリする場合は、次の仮想フィールドを作成します。

- `Segments('.', 0, 3)` から構築された仮想 IPSegment フィールド。この仮想フィールドを使用すると、`IPSegment=192.0.2` をクエリできます。クエリは、「192.0.2」で始まる IPAddress の値を持つすべてのレコードを返します。

  仮想フィールドは一意である必要があります。2 つの仮想フィールドをまったく同じソースフィールドから構築することはできません。

  仮想フィールドとそれらを使用するビーコンの設定については、「仮想フィールドの作成」を参照してください。

複合ビーコン

複合ビーコンは、クエリのパフォーマンスを改善するインデックスを作成し、より複雑なデータベースオペレーションを実行できるようにします。複合ビーコンを使用して、リテラルプレーンテキスト文字列と標準ビーコンを組み合わせて、単一のインデックスから 2 つの異なるレコードタイプをクエリしたり、ソートキーを使用してフィールドの組み合わせをクエリしたりするなど、暗号化されたレコードに対して複雑なクエリを実行できます。複合ビーコンソリューションの例については、「ビーコンタイプを選択する」を参照してください。

複合ビーコンは、標準ビーコン、または標準ビーコンと署名付きフィールドの組み合わせから構築できます。これらは部分のリストから構築されます。すべての複合ビーコンには、ビーコンに含まれる ENCRYPT_AND_SIGN フィールドを識別する暗号化された部分のリストが含まれている必要があります。すべての ENCRYPT_AND_SIGN フィールドは、標準ビーコンによって識別される必要があります。より複雑な複合ビーコンには、ビーコンに含まれるプレーンテキストSIGN_ONLYまたはSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドを識別する署名付きパートのリスト、および複合ビーコンがフィールドをアセンブルできるすべての可能な方法を識別するコンストラクタパートのリストが含まれる場合があります。

> **Note**
>
> AWS Database Encryption SDK は、プレーンテキストSIGN_ONLYとSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドから完全に設定できる署名付きビーコンもサポートしています。署名付きビーコンは、署名されたが暗号化されていないフィールドに対してインデックスを作成し、複雑なクエリを実行する複合ビーコンの一種です。詳細については、「署名付きビーコンの作成」を参照してください。

複合ビーコンの設定については、「複合ビーコンの設定」を参照してください。

複合ビーコンを設定する方法によって、実行できるクエリのタイプが決まります。例えば、一部の暗号化および署名付きの部分をオプションにして、クエリの柔軟性を高めることができます。複合ビーコンが実行できるクエリのタイプの詳細については、「ビーコンのクエリ」を参照してください。

ビーコンの計画

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

ビーコンは、データが入力されていない新しいデータベースに実装されるように設計されています。既存のデータベースで設定されたビーコンは、データベースに書き込まれる新しいレコードのみをマッピングします。ビーコンはフィールドのプレーンテキストの値から計算されます。フィールドが暗号化されると、ビーコンは既存のデータをマッピングできなくなります。ビーコンを持つ新しいレコードを書き込んだ後に、そのビーコンの設定を更新することはできません。ただし、レコードに追加する新しいフィールドに新しいビーコンを追加できます。

検索可能な暗号化を実装するには、AWS KMS 階層キーリングを使用して、レコードを保護するために使用されるデータキーを生成、暗号化、および復号する必要があります。詳細については、「検索可能な暗号化のための階層キーリングの使用」を参照してください。

検索可能な暗号化のためにビーコンを設定する前に、暗号化要件、データベースのアクセスパターン、および脅威モデルを確認して、データベースに最適なソリューションを決定する必要があります。

設定するビーコンのタイプによって、実行できるクエリのタイプが決まります。標準ビーコン設定で指定するビーコンの長さによって、特定のビーコンについて生成される誤検知の想定数が決まります。ビーコンを設定する前に、実行する必要があるクエリのタイプを特定して計画することを強くお勧めします。ビーコンを使用した後に設定を更新することはできません。

ビーコンを設定する前に、次のタスクを確認および完了することを強くお勧めします。

- ビーコンがデータセットに適しているかどうかを判断する
- ビーコンのタイプを選択する
- ビーコンの長さを選択する
- ビーコン名を選択する

データベースのために検索可能な暗号化ソリューションを計画する際には、ビーコンの一意性に関する次の要件に留意してください。

- すべての標準ビーコンには固有のビーコンソースが必要です

  同じ暗号化されたフィールドまたは仮想フィールドから複数の標準ビーコンを構築することはできません。

  ただし、単一の標準ビーコンを使用して複数の複合ビーコンを構築することはできます。
- 既存の標準ビーコンと重複するソースフィールドを含む仮想フィールドを作成しないようにしてください

  別の標準ビーコンを作成するために使用されたソースフィールドを含む仮想フィールドから標準ビーコンを構築すると、両方のビーコンのセキュリティが低下する可能性があります。

  詳細については、「仮想フィールドのセキュリティに関する考慮事項」を参照してください。

マルチテナンシーデータベースに関する考慮事項

マルチテナンシーデータベースで設定されたビーコンをクエリするには、レコードを暗号化したテナンシーに関連付けられた `branch-key-id` を格納するフィールドをクエリに含める必要があります。このフィールドは、ビーコンキーソースを定義する際に定義します。クエリが成功するには、このフィールドの値が、ビーコンの再計算に必要となる適切なビーコンキーマテリアルを識別する必要があります。

ビーコンを設定する前に、クエリに `branch-key-id` をどのように含めるかを決定する必要があります。クエリに `branch-key-id` を含めるさまざまな方法の詳細については、「マルチテナンシーデータベース内のビーコンのクエリ」を参照してください。

ビーコンのタイプの選択

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

検索可能な暗号化では、暗号化されたフィールドのプレーンテキストの値をビーコンでマッピングすることで、暗号化されたレコードを検索できます。設定するビーコンのタイプによって、実行できるクエリのタイプが決まります。

ビーコンを設定する前に、実行する必要があるクエリのタイプを特定して計画することを強くお勧めします。ビーコンを設定した後、暗号化されたフィールドを検索する前に、各ビーコンについてセカンダリインデックスを設定する必要があります。詳細については、「ビーコンを使用したセカンダリインデックスの設定」を参照してください。

ビーコンは、フィールドに書き込まれるプレーンテキストの値と、データベースに実際に格納される暗号化された値との間のマップを作成します。2 つの標準ビーコンの値は、基になる同じプレーンテキストが含まれている場合でも比較できません。2 つの標準ビーコンは、同じプレーンテキストの値について 2 つの異なる HMAC タグを生成します。その結果、標準ビーコンは次のクエリを実行できません。

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

上記のクエリは、複合ビーコンの署名付きの部分を比較する場合にのみ実行できます。ただし、CONTAINS 演算子は例外です。この演算子は、アセンブルされたビーコンに含まれる暗号化または署名されたフィールドの値全体を識別するために複合ビーコンで使用できます。署名付きの部分を比較する場合、オプションで暗号化された部分のプレフィックスを含めることができますが、フィールドの暗号化された値を含めることはできません。標準ビーコンおよび複合ビーコンが実行できるクエリのタイプの詳細については、「ビーコンのクエリ」を参照してください。

データベースのアクセスパターンを確認する際には、次の検索可能な暗号化ソリューションを検討してください。次の例では、暗号化およびクエリに関するさまざまな要件を満たすためにどのビーコンを設定すべきかを定義します。

標準ビーコン

標準ビーコンは、一致検索のみを実行できます。標準ビーコンを使用して、次のクエリを実行できます。

暗号化された単一フィールドをクエリする

暗号化されたフィールドについて特定の値を含むレコードを識別する場合は、標準ビーコンを作成します。

例

次の例では、生産施設の検査データを追跡する UnitInspection という名前のデータベースについて考えてみます。データベース内の各レコードには、work_id、inspection_date、inspector_id_last4、および unit と呼ばれるフィールドが含まれています。完全なインスペクター ID は 0 ~ 99,999,999 の数値です。ただし、データセットが統一的に分布するようにするために、inspector_id_last4 はインスペクターの ID の下 4 桁のみを格納します。データベース内の各フィールドは、プライマリキー work_id によって識別されます。inspector_id_last4 および unit フィールドは、暗号化アクションで ENCRYPT_AND_SIGN とマークされます。

UnitInspection データベース内のプレーンテキストエントリの例を次に示します。

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

レコード内の暗号化された単一フィールドをクエリする

> inspector_id_last4 フィールドを暗号化する必要があるが、完全一致検索のためにクエリする必要もある場合は、inspector_id_last4 フィールドから標準ビーコンを構築します。その後、標準ビーコンを使用してセカンダリインデックスを作成します。このセカンダリインデックスを使用して、暗号化された inspector_id_last4 フィールドをクエリできます。

標準ビーコンの設定については、「標準ビーコンの設定」を参照してください。

仮想フィールドをクエリする

仮想フィールドは、1 つ以上のソースフィールドから構築された概念的なフィールドです。暗号化されたフィールドの特定のセグメントについて一致検索を実行する場合、または複数のフィールドの連結に対して一致検索を実行する場合は、仮想フィールドから標準ビーコンを構築します。すべての仮

想フィールドには、少なくとも 1 つの暗号化されたソースフィールドが含まれている必要があります。

例

次の例では、Employees データベースの仮想フィールドを作成します。Employees データベース内のプレーンテキストレコードの例を次に示します。

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

暗号化されたフィールドのセグメントをクエリする

この例では、SSN フィールドは暗号化されています。

社会保障番号の下 4 桁を使用して SSN フィールドをクエリする場合は、クエリを実行するセグメントを識別する仮想フィールドを作成します。

Suffix(4) から構築された仮想 Last4SSN フィールドを使用すると、Last4SSN=0000 をクエリできます。この仮想フィールドを使用して、標準ビーコンを構築します。その後、標準ビーコンを使用してセカンダリインデックスを作成します。このセカンダリインデックスを使用して、仮想フィールドをクエリできます。このクエリは、指定した下 4 桁で終わる SSN の値を持つすべてのレコードを返します。

複数のフィールドの連結をクエリする

> **Note**
>
> 次の例は、仮想フィールドを使用して実行できる変換とクエリのタイプを示しています。アプリケーションでは、この例で使用されているフィールド例は、ビーコンの分布および相関の一意性に関する推奨事項を満たしていない可能性があります。

FirstName と LastName フィールドの連結に対して一致検索を実行する場合は、FirstName フィールドの最初の文字と、その後に続く LastName フィールドで構築される仮想 NameTag フィールドを作成できます (すべて小文字)。この仮想フィールドを使用して、標準ビーコンを構築します。その後、標準ビーコンを使用してセカンダリインデックスを作成します。このセカンダリインデックスを使用して、仮想フィールドの NameTag=mjones をクエリできます。

少なくとも 1 つのソースフィールドを暗号化する必要があります。FirstName または LastName のいずれかを暗号化することも、両方を暗号化することもできます。プレーンテキストのソースフィールドは、暗号化アクションSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTで SIGN_ONLYまたは としてマークする必要があります。

仮想フィールドとそれらを使用するビーコンの設定については、「仮想フィールドの作成」を参照してください。

複合ビーコン

複合ビーコンは、リテラルプレーンテキスト文字列と標準ビーコンからインデックスを作成し、複雑なデータベースオペレーションを実行します。複号ビーコンを使用して、次のクエリを実行できます。

単一のインデックスで暗号化されたフィールドの組み合わせをクエリする

単一のインデックスで暗号化されたフィールドの組み合わせをクエリする必要がある場合は、暗号化されたフィールドごとに構築された個々の標準ビーコンを組み合わせて単一のインデックスを形成する複合ビーコンを作成します。

複合ビーコンを設定した後、複合ビーコンをパーティションキーとして指定するセカンダリインデックスを作成して完全一致クエリを実行したり、ソートキーを使用してより複雑なクエリを実行したりできます。複合ビーコンをソートキーとして指定するセカンダリインデックスは、完全一致クエリや、よりカスタマイズされた複雑なクエリを実行できます。

例

次の例では、生産施設の検査データを追跡する UnitInspection という名前のデータベースについて考えてみます。データベース内の各レコードには、work_id、inspection_date、inspector_id_last4、および unit と呼ばれるフィールドが含まれています。完全なインスペクター ID は 0 ~ 99,999,999 の数値です。ただし、データセットが統一的に分布するようにするために、inspector_id_last4 はインスペクターの ID の下 4 桁の

みを格納します。データベース内の各フィールドは、プライマリキー work_id によって識別されます。inspector_id_last4 および unit フィールドは、暗号化アクションで ENCRYPT_AND_SIGN とマークされます。

UnitInspection データベース内のプレーンテキストエントリの例を次に示します。

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

暗号化されたフィールドの組み合わせに対して一致検索を実行する

inspector_id_last4.unit の完全一致検索のために UnitInspection データベースをクエリする場合は、まず inspector_id_last4 と unit のフィールドについての個別の標準ビーコンを作成します。その後、2 つの標準ビーコンから複合ビーコンを作成します。

複合ビーコンを設定した後、複合ビーコンをパーティションキーとして指定するセカンダリインデックスを作成します。このセカンダリインデックスを使用して、inspector_id_last4.unit の完全一致検索のためにクエリを実行します。例えば、このビーコンをクエリして、インスペクターが特定のユニットについて実行した検査のリストを検索できます。

暗号化されたフィールドの組み合わせに対して複雑なクエリを実行する

inspector_id_last4 と inspector_id_last4.unit のために UnitInspection データベースをクエリする場合は、まず inspector_id_last4 と unit のフィールドについて個別の標準ビーコンを作成します。その後、2 つの標準ビーコンから複合ビーコンを作成します。

複合ビーコンを設定した後、複合ビーコンをソートキーとして指定するセカンダリインデックスを作成します。このセカンダリインデックスを使用して、特定のインスペクターで始まるエントリや、特定のインスペクターによって検査された特定のユニット ID 範囲内のすべてのユニットのリストを検索するために UnitInspection データベースをクエリできます。inspector_id_last4.unit についての完全一致検索を実行することもできます。

複合ビーコンの設定については、「複合ビーコンの設定」を参照してください。

単一のインデックスで暗号化されたフィールドとプレーンテキストフィールドの組み合わせをクエリする

単一のインデックスで暗号化されたフィールドとプレーンテキストフィールドの組み合わせをクエリする必要がある場合は、個々の標準ビーコンとプレーンテキストフィールドを組み合わせて単一のインデックスを形成する複合ビーコンを作成します。複合ビーコンの構築に使用されるプレーンテキストフィールドは、暗号化アクションSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTでSIGN_ONLYまたは とマークする必要があります。

複合ビーコンを設定した後、複合ビーコンをパーティションキーとして指定するセカンダリインデックスを作成して完全一致クエリを実行したり、ソートキーを使用してより複雑なクエリを実行したりできます。複合ビーコンをソートキーとして指定するセカンダリインデックスは、完全一致クエリや、よりカスタマイズされた複雑なクエリを実行できます。

例

次の例では、生産施設の検査データを追跡する UnitInspection という名前のデータベースについて考えてみます。データベース内の各レコードには、work_id、inspection_date、inspector_id_last4、および unit と呼ばれるフィールドが含まれています。完全なインスペクター ID は 0 ~ 99,999,999 の数値です。ただし、データセットが統一的に分布するようにするために、inspector_id_last4 はインスペクターの ID の下 4 桁のみを格納します。データベース内の各フィールドは、プライマリキー work_id によって識別されます。inspector_id_last4 および unit フィールドは、暗号化アクションで ENCRYPT_AND_SIGN とマークされます。

UnitInspection データベース内のプレーンテキストエントリの例を次に示します。

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

フィールドの組み合わせに対して一致検索を実行する

特定の日付に特定のインスペクターによって実施された検査について UnitInspection データベースをクエリする場合は、まず inspector_id_last4 フィールドについての標準ビーコンを作成します。inspector_id_last4 フィールドは、暗号化アクションで ENCRYPT_AND_SIGN とマークされます。すべての暗号化された部分には独自の標準ビーコンが

必要です。inspection_date フィールドは SIGN_ONLY とマークされており、標準ビーコンは必要ありません。次に、inspection_date フィールドと inspector_id_last4 標準ビーコンから複合ビーコンを作成します。

複合ビーコンを設定した後、複合ビーコンをパーティションキーとして指定するセカンダリインデックスを作成します。このセカンダリインデックスを使用して、特定のインスペクターおよび検査日に完全に一致するレコードを検索するために、データベースをクエリします。例えば、ID が 8744 で終わるインスペクターが特定の日に実施したすべての検査のリストを検索するために、データベースをクエリできます。

フィールドの組み合わせに対して複雑なクエリを実行する

inspection_date の範囲内で実施される検査を検索するため、または inspector_id_last4 もしくは inspector_id_last4.unit によって制約されている特定の inspection_date に対して実施される検査を検索するためにデータベースをクエリする場合は、まず、inspector_id_last4 および unit フィールドについての個別の標準ビーコンを作成します。その後、プレーンテキスト inspection_date フィールドと 2 つの標準ビーコンから複合ビーコンを作成します。

複合ビーコンを設定した後、複合ビーコンをソートキーとして指定するセカンダリインデックスを作成します。このセカンダリインデックスを使用して、特定のインスペクターが特定の日付に実施した検査を検索するためにクエリを実行します。例えば、同日に検査されたすべてのユニットのリストを取得するために、データベースをクエリできます。または、指定された検査期間中に特定のユニットに対して実行されたすべての検査のリストを取得するために、データベースをクエリできます。

複合ビーコンの設定については、「複合ビーコンの設定」を参照してください。

ビーコンの長さの選択

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

検索可能な暗号化用に設定された暗号化されたフィールドに新しい値を書き込むと、 AWS Database Encryption SDK はプレーンテキスト値で HMAC を計算します。この HMAC 出力は、そのフィールドのプレーンテキストの値と 1 対 1 (1:1) で一致します。HMAC 出力は切り詰められ、複数

の個別のプレーンテキストの値が、切り詰められた同じ HMAC タグにマッピングされます。これらのコリジョン、つまり誤検知により、不正ユーザーは、プレーンテキストの値に関する特徴的な情報を識別しにくくなります。

各ビーコンについて生成される誤検知の平均数は、切り詰めた後に残っているビーコンの長さによって決まります。標準ビーコンを設定する場合、必要なのはビーコンの長さを定義することだけです。複合ビーコンは、その構築元となる標準ビーコンのビーコン長を使用します。

ビーコンはフィールドの暗号化状態を変更しません。ただし、ビーコンを使用する場合、クエリの効率性と、データの分布に関して明らかになる情報の量との間には、固有のトレードオフが存在します。

検索可能な暗号化の目標は、ビーコンを使用して暗号化されたデータをクエリすることにより、クライアント側の暗号化されたデータベースに関連するパフォーマンスコストを削減することです。ビーコンは、計算元の暗号化されたフィールドと一緒に格納されます。これは、データセットの分布に関する特徴的な情報を明らかにできることを意味します。極端な場合には、不正ユーザーが分布に関して明らかになった情報を分析し、それを使用してフィールドのプレーンテキストの値を特定できる可能性があります。ビーコンの長さを適切に選択すると、これらのリスクを軽減し、分布の機密性を維持するのに役立ちます。

脅威モデルを確認して、必要なセキュリティのレベルを決定します。例えば、データベースにアクセスできるが、プレーンテキストデータにはアクセスすべきではないユーザーが増えるほど、データセットの分布の機密性を保護する必要が高まる可能性があります。機密性を高めるには、ビーコンはより多くの誤検知を生成する必要があります。機密性が高まると、クエリのパフォーマンスが低下します。

セキュリティとパフォーマンス

- ビーコンが過度に長い場合には、生成される誤検知が過度に少なくなるため、データセットの分布に関する特徴的な情報が明らかになる可能性があります。
- ビーコンが過度に短い場合には、生成される誤検知が過度に多くなるため、データベースの広範なスキャンが必要になり、これに伴ってクエリのパフォーマンスコストが増加します。

ソリューションのために適切なビーコンの長さを決定する際には、クエリのパフォーマンスに必要以上に影響を及ぼすことなく、データのセキュリティを適切に維持できる長さを見つける必要があります。ビーコンによって維持されるセキュリティの量は、データセットの分布と、ビーコンの構築元となるフィールドの相関関係によって異なります。次のトピックは、ビーコンが統一的に分布しており、相関データが含まれていないことを前提としています。

トピック

ビーコンの長さの計算

ビーコンの長さはビット単位で定義され、切り詰め後に保持される HMAC タグのビット数を指します。推奨されるビーコンの長さは、データセットの分布、相関値の存在、セキュリティとパフォーマンスに関する具体的な要件によって異なります。データセットが統一的に分布している場合は、実装に最適なビーコンの長さを特定するために、次の方程式と手順を役立てることができます。これらの方程式は、ビーコンが生成する誤検知の平均数を推定するだけであり、データセット内のすべての一意の値が特定の数の誤検知を生成することを保証するものではありません。

> **Note**
>
> これらの方程式の有効性は、データセットの分布によって異なります。データセットが統一的に分布していない場合は、「ビーコンが適しているデータセット」を参照してください。一般に、データセットが統一的な分布から離れるほど、ビーコンを短くする必要があります。

1.

母集団を推定する

母集団は、標準ビーコンの構築元となるフィールド内の一意の値の想定される数であり、フィールドに格納される値の想定される合計数ではありません。例えば、従業員のミーティングの場所を特定する暗号化された Room フィールドについて考えてみましょう。Room フィールドには合計 100,000 の値が格納されることが想定されますが、従業員がミーティングのために予約できる部屋は 50 室しかありません。これは、Room フィールドに格納できる一意の値が 50 個しかないため、母集団が 50 であることを意味します。

> **Note**
>
> 標準ビーコンの構築元が仮想フィールドである場合、ビーコンの長さを計算するために使用される母集団は、仮想フィールドによって作成された一意の組み合わせの数です。

母集団を推定する際には、データセットの予測される増加を必ず考慮してください。ビーコンを持つ新しいレコードを書き込んだ後に、そのビーコンの長さを更新することはできません。脅威モデルと既存のデータベースソリューションを確認して、今後 5 年間にこのフィールドに格納されることが想定される一意の値の数の見積もりを作成します。

母集団は正確である必要はありません。まず、現在のデータベース内の一意の値の数を特定するか、または最初の 1 年間に格納されることが想定される一意の値の数を見積もります。次に、以下の質問を使用して、今後 5 年間で予測される一意の値の増加を判断します。

- 一意の値が 10 倍になることが想定されますか?
- 一意の値が 100 倍になることが想定されますか?
- 一意の値が 1,000 倍になることが想定されますか?

一意の値が 50,000 個である場合と 60,000 個である場合の差は大きくなく、推奨されるビーコンの長さは両方とも同じです。しかし、一意の値が 50,000 個である場合と 500,000 個である場合、その差は、推奨されるビーコンの長さに大きく影響します。

郵便番号や姓などの一般的なデータタイプの出現頻度について、公開データを確認することを検討してください。例えば、米国には 41,707 の郵便番号があります。使用する母集団は、独自のデータベースに比例する必要があります。データベース内の ZIPCode フィールドに米国全土のデータが含まれている場合は、ZIPCode フィールドに現在 41,707 個の一意の値がない場合でも、母集団を 41,707 と定義することが考えられます。データベース内の ZIPCode フィールドに 1 つの州のデータのみが含まれ、今後も 1 つの州のデータのみが含まれる場合は、母集団を 41,704 ではなく、その州の郵便番号の合計数として定義できます。

2. 想定されるコリジョン数の推奨範囲を計算する

特定のフィールドについての適切なビーコンの長さを決定するには、まず、想定されるコリジョン数の適切な範囲を特定する必要があります。想定されるコリジョン数は、特定の HMAC タグにマッピングされる一意のプレーンテキストの値の平均想定数を表します。1 つの一意のプレーンテキストの値について想定される誤検知の数は、想定されるコリジョン数より 1 少ない数となります。

想定されるコリジョン数は 2 以上、かつ、母集団の平方根未満にすることをお勧めします。次の方程式は、母集団に 16 個以上の一意の値がある場合にのみ機能します。

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

コリジョン数が 2 未満の場合、ビーコンが生成する誤検知が過度に少なくなります。想定されるコリジョンの最小数として 2 が推奨されます。これは、平均して、フィールド内のすべての一意の値が、他の 1 つの一意の値にマッピングされることによって、少なくとも 1 つの誤検知を生成することを意味するためです。

3. ビーコンの長さの推奨範囲を計算する

   想定されるコリジョンの最小数と想定されるコリジョンの最大数を特定したら、次の方程式を使用して適切なビーコンの長さの範囲を特定します。

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

   まず、想定されるコリジョン数が 2 (想定されるコリジョンの推奨最小数) である場合のビーコンの長さを求めます。

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

   その後、想定コリジョン数が母集団の平方根 (想定されるコリジョンの推奨最大数) である場合のビーコンの長さを求めます。

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

   この方程式によって生成される結果を切り捨ててビーコンの長さを算出します。例えば、方程式を解くとビーコンの長さが 15.6 になる場合、その値を 16 ビットになるように切り上げるのではなく、15 ビットになるように切り捨てることをお勧めします。

4. ビーコンの長さを選択する

   これらの方程式は、フィールドのビーコンの長さの推奨範囲を特定するだけです。データセットのセキュリティを維持するために、可能な場合は常に、ビーコンを短くすることをお勧めします。ただし、実際に使用するビーコンの長さは、脅威モデルによって決まります。脅威モデルを確認する際にパフォーマンス要件を考慮して、フィールドに最適なビーコンの長さを決定します。

   ビーコンを短くするとクエリのパフォーマンスが低下し、ビーコンを長くするとセキュリティが低下します。一般的に、データセットが不均一に分布している場合、または相関フィールドから個別のビーコンを構築する場合は、ビーコンをより短くして、データセットの分布に関して明らかになる情報の量を最小限に抑える必要があります。

脅威モデルを確認し、フィールドの分布に関して明らかになる特徴的な情報がセキュリティ全体に脅威を与えるものではないと判断した場合は、計算した推奨範囲よりもビーコンを長くすることを選択することもできます。例えば、フィールドのビーコンの長さの推奨範囲を計算したところ、9～16 ビットと算出されたとしても、パフォーマンスの低下を避けるために 24 ビットのビーコン長を使用することを選択できます。

ビーコンの長さは慎重に選択してください。ビーコンを持つ新しいレコードを書き込んだ後に、そのビーコンの長さを更新することはできません。

例

暗号化アクションで unit フィールドを ENCRYPT_AND_SIGN としてマークしたデータベースを考えてみましょう。unit フィールドの標準ビーコンを設定するには、unit フィールドについて想定される誤検知の数とビーコンの長さを決定する必要があります。

1. 母集団を推定する

   脅威モデルと現在のデータベースソリューションを確認した結果、unit フィールドには最終的に 100,000 個の一意の値が存在することになると想定されます。

   つまり、母集団 = 100,000 です。

2. 想定されるコリジョン数の推奨範囲を計算します。

   この例では、想定されるコリジョン数は 2～316 です。

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

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

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

3. ビーコンの長さの推奨範囲を計算します。

   この例では、ビーコンの長さは 9～16 ビットである必要があります。

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

a. 想定されるコリジョンの最小数がステップ 2 で特定された数である場合のビーコンの長さを計算します。

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

ビーコンの長さ = 15.6、または 15 ビット

b. 想定されるコリジョンの最大数がステップ 2 で特定された数である場合のビーコンの長さを計算します。

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

ビーコンの長さ = 8.3、または 8 ビット

4. セキュリティとパフォーマンスの要件に適したビーコンの長さを決定します。

15 未満のビットごとに、パフォーマンスコストとセキュリティが 2 倍になります。

- 16 ビット
  - 平均すると、それぞれの一意の値は他の 1.5 個のユニットにマッピングされます。
  - セキュリティ: 切り詰められた同じ HMAC タグを持つ 2 つのレコードは、同じプレーンテキストの値を持つ可能性が 66% あります。
  - パフォーマンス: クエリは、実際にリクエストした 10 件のレコードごとに 15 件のレコードを取得します。
- 14 ビット
  - 平均すると、それぞれの一意の値は他の 6.1 個のユニットにマッピングされます。
  - セキュリティ: 切り詰められた同じ HMAC タグを持つ 2 つのレコードは、同じプレーンテキストの値を持つ可能性が 33% あります。
  - パフォーマンス: クエリは、実際にリクエストした 10 件のレコードごとに 30 件のレコードを取得します。

ビーコン名の選択

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

すべてのビーコンは、一意のビーコン名によって識別されます。ビーコンが設定されると、ビーコン名は、暗号化されたフィールドをクエリする際に使用される名前となります。ビーコン名は、暗号化されたフィールドまたは仮想フィールドと同じ名前にすることができますが、暗号化されていないフィールドと同じ名前にすることはできません。2 つの異なるビーコンに同じビーコン名を付けることはできません。

ビーコンに名前を付けて設定する方法を示す例については、「ビーコンの設定」を参照してください。

標準ビーコンの命名

標準ビーコンに名前を付ける場合は、可能な場合は常に、ビーコン名をビーコンソースに解決することを強くお勧めします。これは、ビーコン名と、標準ビーコンの構築元となる暗号化されたフィールドまたは仮想フィールドの名前が同じであることを意味します。例えば、`LastName` という名前の暗号化されたフィールドについての標準ビーコンを作成する場合、ビーコン名も `LastName` である必要があります。

ビーコン名がビーコンソースと同じ場合、設定からビーコンソースを省略できます。 AWS Database Encryption SDK は自動的にビーコン名をビーコンソースとして使用します。

ビーコンの設定

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

検索可能な暗号化をサポートするビーコンには 2 つのタイプがあります。標準ビーコンは、一致検索を実行します。これらは、データベースで検索可能な暗号化を実装する最も簡単な方法です。複合ビーコンは、リテラルプレーンテキスト文字列と標準ビーコンを組み合わせて、より複雑なクエリを実行します。

ビーコンは、データが入力されていない新しいデータベースに実装されるように設計されています。既存のデータベースで設定されたビーコンは、データベースに書き込まれる新しいレコードのみをマッピングします。ビーコンはフィールドのプレーンテキストの値から計算されます。フィールドが暗号化されると、ビーコンは既存のデータをマッピングできなくなります。ビーコンを持つ新しいレ

コードを書き込んだ後に、そのビーコンの設定を更新することはできません。ただし、レコードに追加する新しいフィールドに新しいビーコンを追加できます。

アクセスパターンを決定したら、データベース実装の 2 番目のステップとしてビーコンを設定する必要があります。次に、すべてのビーコンを設定したら、AWS KMS 階層キーリングの作成、ビーコンバージョンの定義、各ビーコンのセカンダリインデックスの設定、暗号化アクションの定義、データベースと AWS Database Encryption SDK クライアントの設定を行う必要があります。詳細については、「ビーコンの使用」を参照してください。

ビーコンのバージョンをより簡単に定義できるように、標準ビーコンと複合ビーコンのリストを作成することをお勧めします。作成した各ビーコンを、設定時にそれぞれの標準ビーコンリストまたは複合ビーコンリストに追加します。

トピック

- 標準ビーコンの設定
- 複合ビーコンの設定
- 設定例

標準ビーコンの設定

標準ビーコンは、データベースで検索可能な暗号化を実装する最も簡単な方法です。単一の暗号化されたフィールドまたは仮想フィールドについてのみ一致検索を実行できます。

設定構文の例

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()?,
```

標準ビーコンを設定するには、次の値を指定します。

ビーコン名

暗号化されたフィールドをクエリする際に使用する名前。

ビーコン名は、暗号化されたフィールドまたは仮想フィールドと同じ名前にすることができますが、暗号化されていないフィールドと同じ名前にすることはできません。可能な場合は常に、標準ビーコンの構築元となる暗号化されたフィールドまたは仮想フィールドの名前を使用することを強くお勧めします。2 つの異なるビーコンに同じビーコン名を付けることはできません。実装に最適なビーコン名を決定する方法については、「ビーコン名の選択」を参照してください。

ビーコンの長さ

切り詰めた後に保持されるビーコンのハッシュ値のビット数。

ビーコンの長さによって、特定のビーコンによって生成される誤検知の平均数が決まります。実装に適切なビーコンの長さを決定する方法の詳細とヘルプについては、「ビーコンの長さの決定」を参照してください。

ビーコンソース (オプション)

標準ビーコンの構築元となるフィールド。

ビーコンソースは、フィールド名、またはネストされたフィールドの値を参照するインデックスである必要があります。ビーコン名がビーコンソースと同じ場合、設定からビーコンソースを省略でき、 AWS Database Encryption SDK は自動的にビーコン名をビーコンソースとして使用します。

仮想フィールドの作成

仮想フィールドを作成するには、仮想フィールドの名前とソースフィールドのリストを指定する必要があります。ソースフィールドを仮想部分のリストに追加する順序によって、仮想フィールドを構築するためにこれらのソースフィールドが連結される順序が決まります。次の例では、2 つのソースフィールド全体を連結して、仮想フィールドを作成します。

> **Note**
>
> データベースに入力する前に、仮想フィールドが期待される結果を生成することを確認することをお勧めします。詳細については、「ビーコン出力のテスト」を参照してください。

Java

完全なコード例を参照: 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

完全なコード例を参照: VirtualBeaconSearchableEncryptionExample.cs

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

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

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

Rust

完全なコード例を参照: 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];
```

ソースフィールドの特定のセグメントを使用して仮想フィールドを作成するには、ソースフィールドを仮想部分のリストに追加する前に、その変換を定義する必要があります。

仮想フィールドのセキュリティに関する考慮事項

ビーコンはフィールドの暗号化状態を変更しません。ただし、ビーコンを使用する場合、クエリの効率性と、データの分布に関して明らかになる情報の量との間には、固有のトレードオフが存在します。ビーコンを設定する方法によって、そのビーコンによって維持されるセキュリティのレベルが決まります。

既存の標準ビーコンと重複するソースフィールドを含む仮想フィールドを作成しないようにしてください。標準ビーコンを作成するために既に使用されているソースフィールドを含む仮想フィールドを作成すると、両方のビーコンのセキュリティレベルが低下する可能性があります。セキュリティが低下する程度は、追加のソースフィールドによって追加されるエントロピーのレベルによって異なります。エントロピーのレベルは、追加のソースフィールド内の固有の値の分布と、追加のソースフィールドが仮想フィールドの全体的なサイズに寄与するビット数によって決まります。

母集団とビーコンの長さを使用して、仮想フィールドのソースフィールドがデータセットのセキュリティを維持するかどうかを判断できます。母集団は、フィールド内の一意の値の想定数です。母集団は正確である必要はありません。フィールドの母集団の推定については、「母集団の推定」を参照してください。

仮想フィールドのセキュリティを確認する際には、次の例を考慮してください。

- Beacon1 は FieldA から構築されます。FieldA の母集団は、$2^{(\text{Beacon1 の長さ})}$ よりも大きいです。
- Beacon2 は VirtualField から構築されます。これは、FieldA、FieldB、FieldC、および FieldD から構築されます。FieldB、FieldC、および FieldD を合わせると、母集団は 2^N よりも大きくなります

次のステートメントが真の場合、Beacon2 は Beacon1 と Beacon2 の両方のセキュリティを維持します。

```
N ≥ (Beacon1 length)/2
```

と

```
N ≥ (Beacon2 length)/2
```

ビーコンスタイルの定義

標準ビーコンを使用して、暗号化されたフィールドまたは仮想フィールドの等価検索を実行できます。または、複合ビーコンを構築して、より複雑なデータベースオペレーションを実行することもできます。標準ビーコンの整理と管理に役立つように、 AWS Database Encryption SDK には、標準ビーコンの用途を定義する以下のオプションビーコンスタイルが用意されています。

> **Note**
>
> ビーコンスタイルを定義するには、 AWS Database Encryption SDK のバージョン 3.2 以降を使用する必要があります。ビーコン設定にビーコンスタイルを追加する前に、すべてのリーダーに新しいバージョンをデプロイします。

PartOnly

として定義された標準ビーコンは、複合ビーコンの暗号化された部分を定義するためにPartOnlyのみ使用できます。PartOnly 標準ビーコンを直接クエリすることはできません。

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

デフォルトでは、すべての標準ビーコンはビーコン計算用の一意の HMAC キーを生成します。そのため、2 つの異なる標準ビーコンから暗号化されたフィールドに対して等価検索を実行することはできません。として定義された標準ビーコンSharedは、別の標準ビーコンの HMAC キーを計算に使用します。

たとえば、beacon1フィールドとbeacon2フィールドを比較する必要がある場合は、 を、 の計算に の HMAC キーを使用するSharedビーコンbeacon2として定義beacon1します。

> **Note**
>
> Shared ビーコンを設定する前に、セキュリティとパフォーマンスのニーズを考慮してください。 Shared ビーコンは、データセットの分布に関して識別できる統計情報の量を増やす可能性があります。たとえば、どの共有フィールドに同じプレーンテキスト値が含まれているかを明らかにすることができます。

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

デフォルトでは、フィールド値がセットの場合、 AWS Database Encryption SDK はセットの単一の標準ビーコンを計算します。そのため、 が暗号化されたフィールドCONTAINS(*a*, :*value*)*a*であるクエリを実行することはできません。として定義された標準ビーコンは、セットの各要素の個々の標準ビーコン値をAsSet計算し、ビーコン値をセットとして項目に保存します。これにより、 AWS Database Encryption SDK はクエリ を実行できますCONTAINS(*a*, :*value*)。

AsSet 標準ビーコンを定義するには、セット内の要素が同じビーコンの長さを使用できるように、同じ母集団の要素である必要があります。ビーコン値の計算時に衝突が発生した場合、ビーコンセットの要素数がプレーンテキストセットよりも少なくなる可能性があります。

> **Note**
>
> AsSet ビーコンを設定する前に、セキュリティとパフォーマンスのニーズを考慮してください。 AsSet ビーコンは、データセットの分布に関して識別できる統計情報の量を増やす可能性があります。たとえば、プレーンテキストセットのサイズが明らかになる場合があります。

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

として定義された標準ビーコンは、 関数Sharedと AsSet関数SharedSetを組み合わせて、セットとフィールドの暗号化された値に対して等価検索を実行できるようにします。これにより、 AWS Database Encryption SDK はCONTAINS(*a*, *b*)、 *a*が暗号化されたセットであり、 が暗号化されたフィールド*b*であるクエリを実行できます。

> **Note**
>
> Shared ビーコンを設定する前に、セキュリティとパフォーマンスのニーズを考慮してください。 SharedSet ビーコンは、データセットの分布に関して識別できる統計情報の量を増やす可能性があります。たとえば、プレーンテキストセットのサイズや、同じプレーンテキスト値を含む共有フィールドが明らかになる場合があります。

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()?
```

複合ビーコンの設定

複合ビーコンは、リテラルプレーンテキスト文字列と標準ビーコンを組み合わせて、単一のインデックスから 2 つの異なるレコードタイプをクエリしたり、ソートキーを使用してフィールドの組み合わせをクエリしたりするなど、複雑なデータベースオペレーションを実行します。複合ビーコンは、ENCRYPT_AND_SIGN、、SIGN_ONLYおよびSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドから構築できます。複合ビーコンに含まれる暗号化されたフィールドごとに標準ビーコンを作成する必要があります。

> **Note**
>
> データベースに入力する前に、複合ビーコンが期待される結果を生成することを確認することをお勧めします。詳細については、「ビーコン出力のテスト」を参照してください。

設定構文の例

Java

複合ビーコン設定

次の例では、複合ビーコン設定内で暗号化および署名されたパートリストをローカルに定義します。

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

ビーコンバージョン定義

次の例では、ビーコンバージョンで暗号化および署名されたパートリストをグローバルに定義します。ビーコンバージョンの定義の詳細については、「ビーコンの使用」を参照してください。

```
 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

完全なコードサンプルを参照: BeaconConfig.cs

複合ビーコン設定

次の例では、複合ビーコン設定内で暗号化および署名されたパートリストをローカルに定義します。

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

ビーコンバージョン定義

次の例では、ビーコンバージョンで暗号化および署名されたパートリストをグローバルに定義します。ビーコンバージョンの定義の詳細については、「ビーコンの使用」を参照してください。

```
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

完全なコードサンプル「beacon_config.rs」を参照してください。

複合ビーコン設定

次の例では、複合ビーコン設定内で暗号化および署名されたパートリストをローカルに定義します。

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

ビーコンバージョン定義

次の例では、ビーコンバージョンで暗号化および署名されたパートリストをグローバルに定義します。ビーコンバージョンの定義の詳細については、「ビーコンの使用」を参照してください。

```
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];
```

暗号化されたパートと署名されたパートは、ローカルまたはグローバルに定義されたリストで定義できます。可能な限り、ビーコンバージョンのグローバルリストで暗号化および署名されたパートを定義することをお勧めします。暗号化されたパートと署名されたパートをグローバルに定義することで、各パートを 1 回定義し、そのパートを複数の複合ビーコン設定で再利用できます。暗号化または署名されたパートを 1 回だけ使用する場合は、複合ビーコン設定のローカルリストで定義できます。コンストラクタリストでは、ローカルパートとグローバルパートの両方を参照できます。

暗号化および署名されたパートリストをグローバルに定義する場合は、複合ビーコンが複合ビーコン設定のフィールドをアセンブルできるすべての方法を識別するコンストラクタパートのリストを提供する必要があります。

> **Note**
>
> 暗号化および署名されたパートリストをグローバルに定義するには、バージョン 3.2 以降の AWS Database Encryption SDK を使用する必要があります。新しいパートをグローバルに定義する前に、すべてのリーダーに新しいバージョンをデプロイします。
> 既存のビーコン設定を更新して、暗号化および署名されたパートリストをグローバルに定義することはできません。

複合ビーコンを設定するには、次の値を指定します。

ビーコン名

暗号化されたフィールドをクエリする際に使用する名前。

ビーコン名は、暗号化されたフィールドまたは仮想フィールドと同じ名前にすることができますが、暗号化されていないフィールドと同じ名前にすることはできません。2 つのビーコンを同じ名前にすることはできません。実装に最適なビーコン名を決定する方法については、「ビーコン名の選択」を参照してください。

分割文字

複合ビーコンを設定する部分を分離するために使用される文字。

分割文字は、複合ビーコンの構築元となるフィールドのプレーンテキストの値に出現することはできません。

暗号化されたパートリスト

複合ビーコンに含まれる ENCRYPT_AND_SIGN フィールドを識別します。

各部分には、名前とプレフィックスが含まれている必要があります。部分の名前は、暗号化されたフィールドから構築された標準ビーコンの名前である必要があります。プレフィックスには任意の文字列を指定できますが、一意である必要があります。暗号化された部分は、署名付きの部分と同じプレフィックスを持つことはできません。複合ビーコンによって提供される部分と他の部分を区別する短い値を使用することをお勧めします。

可能な限り、暗号化されたパートをグローバルに定義することをお勧めします。1 つの複合ビーコンでのみ使用する場合は、暗号化された部分をローカルで定義することを検討してください。ローカルに定義された暗号化されたパートは、グローバルに定義された暗号化されたパートと同じプレフィックスまたは名前を持つことはできません。

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()?
];
```

署名付きパートリスト

複合ビーコンに含まれる署名付きフィールドを識別します。

> **Note**
>
> 署名付きパートはオプションです。署名付きパートを参照しない複合ビーコンを設定できます。

各部分には、名前、ソース、プレフィックスが含まれている必要があります。ソースは、パートが識別する SIGN_ONLYまたは SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドです。ソースは、フィールド名、またはネストされたフィールドの値を参照するインデックスである必要があります。パーツ名がソースを識別する場合、ソースを省略すると、 AWS Database Encryption SDK は自動的にその名前をソースとして使用します。可能な場合は常に、部分名としてソースを指定することをお勧めします。プレフィックスには任意の文字列を指定できますが、一意である必要があります。署名付きの部分は、暗号化された部分と同じプレフィックスを持つことはできません。複合ビーコンによって提供される部分と他の部分を区別する短い値を使用することをお勧めします。

可能な限り、署名付きパートをグローバルに定義することをお勧めします。署名付きパートを 1 つの複合ビーコンでのみ使用する場合は、ローカルで定義することを検討してください。ローカルに定義された署名付きパートは、グローバルに定義された署名付きパートと同じプレフィックスまたは名前を持つことはできません。

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()?,
];
```

コンストラクタリスト

暗号化および署名付きの部分を複合ビーコンによってアセンブルするさまざまな方法を定義するコンストラクターを識別します。コンストラクタリストでは、ローカルパートとグローバルパートの両方を参照できます。

グローバルに定義された暗号化および署名された部分から複合ビーコンを構築する場合は、コンストラクタリストを指定する必要があります。

グローバルに定義された暗号化または署名されたパートを使用して複合ビーコンを構築しない場合、コンストラクタリストはオプションです。コンストラクタリストを指定しない場合、 AWS Database Encryption SDK は次のデフォルトのコンストラクタを使用して複合ビーコンをアセンブルします。

- すべての署名付きの部分 (署名付きの部分のリストに追加された順)
- 暗号化されたすべての部分 (暗号化された部分のリストに追加された順)
- すべての部分は必須です

コンストラクタ

各コンストラクターは、複合ビーコンをアセンブルする 1 つの方法を定義するコンストラクター部分の順序付きリストです。コンストラクター部分はリストに追加された順序で結合され、各部分は指定された分割文字で区切られます。

各コンストラクター部分は、暗号化された部分または署名付きの部分に名前を付け、その部分がコンストラクター内で必須であるか、またはオプションであるかを定義します。例えば、`Field1`、`Field1.Field2`、および `Field1.Field2.Field3` で複合ビーコンをクエリする場合は、`Field2` および `Field3` をオプションとしてマークし、コンストラクターを 1 つ作成します。

各コンストラクターには、少なくとも 1 つの必須部分が必要です。クエリで `BEGINS_WITH` 演算子を使用できるように、各コンストラクターの最初の部分を必須にすることをお勧めします。

コンストラクターは、必要な部分がすべてレコード内に存在する場合に成功します。新しいレコードを書き込む際に、複合ビーコンはコンストラクターのリストを使用して、指定された値からビーコンをアセンブルできるかどうかを判断します。コンストラクターがコンストラクターのリストに追加された順序でビーコンのアセンブルを試み、成功した最初のコンストラクターを使用します。コンストラクターが成功しない場合、ビーコンはレコードに書き込まれません。

すべてのリーダーとライターは、クエリの結果が確実に正しくなるようにコンストラクターの同じ順序を指定する必要があります。

独自のコンストラクターのリストを指定するには、次の手順を使用します。

1. 暗号化部分と署名付きの部分ごとにコンストラクター部分を作成し、その部分が必須かどうかを定義します。

   コンストラクター部分の名前は、標準ビーコンの名前、またはそれが表す署名されたフィールドの名前である必要があります。

   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. ステップ 1 で作成したコンストラクター部分を使用して、複合ビーコンをアセンブルする可能な方法ごとにコンストラクターを作成します。

   例えば、Field1.Field2.Field3 と Field4.Field2.Field3 をクエリする場合は、2 つのコンストラクターを作成する必要があります。Field1 と Field4 は、2 つの別個のコンストラクターで定義されているため、両方とも必須にすることができます。

   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. ステップ 2 で作成したすべてのコンストラクターを含むコンストラクターのリストを作成します。

   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. 複合ビーコンを作成するconstructorListときに を指定します。

設定例

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

次の例は、標準ビーコンと複合ビーコンを設定する方法を示しています。次の設定は、ビーコンの長さを指定しません。設定用に適切なビーコンの長さを決定する方法については、「ビーコンの長さを選択する」を参照してください。

ビーコンの設定方法と使用方法を示す完全なコード例については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリにある Java、.NET、および Rust の検索可能な暗号化の例を参照してください。

トピック

- 標準ビーコン
- 複合ビーコン

標準ビーコン

完全一致を検索するために inspector_id_last4 フィールドをクエリする場合は、次の設定を使用して標準ビーコンを作成します。

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];
```

複合ビーコン

inspector_id_last4 および inspector_id_last4.unit で UnitInspection データベースをクエリする場合は、次の設定で複合ビーコンを作成します。この複合ビーコンには暗号化された部分のみが必要です。

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()?];
```

ビーコンの使用

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

ビーコンを使用すると、クエリ対象のデータベース全体を復号することなく、暗号化されたレコードを検索できます。ビーコンは、データが入力されていない新しいデータベースに実装されるように設計されています。既存のデータベースで設定されたビーコンは、データベースに書き込まれる新しいレコードのみをマッピングします。ビーコンはフィールドのプレーンテキストの値から計算されます。フィールドが暗号化されると、ビーコンは既存のデータをマッピングできなくなります。ビーコンを持つ新しいレコードを書き込んだ後に、そのビーコンの設定を更新することはできません。ただし、レコードに追加する新しいフィールドに新しいビーコンを追加できます。

ビーコンを設定した後、データベースにデータを入力し、ビーコンをクエリする前に、次のステップを完了する必要があります。

1. AWS KMS 階層キーリングを作成する

   検索可能な暗号化を使用するには、AWS KMS 階層キーリングを使用して、レコードを保護するために使用されるデータキーを生成、暗号化、および復号する必要があります。

   ビーコンを設定した後、階層キーリングの前提条件をアセンブルし、階層キーリングを作成します。

   階層キーリングが必要な理由の詳細については、「検索可能な暗号化のための階層キーリングの使用」を参照してください。

2. ビーコンのバージョンを定義する

   keyStore、keySource、設定したすべての標準ビーコンのリスト、設定したすべての複合ビーコンのリスト、暗号化されたパートのリスト、署名されたパートのリスト、ビーコンバージョンを指定します。ビーコンのバージョンとして 1 を指定する必要があります。keySource の定義に関するガイダンスについては、「ビーコンキーソースの定義」を参照してください。

次の Java の例では、シングルテナンシーデータベースのビーコンバージョンを定義します。マルチテナンシーデータベースのビーコンバージョンの定義については、「マルチテナンシーデータベースの検索可能な暗号化」を参照してください。

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. セカンダリインデックスを設定する

   ビーコンを設定した後、暗号化されたフィールドを検索する前に、各ビーコンを反映するセカンダリインデックスを設定する必要があります。詳細については、「ビーコンを使用したセカンダリインデックスの設定」を参照してください。

4. 暗号化アクションを定義する

   標準ビーコンの構築に使用されるすべてのフィールドを ENCRYPT_AND_SIGN とマークする必要があります。ビーコンの構築に使用される他のすべてのフィールドは、 SIGN_ONLYまたは とマークする必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

5. AWS Database Encryption SDK クライアントを設定する

   DynamoDB テーブルのテーブル項目を保護する AWS Database Encryption SDK クライアントを設定するには、DynamoDB の Java クライアント側の暗号化ライブラリ」を参照してください。

ビーコンのクエリ

設定するビーコンのタイプによって、実行できるクエリのタイプが決まります。標準ビーコンは、フィルター式を使用して一致検索を実行します。複合ビーコンは、リテラルプレーンテキスト文字列と標準ビーコンを組み合わせて、複雑なクエリを実行します。暗号化されたデータをクエリする際には、ビーコン名で検索します。

2 つの標準ビーコンの値は、基になる同じプレーンテキストが含まれている場合でも比較できません。2 つの標準ビーコンは、同じプレーンテキストの値について 2 つの異なる HMAC タグを生成します。その結果、標準ビーコンは次のクエリを実行できません。

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

複合ビーコンは次のクエリを実行できます。

- BEGINS_WITH(*a*)。ここで、*a* は、アセンブルされた複合ビーコンの先頭のフィールドの値全体を反映します。BEGINS_WITH 演算子を使用して、特定の部分文字列で始まる値を識別することはできません。ただし、BEGINS_WITH(*S_*) を使用することはできます。ここで、*S_* は、アセンブルされた複合ビーコンの先頭の部分のプレフィックスを反映します。
- CONTAINS(*a*)。ここで、*a* は、アセンブルされた複合ビーコンが含むフィールドの値全体を反映します。CONTAINS 演算子を使用して、セット内の特定の部分文字列または値を含むレコードを識別することはできません。

  例えば、クエリ CONTAINS(*path*, "*a*" を実行することはできません。ここで、*a* は、セット内の値を反映します。
- 複合ビーコンの署名付きの部分を比較できます。署名付きの部分を比較する場合、必要に応じて、暗号化部分のプレフィックスを 1 つ以上の署名付きの部分に付加できますが、暗号化されたフィールドの値をクエリに含めることはできません。

  例えば、署名付きの部分を比較し、*signedField1* = *signedField2* または *value* IN (*signedField1*, *signedField2*, ...) をクエリできます。

  署名付きの部分と暗号化部分のプレフィックスを、*signedField1.A_* = *signedField2.B_* に対するクエリによって比較することもできます。

- *field* BETWEEN *a* AND *b*。ここで、*a* と *b* は署名付きの部分です。必要に応じて、暗号化部分のプレフィックスを 1 つ以上の署名付きの部分に付加できますが、暗号化されたフィールドの値をクエリに含めることはできません。

複合ビーコンに対するクエリに含める各部分のプレフィックスを含める必要があります。
例えば、encryptedField および signedField の 2 つのフィールドから複合ビーコン compoundBeacon を構築した場合、ビーコンをクエリする際に、これらの 2 つの部分について設定されたプレフィックスを含める必要があります。

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue
```

マルチテナンシーデータベースの検索可能な暗号化

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

データベースで検索可能な暗号化を実装するには、AWS KMS 階層キーリングを使用する必要があります。 AWS KMS 階層キーリングは、レコードの保護に使用されるデータキーを生成、暗号化、復号します。また、ビーコンを生成するために使用されるビーコンキーも作成します。マルチテナントデータベースで AWS KMS 階層キーリングを使用する場合、テナントごとに個別のブランチキーとビーコンキーがあります。マルチテナンシーデータベース内の暗号化されたデータをクエリするには、クエリしているビーコンを生成するために使用されたビーコンキーマテリアルを特定する必要があります。詳細については、「the section called “検索可能な暗号化のための階層キーリングの使用”」を参照してください。

マルチテナンシーデータベースのビーコンバージョンを定義する場合は、設定したすべての標準ビーコンのリスト、設定したすべての複合ビーコンのリスト、ビーコンバージョン、および keySource を指定します。ビーコンキーソースを MultiKeyStore として定義し、keyFieldName、ローカルビーコンキーキャッシュのキャッシュ Time To Live、およびローカルビーコンキーキャッシュの最大キャッシュサイズを含める必要があります。

署名付きビーコンを設定した場合は、それらを compoundBeaconList に含める必要があります。署名付きビーコンは、 SIGN_ONLYフィールドと SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドに対してインデックスを作成し、複雑なクエリを実行する複合ビーコンの一種です。

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

keyFieldName は、特定のテナンシーについて生成されたビーコンに使用されるビーコンキーに関連付けられた branch-key-id を格納するフィールドの名前を定義します。

新しいレコードをデータベースに書き込むと、そのレコードについてのビーコンを生成するために使用されるビーコンキーを識別する branch-key-id がこのフィールドに格納されます。

デフォルトでは、keyField はデータベースに明示的に格納されない概念的なフィールドです。AWS Database Encryption SDK は、マテリアルの説明で暗号化されたデータキーbranch-key-idから を識別し、複合ビーコンと署名付きビーコンで参照keyFieldできるように概念に値を保存します。マテリアルの説明は署名されているため、概念的な keyField は署名付きの部分とみなされます。

暗号化アクションkeyFieldに を SIGN_ONLYまたは SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドとして含めて、 フィールドをデータベースに明示的に保存することもできます。これを実行するには、データベースにレコードを書き込むたびに、branch-key-id を手動で keyField に含める必要があります。

マルチテナンシーデータベース内のビーコンのクエリ

ビーコンをクエリするには、クエリに keyField を含めて、ビーコンの再計算に必要となる適切なビーコンキーマテリアルを識別する必要があります。レコードのビーコンを生成するために使用されるビーコンキーに関連付けられた branch-key-id を指定する必要があります。ブランチキー ID サプライヤーのテナンシーの branch-key-id を識別するフレンドリ名を指定することはできません。次の方法でクエリに keyField を含めることができます。

複合ビーコン

keyField をレコードに明示的に格納するかどうかにかかわらず、複合ビーコンに署名付きの部分として keyField を直接含めることができます。keyField の署名付きの部分は必須である必要があります。

例えば、encryptedField および signedField の 2 つのフィールドから複合ビーコン compoundBeacon を構築する場合は、署名付きの部分として keyField も含める必要があります。これにより、compoundBeacon に対して次のクエリを実行できるようになります。

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

署名付きビーコン

AWS Database Encryption SDK は、標準ビーコンと複合ビーコンを使用して、検索可能な暗号化ソリューションを提供します。これらのビーコンには、少なくとも 1 つの暗号化されたフィールドが含まれている必要があります。ただし、 AWS Database Encryption SDK は、プレーンテキストSIGN_ONLYとSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTフィールドから完全に設定できる署名付きビーコンもサポートしています。

署名付きビーコンは単一の部分から構築できます。keyField をレコードに明示的に格納するかどうかにかかわらず、keyField から署名付きビーコンを構築し、それを使用して、keyField 署名付きビーコンに対するクエリと、他のビーコンの 1 つに対するクエリを組み合わせる複合クエリを作成できます。例えば、次のクエリを実行できます。

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

署名付きビーコンの設定については、「署名付きビーコンの作成」を参照してください

keyField に対する直接的なクエリの実行

暗号化アクションで keyField を指定し、そのフィールドをレコードに明示的に格納した場合は、ビーコンに対するクエリと、keyField に対するクエリを組み合わせた複合クエリを作成できます。標準ビーコンをクエリする場合は、keyField に対して直接クエリを実行することを選択できます。例えば、次のクエリを実行できます。

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

AWS Database Encryption SDK for DynamoDB

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK for DynamoDB は、Amazon DynamoDB 設計にクライアント側の暗号化を含めることができるソフトウェアライブラリです。 AWS Database Encryption SDK for DynamoDB は属性レベルの暗号化を提供し、暗号化する項目と、データの信頼性を保証する署名に含める項目を指定できます。伝送中および保管時の機密データを暗号化することで、 AWSなどのサードパーティーがお客様のプレーンテキストデータを使用することはできません。

> **Note**
>
> AWS Database Encryption SDK は PartiQL をサポートしていません。

DynamoDB では、テーブルは項目のコレクションです。各項目は、属性の集合です。各属性には名前と値があります。 AWS Database Encryption SDK for DynamoDB は、属性の値を暗号化します。次に、属性に対する署名を計算します。暗号化アクションでどの属性値を暗号化し、署名にどの属性値を含めるかを指定します。

この章のトピックでは、暗号化されるフィールド、クライアントのインストールと設定に関するガイダンス、開始に役立つ Java の例など、 AWS Database Encryption SDK for DynamoDB の概要について説明します。

トピック

- クライアント側とサーバー側の暗号化
- どのフィールドが暗号化および署名されますか?
- DynamoDB での検索可能な暗号化
- データモデルの更新
- AWS Database Encryption SDK for DynamoDB で利用可能なプログラミング言語
- レガシー DynamoDB 暗号化クライアント

クライアント側とサーバー側の暗号化

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK for DynamoDB は、テーブルデータをデータベースに送信する前に暗号化するクライアント側の暗号化をサポートしています。ただし、DynamoDB では、ディスクに保管されているテーブルを透過的に暗号化するサーバー側の保管時の暗号化機能を提供しており、ユーザーがテーブルにアクセスすると復号します。

選択するツールは、データの重要度と、アプリケーションのセキュリティ要件に応じて異なります。AWS Database Encryption SDK for DynamoDB と保管時の暗号化の両方を使用できます。暗号化されて署名された項目を DynamoDB に送信しても、保護されている項目は DynamoDB によって認識されません。バイナリ属性値を含む従来のテーブル項目を検出します。

サーバー側の保管時の暗号化

DynamoDB では、保管時の暗号化がサポートされています。これは、テーブルがディスクに保持されるときに DynamoDB がテーブルを透過的に暗号化し、ユーザーがテーブルデータにアクセスするときにテーブルを復号するサーバー側の暗号化機能です。

AWS SDK を使用して DynamoDB を操作する場合、デフォルトでは、データは HTTPS 接続を介して転送中に暗号化され、DynamoDB エンドポイントで復号され、DynamoDB に保存される前に再暗号化されます。

- デフォルトでの暗号化。DynamoDB は、書き込まれる際に、すべてのテーブルを透過的に暗号化および復号します。保管時の暗号化を有効または無効にするオプションはありません。
- DynamoDB は暗号化キーを作成および管理します。各テーブルの一意のキーは、AWS KMS key で保護されるため、AWS Key Management Service (AWS KMS) が未暗号化のままになることはありません。デフォルトでは、DynamoDB は DynamoDB サービス アカウントの AWS 所有のキー を使用しますが、一部またはすべてのテーブルを保護するために、自分のアカウントの AWS マネージドキー または カスタマーマネージドキー を選択することもできます。
- テーブルデータはすべて、ディスク上で暗号化されます。暗号化されたテーブルがディスクに保存されると、DynamoDB は、プライマリキーおよびローカルとグローバルのセカンダリインデックスなど、すべてのテーブルデータを暗号化します。テーブルにソートキーが存在する場合、範囲の境界線を示すソートキーの一部が、プレーンテキスト形式でテーブルメタデータに保存されます。

- テーブルに関連するオブジェクトも暗号化されます。保管時の暗号化は、永続的なメディアに書き込まれるたびに、DynamoDBストリーム、グローバルテーブル、バックアップを保護します。
- アクセスすると、項目は復号されます。テーブルがアクセスされるとき、DynamoDB は、ターゲット項目を含むテーブル部分を復号し、プレーンテキスト形式で項目を返します。

AWS Database Encryption SDK for DynamoDB

クライアント側の暗号化では、ソースから DynamoDB のストレージまで、伝送時および保管時のデータをエンドツーエンド保護します。プレーンテキストデータは、以下を含む第三者に公開されることはありません AWS。 AWS Database Encryption SDK for DynamoDB を新しい DynamoDB テーブルで使用するか、既存の Amazon DynamoDB テーブルを AWS Database Encryption SDK for DynamoDB の最新バージョンに移行できます。

- 転送時と保管時のデータは保護されます。以下を含む第三者に公開されることはありません AWS。
- テーブル項目に署名できます。プライマリキー属性など、テーブル項目のすべてまたは一部の署名を計算するように、 AWS Database Encryption SDK for DynamoDB に指示できます。この署名により、属性の追加や削除、属性値のスワップなど、項目全体への不正な変更を検出することができます。
- キーリングを選択することで、データを保護する方法を決定します。キーリングは、データキー、そして最終的にはデータを保護するラッピングキーを決定します。タスクに実用的で、最も安全なラッピングキーを使用してください。
- AWS Database Encryption SDK for DynamoDB はテーブル全体を暗号化しません。項目内でどの属性を暗号化するかを選択します。 AWS Database Encryption SDK for DynamoDB は項目全体を暗号化しません。属性名、プライマリキー (パーティションキーおよびソートキー) 属性の名前または値は暗号化されません。

AWS Encryption SDK

DynamoDB に保存するデータを暗号化する場合は、 AWS Database Encryption SDK for DynamoDB をお勧めします。

AWS Encryption SDK は、クライアント側暗号化ライブラリで、汎用データの暗号化および復号に役立ちます。任意のタイプのデータを保護することはできますが、データベースレコードなどの構造化データは操作できません。 AWS Database Encryption SDK for DynamoDB とは異なり、 は項目レベルの整合性チェックを提供 AWS Encryption SDK できず、属性を認識したり、プライマリキーの暗号化を防ぐロジックはありません。

を使用してテーブルの要素を AWS Encryption SDK 暗号化する場合は、 AWS Database Encryption SDK for DynamoDB と互換性がないことに注意してください。1 つのライブラリで暗号化し、もう 1 つのライブラリを使用して復号することはできません。

どのフィールドが暗号化および署名されますか?

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

AWS Database Encryption SDK for DynamoDB は、特に Amazon DynamoDB アプリケーション用に設計されたクライアント側の暗号化ライブラリです。Amazon DynamoDB は、項目のコレクションであるテーブルにデータを格納します。各項目は、属性の集合です。各属性には名前と値があります。 AWS Database Encryption SDK for DynamoDB は、属性の値を暗号化します。次に、属性に対する署名を計算します。暗号化される属性値、および署名に含めるか指定できます。

暗号化は、属性値の機密保持を保護します。署名は、署名されたすべての属性とその相互の関係を保全し、認証を提供します。これにより、属性の追加や削除、暗号化された値の別の値への置換など、項目全体への不正な変更を検出することができます。

暗号化された項目では、テーブル名、すべての属性名、暗号化していない属性値、プライマリキー (パーティションキーとソートキー) 属性の名前と値、属性タイプなど、一部のデータはプレーンテキストで残ります。これらのフィールドに機密データを保存しないでください。

AWS Database Encryption SDK for DynamoDB の仕組みの詳細については、「」を参照してくださいAWS Database Encryption SDK の仕組み。

> **Note**
>
> AWS Database Encryption SDK for DynamoDB トピックの属性アクションに関するすべての言及は、暗号化アクションを指します。

トピック

- 暗号化の属性値
- 項目の署名

暗号化の属性値

AWS Database Encryption SDK for DynamoDB は、指定した属性の値 (属性名またはタイプではありません) を暗号化します。どの属性値が暗号化されているかを確認するには、属性アクションを使用します。

たとえば、この項目には `example` および `test` 属性が含まれます。

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

`example` 属性を暗号化し、`test` 属性を暗号化しない場合、結果は次のようになります。暗号化された `example` 属性値は、文字列ではなくバイナリデータです。

```
'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'
...
```

各項目のプライマリキー属性 (パーティションキーおよびソートキー) を使用して DynamoDB はテーブル内の項目を検索するため、それらの属性はプレーンテキストのままである必要があります。署名は必要ですが、暗号化の必要はありません。

AWS Database Encryption SDK for DynamoDB は、プライマリキー属性を識別し、その値が署名されているが暗号化されていないことを確認します。また、プライマリキーを特定してそれを暗号化しようとすると、クライアントは例外をスローします。

クライアントは、項目に追加する新しい属性 (aws_dbe_head) にマテリアルの説明を格納します。マテリアルの説明は、項目がどのように暗号化および署名されたかを説明するものです。クライアントは、この情報を使用して項目の検証と復号を行います。マテリアルの説明を格納するフィールドは暗号化されません。

項目の署名

指定された属性値を暗号化した後、 AWS Database Encryption SDK for DynamoDB は、マテリアルの説明、暗号化コンテキスト、および属性アクションSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTで ENCRYPT_AND_SIGN、SIGN_ONLY、または とマークされた各フィールドの正規化にわたって、ハッシュベースのメッセージ認証コード (HMACs) とデジタル署名を計算します。ECDSA 署名はデフォルトで有効になっていますが、必須

ではありません。クライアントは、項目に追加する新しい属性 (aws_dbe_foot) に HMAC と署名を格納します。

DynamoDB での検索可能な暗号化

検索可能な暗号化のために Amazon DynamoDB テーブルを設定するには、AWS KMS 階層キーリングを使用して、項目を保護するために使用されるデータキーを生成、暗号化、および復号する必要があります。また、テーブル暗号化設定に `SearchConfig` を含める必要があります。

> **Note**
>
> DynamoDB 用の Java クライアント側の暗号化ライブラリを使用している場合は、低レベルの AWS Database Encryption SDK for DynamoDB API を使用して、テーブル項目を暗号化、署名、検証、復号化する必要があります。DynamoDB Enhanced Client と下位レベルの `DynamoDBItemEncryptor` は、検索可能な暗号化をサポートしていません。

トピック

- ビーコンを使用したセカンダリインデックスの設定
- ビーコン出力のテスト

ビーコンを使用したセカンダリインデックスの設定

ビーコンを設定した後、暗号化された属性を検索する前に、各ビーコンを反映するセカンダリインデックスを設定する必要があります。

標準ビーコンまたは複合ビーコンを設定すると、 AWS Database Encryption SDK はビーコン名にaws_dbe_b_プレフィックスを追加して、サーバーがビーコンを簡単に識別できるようにします。例えば、複合ビーコンに compoundBeacon という名前を付けた場合、実際の完全なビーコン名は aws_dbe_b_compoundBeacon です。標準ビーコンまたは複合ビーコンを含むセカンダリインデックスを設定する場合は、ビーコン名を識別するときに aws_dbe_b_ プレフィックスを含める必要があります。

パーティションキーとソートキー

プライマリキーの値を暗号化することはできません。パーティションキーとソートキーは署名されている必要があります。プライマリキーの値を標準ビーコンまたは複合ビーコンにすることはできません。

属性を指定しない限りSIGN_ONLY、プライマリキーの値は である必要があります。SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTパーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

プライマリキーの値を署名付きビーコンにすることができます。プライマリキーの値ごとに個別の署名付きビーコンを設定した場合は、プライマリキーの値を識別する属性名を署名付きビーコン名として指定する必要があります。ただし、 AWS Database Encryption SDK は署名付きビーコンにaws_dbe_b_プレフィックスを追加しません。プライマリキーの値に個別の署名付きビーコンを設定した場合でも、必要なのは、セカンダリインデックスを設定する際に、プライマリキーの値の属性名を指定することだけです。

ローカルセカンダリインデックス

ローカルセカンダリインデックスのソートキーはビーコンにすることができます。

ソートキーにビーコンを指定する場合、タイプは String である必要があります。ソートキーに標準ビーコンまたは複合ビーコンを指定する場合は、ビーコン名を指定する際に aws_dbe_b_ プレフィックスを含める必要があります。署名付きビーコンを指定する場合は、プレフィックスなしでビーコン名を指定します。

グローバルセカンダリインデックス

グローバルセカンダリインデックスのパーティションキーとソートキーは両方ともビーコンにすることができます。

パーティションキーまたはソートキーにビーコンを指定する場合、タイプは String である必要があります。ソートキーに標準ビーコンまたは複合ビーコンを指定する場合は、ビーコン名を指定する際に aws_dbe_b_ プレフィックスを含める必要があります。署名付きビーコンを指定する場合は、プレフィックスなしでビーコン名を指定します。

属性の射影

射影とは、テーブルからセカンダリインデックスにコピーされる属性のセットです。テーブルのパーティションキーとソートキーは常にインデックスに射影されます。アプリケーションのクエリ要件をサポートするために、他の属性を射影できます。DynamoDB は、属性プロジェクションのために、KEYS_ONLY、INCLUDE、ALL の 3 つの異なるオプションを提供します。

INCLUDE 属性プロジェクションを使用してビーコンを検索する場合は、ビーコンが構築されるすべての属性の名前と、aws_dbe_b_ プレフィックスを持つビーコン名を指定する必要があります。例えば、field1、field2、および field3 から複合ビーコン compoundBeacon を設定した場合、プロジェクション内で、aws_dbe_b_compoundBeacon、field1、field2、field3 を指定する必要があります。

グローバルセカンダリインデックスはプロジェクションで明示的に指定された属性のみを使用できますが、ローカルセカンダリインデックスは任意の属性を使用できます。

ビーコン出力のテスト

複合ビーコンを設定した場合、または仮想フィールドを使用してビーコンを構築した場合は、DynamoDB テーブルに入力する前に、これらのビーコンが期待される出力を生成することを確認することをお勧めします。

AWS Database Encryption SDK は、仮想フィールドと複合ビーコン出力のトラブルシューティングに役立つ DynamoDbEncryptionTransformsサービスを提供します。

仮想フィールドのテスト

次のスニペットは、テスト項目を作成し、DynamoDB テーブル暗号化設定を使用してDynamoDbEncryptionTransformsサービスを定義し、 ResolveAttributes を使用して仮想フィールドが期待される出力を生成することを確認する方法を示しています。

Java

完全なコードサンプルを参照: VirtualBeaconSearchableEncryptionExample.java

```
// 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

完全なコードサンプル「VirtualBeaconSearchableEncryptionExample.cs」を参照してください。

```
 // 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

完全なコードサンプル「virtual_beacon_searchable_encryption.rs」を参照してください。

```
// 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");
```

複合ビーコンのテスト

次のスニペットでは、テスト項目を作成し、DynamoDB テーブル暗号化設定を使用してDynamoDbEncryptionTransformsサービスを定義し、 ResolveAttributesを使用して複合ビーコンが期待される出力を生成することを確認する方法を示します。

Java

完全なコードサンプルを参照: CompoundBeaconSearchableEncryptionExample.java

```
// 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

完全なコードサンプルを参照: CompoundBeaconSearchableEncryptionExample.cs

```
// 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

完全なコードサンプルを参照: 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
```

データモデルの更新

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

Database AWS Encryption SDK for DynamoDB を設定するときは、属性アクションを指定します。暗号化では、 AWS データベース暗号化 SDK は 属性アクションを使用して、暗号化して署名する属性、署名する (暗号化しない) 属性、無視する属性を識別します。また、許可された署名なし属性を定義して、どの属性が署名から除外されるかをクライアントに明示的に伝えます。復号時に、 AWS Database Encryption SDK は、定義した許可された署名なし属性を使用して、署名に含まれていない属性を識別します。属性アクションは暗号化された項目に保存されず、 AWS Database Encryption SDK は属性アクションを自動的に更新しません。

属性アクションを慎重に選択します。不確かな場合は、暗号化と署名を使用します。 AWS Database Encryption SDK を使用して項目を保護すると、既存の ENCRYPT_AND_SIGN、SIGN_ONLY、または SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を に変更することはできませんDO_NOTHING。ただし、次の変更は安全に行うことができます。

- 新しい ENCRYPT_AND_SIGN、SIGN_ONLY、および SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を追加する
- 既存の属性を削除する
- 既存のENCRYPT_AND_SIGN属性を SIGN_ONLYまたは に変更する SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT
- 既存の SIGN_ONLYまたは SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を に変更する ENCRYPT_AND_SIGN
- 新しい DO_NOTHING 属性を追加する
- 既存の SIGN_ONLY 属性を SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT に変更する
- 既存の SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を SIGN_ONLY に変更する

検索可能な暗号化に関する考慮事項

データモデルを更新する前に、属性から構築したビーコンに対して、その更新がどのような影響を及ぼす可能性があるかを慎重に検討してください。ビーコンを持つ新しいレコードを書き込んだ後に、そのビーコンの設定を更新することはできません。ビーコンを構築するために使用した属性に関連付けられた属性アクションを更新することはできません。既存の属性とそれに関連付けられたビーコンを削除すると、そのビーコンを使用して既存のレコードをクエリできなくなります。レコードに追加する新しいフィールドについての新しいビーコンを作成することはできますが、既存のビーコンを更新して新しいフィールドを含めることはできません。

SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性に関する考慮事項

デフォルトでは、パーティションキーとソートキーは、暗号化コンテキストに含まれる唯一の属性です。AWS KMS 階層キーリングのブランチキー ID サプライヤーが暗号化コンテキストからの復号に必要なブランチキーを特定SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTできるように、追加のフィールドを として定義することを検討してください。詳細については、「ブランチキー ID サプライヤー」を参照してください。SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を指定する場合、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 暗号化アクションを使用するには、バージョン 3.3 以降の AWS Database Encryption SDK を使用する必要があります。データモデルを更新して を含める前に、すべてのリーダーに新しいバージョンをデプロイしますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

新しい ENCRYPT_AND_SIGN、SIGN_ONLY、および SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を追加する

新しい ENCRYPT_AND_SIGN、SIGN_ONLY、または SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を追加するには、属性アクションで新しい属性を定義します。

既存のDO_NOTHING属性を削除して、ENCRYPT_AND_SIGN、SIGN_ONLY、または SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性として再度追加することはできません。

アノテーション付きデータクラスの使用

TableSchema を使用して属性アクションを定義した場合は、新しい属性をアノテーション付きデータクラスに追加します。新しい属性の属性アクションのアノテーションを指定しない場合、クライアントは、デフォルトで新しい属性を暗号化して署名します (属性がプライマリキーの一部である場合を除きます)。新しい属性のみに署名する場合は、 @DynamoDBEncryptionSignOnlyまたは @DynamoDBEncryptionSignAndIncludeInEncryptionContext注釈で新しい属性を追加する必要があります。

オブジェクトモデルの使用

属性アクションを手動で定義した場合は、オブジェクトモデルの属性アクションに新しい属性を追加し、属性アクションSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTとして ENCRYPT_AND_SIGN、SIGN_ONLY、または を指定します。

既存の属性を削除する

属性が必要なくなったと判断した場合は、その属性に対するデータの書き込みを停止することも、属性アクションから正式に削除することもできます。属性に対する新しいデータの書き込みを停止しても、その属性は引き続き属性アクションに表示されます。これは、将来再び属性の使用を開始する必要がある場合に役立ちます。属性アクションから属性を正式に削除しても、データセットからは削除されません。データセットには、その属性を含む項目が引き続き含まれます。

既存の ENCRYPT_AND_SIGN、SIGN_ONLY、SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、または DO_NOTHING 属性を正式に削除するには、属性アクションを更新します。

DO_NOTHING 属性を削除する場合でも、許可された署名なし属性からその属性を削除しないでください。その属性に対して新しい値を書き込まなくなった場合でも、クライアントは、その属性を含む既存の項目を読み取るために、その属性が署名されていないことを認識する必要があります。

アノテーション付きデータクラスの使用

TableSchema を使用して属性アクションを定義した場合は、アノテーション付きデータクラスからその属性を削除します。

オブジェクトモデルの使用

属性アクションを手動で定義した場合は、オブジェクトモデルの属性アクションから属性を削除します。

既存のENCRYPT_AND_SIGN属性を SIGN_ONLYまたは に変更する SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

既存のENCRYPT_AND_SIGN属性を SIGN_ONLYまたは に変更するには SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、属性アクションを更新する必要があります。更新をデプロイした後、クライアントは属性に書き込まれた既存の値を検証して復号できるようになりますが、実行するアクションは属性に対して書き込まれた新しい値に署名することだけです。

> **Note**
>
> 既存のENCRYPT_AND_SIGN属性を SIGN_ONLYまたは に変更する前に、セキュリティ要件を慎重に検討してくださいSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。機密データを保存できる属性はすべて暗号化する必要があります。

アノテーション付きデータクラスの使用

で属性アクションを定義した場合はTableSchema、既存の属性を更新して、注釈付きデータクラスに @DynamoDBEncryptionSignOnlyまたは @DynamoDBEncryptionSignAndIncludeInEncryptionContext注釈を含めます。

オブジェクトモデルの使用

属性アクションを手動で定義した場合は、既存の属性に関連付けられた属性アクションを オブジェクトモデルSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTで から SIGN_ONLYまたは ENCRYPT_AND_SIGN に更新します。

既存の SIGN_ONLYまたは SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を に変更する ENCRYPT_AND_SIGN

既存の SIGN_ONLYまたは SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を に変更するにはENCRYPT_AND_SIGN、属性アクションを更新する必要があります。更新をデプロイした後、クライアントは属性に書き込まれた既存の値を検証できるようになり、属性に対して書き込まれた新しい値を暗号化して署名します。

アノテーション付きデータクラスの使用

で属性アクションを定義した場合はTableSchema、既存の属性から
@DynamoDBEncryptionSignOnlyまたは
@DynamoDBEncryptionSignAndIncludeInEncryptionContext注釈を削除します。

オブジェクトモデルの使用

属性アクションを手動で定義した場合は、オブジェクトモデルの 属性に関連付けられた属性アクションを SIGN_ONLYまたは から SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT ENCRYPT_AND_SIGNに更新します。

新しい DO_NOTHING 属性を追加する

新しい DO_NOTHING 属性を追加する際のエラーのリスクを軽減するには、DO_NOTHING 属性に名前を付ける際に個別のプレフィックスを指定し、そのプレフィックスを使用して許可された署名なし属性を定義することをお勧めします。

注釈付きデータクラスから既存の ENCRYPT_AND_SIGN、SIGN_ONLY、または SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を削除し、属性をDO_NOTHING属性として再度追加することはできません。まったく新しい DO_NOTHING 属性のみを追加できます。

新しい DO_NOTHING 属性を追加するために実行するステップは、許可された署名なし属性をリスト内で明示的に定義したか、またはプレフィックスを使用して定義したかによって異なります。

許可された署名なし属性プレフィックスの使用

TableSchema を使用して属性アクションを定義した場合は、@DynamoDBEncryptionDoNothing アノテーションを使用して新しい DO_NOTHING 属性をアノテーション付きデータクラスに追加します。属性アクションを手動で定義した場合は、新しい属性を含むように属性アクションを更新します。必ず DO_NOTHING 属性アクションを使用して新しい属性を明示的に設定してください。新しい属性の名前には、同じ個別のプレフィックスを含める必要があります。

許可された署名なし属性リストの使用

1. 許可された署名なし属性リストに新しい DO_NOTHING 属性を追加し、更新されたリストをデプロイします。
2. ステップ 1 の変更をデプロイします。

   このデータを読み取る必要があるすべてのホストに変更が反映されるまで、ステップ 3 に進むことはできません。
3. 新しい DO_NOTHING 属性を属性アクションに追加します。

a. TableSchema を使用して属性アクションを定義した場合は、@DynamoDBEncryptionDoNothing アノテーションを使用して新しい DO_NOTHING 属性をアノテーション付きデータクラスに追加します。

b. 属性アクションを手動で定義した場合は、新しい属性を含むように属性アクションを更新します。必ず DO_NOTHING 属性アクションを使用して新しい属性を明示的に設定してください。

4. ステップ 3 の変更をデプロイします。

既存の SIGN_ONLY 属性を SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT に変更する

既存の SIGN_ONLY 属性を SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT に変更するには、属性アクションを更新する必要があります。更新をデプロイすると、クライアントは 属性に書き込まれた既存の値を検証でき、 属性に書き込まれた新しい値に署名し続けます。属性に書き込まれた新しい値は、暗号化コンテキストに含まれます。

SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を指定する場合、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

アノテーション付きデータクラスの使用

で属性アクションを定義した場合はTableSchema、属性に関連付けられた属性アクションを から @DynamoDBEncryptionSignOnlyに更新します@DynamoDBEncryptionSignAndIncludeInEncryptionContext。

オブジェクトモデルの使用

属性アクションを手動で定義した場合は、オブジェクトモデル内で属性に関連付けられた属性アクションを SIGN_ONLY から SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT に更新します。

既存の SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を SIGN_ONLY に変更する

既存の SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を SIGN_ONLY に変更するには、属性アクションを更新する必要があります。更新をデプロイすると、クライアントは 属性に書き込まれた既存の値を検証でき、 属性に書き込まれた新しい値に署名し続けます。属性に書き込まれた新しい値は、暗号化コンテキストに含まれません。

既存のSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性を に変更する前にSIGN_ONLY、更新がブランチキー ID サプライヤの機能にどのように影響するかを慎重に検討してください。

アノテーション付きデータクラスの使用

で属性アクションを定義した場合はTableSchema、属性に関連付けられた属性アクションを から @DynamoDBEncryptionSignAndIncludeInEncryptionContextに更新します@DynamoDBEncryptionSignOnly。

オブジェクトモデルの使用

属性アクションを手動で定義した場合は、オブジェクトモデル内で属性に関連付けられた属性アクションを SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT から SIGN_ONLY に更新します。

AWS Database Encryption SDK for DynamoDB で利用可能なプログラミング言語

AWS Database Encryption SDK for DynamoDB は、次のプログラミング言語で使用できます。言語固有のライブラリはさまざまですが、結果として得られる実装は相互運用ができます。ある言語実装で暗号化し、別の言語実装で復号できます。相互運用性は、言語の制約を受ける可能性があります。その場合の制約については、言語実装に関するトピックで説明します。

トピック

- Java
- .NET
- Rust

Java

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

このトピックでは、DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x をインストールして使用する方法について説明します。 AWS Database Encryption SDK for DynamoDB

を使用したプログラミングの詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの Java の例を参照してください。

> **Note**
>
> 次のトピックでは、DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x に焦点を当てます。
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。 AWS Database Encryption SDK は、引き続きレガシー DynamoDB 暗号化クライアントバージョンをサポートします。

トピック

前提条件

DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x をインストールする前に、次の前提条件を満たしていることを確認してください。

Java 開発環境

Java 8 以降が必要になります。Oracle のウェブサイトで Java SE のダウンロードに移動し、Java SE Development Kit (JDK) をダウンロードして、インストールします。

Oracle JDK を使用する場合は、Java Cryptography Extension (JCE) 無制限強度の管轄ポリシーファイルをダウンロードして、インストールする必要があります。

AWS SDK for Java 2.x

AWS Database Encryption SDK for DynamoDB には、 の DynamoDB 拡張クライアントモジュールが必要です AWS SDK for Java 2.x。SDK 全体またはこのモジュールだけをインストールできます。

のバージョンの更新については AWS SDK for Java、「 のバージョン 1.x から 2.x への移行 AWS SDK for Java」を参照してください。

AWS SDK for Java は Apache Maven から入手できます。全体の依存関係を宣言することも AWS SDK for Java、dynamodb-enhancedモジュールのみの依存関係を宣言することもできます。

Apache Maven AWS SDK for Java を使用して をインストールする

- 依存関係として AWS SDK for Java全体をインポートするには、pom.xml ファイルでそれを宣言します。
- AWS SDK for Javaで Amazon DynamoDB モジュールのみの依存関係を作成するには、特定のモジュールを指定する手順に従います。groupId を software.amazon.awssdk に、artifactID を dynamodb-enhanced に設定します。

> **Note**
>
> AWS KMS キーリングまたは AWS KMS 階層キーリングを使用する場合は、AWS KMS モジュールの依存関係も作成する必要があります。groupId を software.amazon.awssdk に、artifactID を kms に設定します。

インストール

DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x は、次の方法でインストールできます。

Apache Maven の使用

Amazon DynamoDB Encryption Client for Java は、以下の依存定義を使用して、Apache Maven を介して利用できます。

```
<dependency>
  <groupId>software.amazon.cryptography</groupId>
```

```
  <artifactId>aws-database-encryption-sdk-dynamodb</artifactId>
  <version>version-number</version>
</dependency>
```

Gradle Kotlin の使用

Gradle プロジェクトの依存関係セクションに次を追加することで、Gradle を使用して Amazon DynamoDB Encryption Client for Java に対する依存関係を宣言できます。

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

手動

DynamoDB 用の Java クライアント側の暗号化ライブラリをインストールするには、aws-database-encryption-sdk-dynamodb GitHub リポジトリのクローンを作成するか、ダウンロードします。

SDK をインストールしたら、このガイドのコード例と GitHub の aws-database-encryption-sdk-dynamodb リポジトリの Java の例を確認して開始します。

DynamoDB 用の Java クライアント側の暗号化ライブラリの使用

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

このトピックでは、DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x の関数とヘルパークラスの一部について説明します。

DynamoDB 用の Java クライアント側の暗号化ライブラリを使用したプログラミングの詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリにある Java の例、Java の例を参照してください。

トピック

- 項目エンクリプタ
- AWS Database Encryption SDK for DynamoDB の属性アクション

- AWS Database Encryption SDK for DynamoDB の暗号化設定
- AWS Database Encryption SDK を使用した項目の更新
- 署名付きセットの復号化

項目エンクリプタ

その中核となる AWS Database Encryption SDK for DynamoDB は項目エンクリプタです。DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x を使用して、次の方法で DynamoDB テーブル項目の暗号化、署名、検証、および復号を行うことができます。

DynamoDB Enhanced Client

DynamoDB `PutItem` リクエストを使用してクライアント側で項目を自動的に暗号化して署名するように、`DynamoDbEncryptionInterceptor` で DynamoDB Enhanced Client を設定できます。DynamoDB Enhanced Client を使用すると、アノテーション付きデータクラスを使用して属性アクションを定義できます。可能な場合は常に、DynamoDB Enhanced Client を使用することをお勧めします。

DynamoDB Enhanced Client は、検索可能な暗号化をサポートしていません。

> **Note**
>
> AWS Database Encryption SDK は、ネストされた属性の注釈をサポートしていません。

下位レベルの DynamoDB API

DynamoDB `PutItem` リクエストを使用してクライアント側で項目を自動的に暗号化して署名するように、`DynamoDbEncryptionInterceptor` で下位レベルの DynamoDB API を設定できます。

検索可能な暗号化を使用するには、下位レベルの DynamoDB API を使用する必要があります。

下位レベルの `DynamoDbItemEncryptor`

下位レベルの `DynamoDbItemEncryptor` は、DynamoDB を呼び出すことなく、テーブル項目を直接暗号化して署名するか、または復号して検証します。DynamoDB の `PutItem` または `GetItem` リクエストは実行しません。例えば、下位レベルの `DynamoDbItemEncryptor` を使用して、既に取得した DynamoDB 項目を直接復号して検証できます。

下位レベルの DynamoDbItemEncryptor は、検索可能な暗号化をサポートしていません。

AWS Database Encryption SDK for DynamoDB の属性アクション

属性アクションは、暗号化および署名される属性値を決定します。この属性値は署名のみされ、署名されて暗号化コンテキストに含まれ、無視されます。

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 暗号化アクションを使用するには、AWS Database Encryption SDK のバージョン 3.3 以降を使用する必要があります。データモデルを更新して を含める前に、すべてのリーダーに新しいバージョンをデプロイしますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

下位レベルの DynamoDB API または下位レベルの DynamoDbItemEncryptor を使用する場合は、属性アクションを手動で定義する必要があります。DynamoDB Enhanced Client を使用する場合は、属性アクションを手動で定義するか、またはアノテーション付きデータクラスを使用して、TableSchema を生成できます。設定プロセスを簡素化するには、アノテーション付きデータクラスを使用することをお勧めします。アノテーション付きデータクラスを使用する場合、オブジェクトを 1 回だけモデル化する必要があります。

> **Note**
>
> 属性アクションを定義した後、どの属性を署名から除外するかを定義する必要があります。将来、新しい署名なし属性を簡単に追加できるように、署名なし属性を識別するための個別のプレフィックス (「:」など) を選択することをお勧めします。DynamoDB スキーマと属性アクションを定義するときに DO_NOTHING とマークされたすべての属性の属性名にこのプレフィックスを含めます。

アノテーション付きデータクラスを使用する

アノテーション付きデータクラスを使用して、DynamoDB Enhanced Client および DynamoDbEncryptionInterceptor で属性アクションを指定します。 AWS Database Encryption SDK for DynamoDB は、標準の DynamoDB 属性の注釈を使用して、属性を保護する方法を決定する属性のタイプを定義します。デフォルトでは、プライマリキーを除く属性がすべて暗号化されます。これらの属性は署名されますが、暗号化はされません。

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 暗号化アクションを使用するには、AWS Database Encryption SDK のバージョン 3.3 以降を使用する必要があります。データモデルを更新して を含める前に、すべてのリーダーに新しいバージョンをデプロイしますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

DynamoDB 拡張クライアント注釈の詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの SimpleClass.java を参照してください。

デフォルトでは、プライマリキー属性は署名されてはいるが、暗号化されておらず (SIGN_ONLY)、他のすべての属性は暗号化されて署名されています (ENCRYPT_AND_SIGN)。属性を として定義する場合SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。例外を指定するには、DynamoDB 用の Java クライアント側の暗号化ライブラリで定義されている暗号化アノテーションを使用します。例えば、特定の属性を署名のみにしたい場合は、@DynamoDbEncryptionSignOnly アノテーションを使用します。特定の属性に署名して暗号化コンテキストに含める場合は、 を使用しますす@DynamoDbEncryptionSignAndIncludeInEncryptionContext。特定の属性が署名も暗号化もされないようにしたい場合 (DO_NOTHING) は、@DynamoDbEncryptionDoNothing アノテーションを使用します。

> **Note**
>
> AWS Database Encryption SDK は、ネストされた属性の注釈をサポートしていません。

次の例は、ENCRYPT_AND_SIGN、、SIGN_ONLYおよび DO_NOTHING属性アクションの定義に使用される注釈を示しています。の定義に使用される注釈の例についてはSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、「SimpleClass4.java」を参照してください。

```
@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;
    }

}
```

次のスニペットに示すように、アノテーション付きデータクラスを使用して TableSchema を作成します。

```
final TableSchema<SimpleClass> tableSchema = TableSchema.fromBean(SimpleClass.class);
```

属性アクションを手動で定義する

属性アクションを手動で指定するには、名前と値のペアが属性名と指定されたアクションを表す Map オブジェクトを作成します。

属性を暗号化して署名するように ENCRYPT_AND_SIGN を指定します。属性に署名するが暗号化はしないように SIGN_ONLY を指定します。SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT を指定して属性に署名し、暗号化コンテキストに含めます。属性に署名することなく、その属性を暗号化することはできません。属性を無視するように DO_NOTHING を指定します。

パーティション属性とソート属性は、 SIGN_ONLY または のいずれかである必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。属性を として定義する場合SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 暗号化アクションを使用するには、AWS Database Encryption SDK のバージョン 3.3 以降を使用する必要があります。データモデルを更新して を含める前に、すべてのリーダーに新しいバージョンをデプロイしますSIGN_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 DynamoDB の暗号化設定

AWS Database Encryption SDK を使用する場合は、DynamoDB テーブルの暗号化設定を明示的に定義する必要があります。暗号化設定に必要な値は、属性アクションを手動で定義したか、またはアノテーション付きデータクラスを使用して定義したかによって異なります。

次のスニペットは、DynamoDB Enhanced Client、`TableSchema`、および個別のプレフィックスによって定義された、許可された署名なし属性を使用して、DynamoDB テーブルの暗号化設定を定義します。

```
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());
```

論理テーブル名

DynamoDB テーブルの論理テーブル名。

論理テーブル名は、DynamoDB の復元オペレーションを簡素化するために、テーブルに格納されているすべてのデータに暗号的にバインドされます。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。常に同じ論理テーブル名を指定する必要があります。復号を成功させるには、論理テーブル名が、暗号

化の際に指定された名前と一致する必要があります。DynamoDB テーブルをバックアップから復元した後に DynamoDB テーブル名が変更された場合でも、論理テーブル名を使用することで、復号オペレーションで引き続きテーブルが確実に認識されます。

許可された署名なし属性

属性アクションで DO_NOTHING とマークされた属性。

許可された署名なし属性は、どの属性が署名から除外されるかをクライアントに伝えます。クライアントは、他のすべての属性が署名に含まれていると想定します。その後、レコードを復号する際に、クライアントは、ユーザーが指定する、許可された署名なし属性の中からどの属性を検証する必要があり、どの属性を無視する必要があるかを決定します。許可された署名なし属性から属性を削除することはできません。

すべての DO_NOTHING 属性をリストする配列を作成することで、許可された署名なし属性を明示的に定義できます。また、DO_NOTHING 属性に名前を付ける際に個別のプレフィックスを指定し、そのプレフィックスを使用してどの属性が署名されていないかをクライアントに伝えることもできます。将来新しい DO_NOTHING 属性を追加するプロセスが簡素化されるため、個別のプレフィックスを指定することを強くお勧めします。詳細については、「データモデルの更新」を参照してください。

すべての DO_NOTHING 属性のためにプレフィックスを指定しない場合は、クライアントが復号時に署名されていないことを想定するすべての属性を明示的にリストする allowedUnsignedAttributes 配列を設定できます。どうしても必要な場合にのみ、許可された署名なし属性を明示的に定義する必要があります。

検索設定 (オプション)

SearchConfig はビーコンのバージョンを定義します。

検索可能な暗号化または署名付きビーコンを使用するには、SearchConfig を指定する必要があります。

アルゴリズムスイート (オプション)

algorithmSuiteId は、 AWS Database Encryption SDK が使用するアルゴリズムスイートを定義します。

代替アルゴリズムスイートを明示的に指定しない限り、 AWS Database Encryption SDK はデフォルトのアルゴリズムスイートを使用します。デフォルトのアルゴリズムスイートは、キーの導出、デジタル署名、およびキーコミットメントを備えた AES-GCM アルゴリズムを使用します。デフォルトのアルゴリズムスイートはほとんどのアプリケーションに適している可能性

がありますが、代替アルゴリズムスイートを選択できます。例えば、一部の信頼モデルは、デジタル署名を含まないアルゴリズムスイートによって満たされます。 AWS Database Encryption SDK がサポートするアルゴリズムスイートの詳細については、「」を参照してくださいAWS Database Encryption SDK でサポートされているアルゴリズムスイート。

ECDSA デジタル署名のない AES-GCM アルゴリズムスイートを選択するには、テーブル暗号化設定に次のスニペットを含めます。

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

AWS Database Encryption SDK を使用した項目の更新

AWS Database Encryption SDK は、暗号化または署名された項目に対して ddb:UpdateItem をサポートしていません。暗号化または署名された項目を更新するには、ddb:PutItem を使用する必要があります。PutItem リクエストで既存の項目と同じプライマリキーを指定すると、新しい項目が既存の項目に完全に置き換わります。CLOBBER を使用して、項目を更新した後、保存する際にすべての属性をクリアして置き換えることもできます。

署名付きセットの復号化

AWS Database Encryption SDK のバージョン 3.0.0 および 3.1.0 では、セットタイプ属性を として定義した場合SIGN_ONLY、セットの値は指定された順序で正規化されます。DynamoDB はセットの順序を保持しません。その結果、セットを含む項目の署名の検証が失敗する可能性があります。セットの値が AWS Database Encryption SDK に提供された順序とは異なる順序で返されると、セット属性に同じ値が含まれている場合でも、署名の検証は失敗します。

> **Note**
>
> AWS Database Encryption SDK のバージョン 3.1.1 以降では、すべてのセットタイプ属性の値が正規化されるため、DynamoDB に書き込まれたのと同じ順序で値が読み取られます。

署名の検証が失敗すると、復号オペレーションは失敗して以下のエラーメッセージを返します。

```
software.amazon.cryptography.dbencryptionsdk.structuredencryption.model.StructuredEncryptionException: 受信者タグが一致しませんでした。
```

上記のエラーメッセージが表示され、復号化しようとしているアイテムにバージョン 3.0.0 または 3.1.0 を使用して署名されたセットが含まれていると思われる場合、セットを正常に検証する方法の詳細として GitHub の aws-database-encryption-sdk-dynamodb-java リポジトリの DecryptWithPermute ディレクトリを参照してください。

Java の例

> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

次の例は、DynamoDB 用の Java クライアント側の暗号化ライブラリを使用して、アプリケーション内のテーブル項目を保護する方法を示しています。その他の例については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリにある Java の例を参照してください (また、独自の例を提供してください)。

次の例は、データが入力されていない新しい Amazon DynamoDB テーブルで DynamoDB 用の Java クライアント側の暗号化ライブラリを設定する方法を示しています。既存の Amazon DynamoDB テーブルをクライアント側の暗号化のために設定する場合は、「既存のテーブルにバージョン 3.x を追加する」を参照してください。

トピック

- DynamoDB 拡張クライアントの使用
- 下位レベルの DynamoDB API の使用
- 下位レベルの DynamoDbItemEncryptor の使用

DynamoDB 拡張クライアントの使用

次の例は、AWS KMS キーリングで DynamoDB Enhanced Client と `DynamoDbEncryptionInterceptor` を使用して、DynamoDB API 呼び出しの一部として DynamoDB テーブルの項目を暗号化する方法を示しています。

DynamoDB 拡張クライアントではサポートされている任意のキーリングを使用できますが、可能な限りいずれかの AWS KMS キーリングを使用することをお勧めします。

> **Note**
>
> DynamoDB Enhanced Client は、検索可能な暗号化をサポートしていません。検索可能な暗号化を使用するには、下位レベルの DynamoDB API とともに `DynamoDbEncryptionInterceptor` を使用します。

完全なコードサンプルを参照してください: EnhancedPutGetExample.java

ステップ 1: AWS KMS キーリングを作成する

次の例では`CreateAwsKmsMrkMultiKeyring`、 を使用して、対称暗号化 KMS AWS KMS キーを使用して キーリングを作成します。`CreateAwsKmsMrkMultiKeyring` メソッドにより、キーリングは、単一リージョンのキーとマルチリージョンのキーの両方を確実に正しく処理します。

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

ステップ 2: アノテーション付きデータクラスからテーブルスキーマを作成する

次の例では、アノテーション付きデータクラスを使用して、`TableSchema` を作成します。

この例では、アノテーション付きのデータクラスと属性アクションが SimpleClass.java を使用して定義されていることを前提としています。属性アクションにアノテーションを付ける方法のガイダンスについては、「アノテーション付きデータクラスを使用する」を参照してください。

> **Note**
>
> AWS Database Encryption SDK は、ネストされた属性の注釈をサポートしていません。

```
final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);
```

ステップ 3: 署名から除外する属性を定義する

次の例では、すべての DO_NOTHING 属性が個別のプレフィックス「:」を共有し、そのプレフィックスを使用して、許可される署名なし属性を定義すると想定しています。クライアントは、「:」というプレフィックスが付いた属性名は署名から除外されると想定します。詳細については、「Allowed unsigned attributes」を参照してください。

```
final String unsignedAttrPrefix = ":";
```

ステップ 4: 暗号化設定を作成する

次の例では、DynamoDB テーブルの暗号化設定を表す tableConfigs マップを定義します。

この例では、DynamoDB テーブル名を論理テーブル名として指定します。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。詳細については、「AWS Database Encryption SDK for DynamoDB の暗号化設定」を参照してください。

> **Note**
>
> 検索可能な暗号化または署名付きビーコンを使用するには、暗号化設定に SearchConfig も含める必要があります。

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
    DynamoDbEnhancedTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .schemaOnEncrypt(tableSchema)
        .build());
```

ステップ 5: **DynamoDbEncryptionInterceptor** を作成する

次の例では、ステップ 4 の tableConfigs を使用して新しい DynamoDbEncryptionInterceptor を作成します。

```
final DynamoDbEncryptionInterceptor interceptor =
```

```
    DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
        CreateDynamoDbEncryptionInterceptorInput.builder()
            .tableEncryptionConfigs(tableConfigs)
            .build()
    );
```

ステップ 6: 新しい AWS SDK DynamoDB クライアントを作成する

次の例では、ステップ 5 interceptorの を使用して新しい AWS SDK DynamoDB クライアントを作成します。

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                       .addExecutionInterceptor(interceptor)
                       .build())
        .build();
```

ステップ 7: DynamoDB Enhanced Client を作成し、テーブルを作成する

次の例では、ステップ 6 で作成した AWS SDK DynamoDB クライアントを使用して DynamoDB Enhanced Client を作成し、アノテーション付きデータクラスを使用してテーブルを作成します。

```
final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
        .build();
final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
```

ステップ 8: テーブル項目を暗号化して署名する

次の例では、DynamoDB Enhanced Client を使用して項目を DynamoDB テーブルに配置します。項目は、DynamoDB に送信される前に、クライアント側で暗号化および署名されます。

```
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);
```

下位レベルの DynamoDB API の使用

次の例は、AWS KMS キーリングとともに下位レベルの DynamoDB API を使用し、DynamoDB PutItem リクエストを使用してクライアント側で項目を自動的に暗号化して署名する方法を示しています。

サポートされている任意のキーリングを使用できますが、可能な限りいずれかの AWS KMS キーリングを使用することをお勧めします。

完全なコードサンプルを参照してください: BasicPutGetExample.java

ステップ 1: AWS KMS キーリングを作成する

次の例ではCreateAwsKmsMrkMultiKeyring、 を使用して、対称暗号化 KMS AWS KMS キーを使用して キーリングを作成します。CreateAwsKmsMrkMultiKeyring メソッドにより、キーリングは、単一リージョンのキーとマルチリージョンのキーの両方を確実に正しく処理します。

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

ステップ 2: 属性アクションを設定する

次の例では、テーブル項目のサンプル属性アクションを表す attributeActionsOnEncrypt マップを定義します。

> **Note**
>
> 次の例では、属性を として定義していませんSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。SIGN_AND_INCLUDE_IN_ENCRYPTION_C属性を指定する場合、パーティション属性とソート属性も である必要があります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);
```

ステップ 3: 署名から除外する属性を定義する

次の例では、すべての DO_NOTHING 属性が個別のプレフィックス「:」を共有し、そのプレフィックスを使用して、許可される署名なし属性を定義すると想定しています。クライアントは、「:」というプレフィックスが付いた属性名は署名から除外されると想定します。詳細については、「Allowed unsigned attributes」を参照してください。

```
final String unsignedAttrPrefix = ":";
```

ステップ 4: DynamoDB テーブルの暗号化設定を定義する

次の例では、この DynamoDB テーブルの暗号化設定を表す `tableConfigs` マップを定義します。

この例では、DynamoDB テーブル名を論理テーブル名として指定します。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。詳細については、「AWS Database Encryption SDK for DynamoDB の暗号化設定」を参照してください。

> **Note**
>
> 検索可能な暗号化または署名付きビーコンを使用するには、暗号化設定に `SearchConfig` も含める必要があります。

```
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);
```

ステップ 5: **DynamoDbEncryptionInterceptor** を作成する

次の例では、ステップ 4 の tableConfigs を使用して DynamoDbEncryptionInterceptor を作成します。

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

ステップ 6: 新しい AWS SDK DynamoDB クライアントを作成する

次の例では、ステップ 5 interceptorの を使用して新しい AWS SDK DynamoDB クライアントを作成します。

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

ステップ 7: DynamoDB テーブル項目を暗号化して署名する

次の例では、サンプルテーブル項目を表す item マップを定義し、その項目を DynamoDB テーブルに配置します。項目は、DynamoDB に送信される前に、クライアント側で暗号化および署名されます。

```
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);
```

下位レベルの DynamoDbItemEncryptor の使用

次の例は、下位レベルの DynamoDbItemEncryptor を AWS KMS キーリングとともに使用して、テーブル項目を直接暗号化して署名する方法を示しています。DynamoDbItemEncryptor は項目を DynamoDB テーブルに配置しません。

DynamoDB 拡張クライアントではサポートされている任意のキーリングを使用できますが、可能な限りいずれかの AWS KMS キーリングを使用することをお勧めします。

> **Note**
>
> 下位レベルの DynamoDbItemEncryptor は、検索可能な暗号化をサポートしていません。検索可能な暗号化を使用するには、下位レベルの DynamoDB API とともに DynamoDbEncryptionInterceptor を使用します。

完全なコードサンプルを参照してください: ItemEncryptDecryptExample.java

ステップ 1: AWS KMS キーリングを作成する

次の例ではCreateAwsKmsMrkMultiKeyring、 を使用して、対称暗号化 KMS AWS KMS キーを使用して キーリングを作成します。CreateAwsKmsMrkMultiKeyring メソッドにより、キーリングは、単一リージョンのキーとマルチリージョンのキーの両方を確実に正しく処理します。

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

ステップ 2: 属性アクションを設定する

次の例では、テーブル項目のサンプル属性アクションを表す attributeActionsOnEncrypt マップを定義します。

> **Note**
>
> 次の例では、属性を として定義していませんSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。SIGN_AND_INCLUDE_IN_ENCRYPTION_C 属性を指定する場合、パーティション属性とソート属性も である必要があります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);
```

ステップ 3: 署名から除外する属性を定義する

次の例では、すべての DO_NOTHING 属性が個別のプレフィックス「:」を共有し、そのプレフィックスを使用して、許可される署名なし属性を定義すると想定しています。クライアントは、「:」というプレフィックスが付いた属性名は署名から除外されると想定します。詳細については、「Allowed unsigned attributes」を参照してください。

```
final String unsignedAttrPrefix = ":";
```

ステップ 4: **DynamoDbItemEncryptor** 設定を定義する

次の例では、DynamoDbItemEncryptor の設定を定義します。

この例では、DynamoDB テーブル名を論理テーブル名として指定します。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。詳細については、「AWS Database Encryption SDK for DynamoDB の暗号化設定」を参照してください。

```
final DynamoDbItemEncryptorConfig config = DynamoDbItemEncryptorConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

ステップ 5: **DynamoDbItemEncryptor** を作成する

次の例では、ステップ 4 の config を使用して新しい DynamoDbItemEncryptor を作成します。

```
final DynamoDbItemEncryptor itemEncryptor = DynamoDbItemEncryptor.builder()
        .DynamoDbItemEncryptorConfig(config)
        .build();
```

ステップ 6: テーブル項目を直接暗号化して署名する

次の例では、DynamoDbItemEncryptor を使用して項目を直接暗号化し、署名します。DynamoDbItemEncryptor は項目を DynamoDB テーブルに配置しません。

```
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();
```

AWS Database Encryption SDK for DynamoDB を使用するように既存の DynamoDB テーブルを設定する

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x を使用すると、既存の Amazon DynamoDB テーブルをクライアント側の暗号化用に設定できます。このトピックでは、データが入力されている既存の DynamoDB テーブルにバージョン 3.x を追加するために必要な 3 つのステップについてのガイダンスを提供します。

前提条件

DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x では、 AWS SDK for Java 2.x で提供される DynamoDB Enhanced Client が必要です。DynamoDBMapper を引き続き使用する場合は、DynamoDB 拡張クライアント AWS SDK for Java 2.x を使用するには に移行する必要があります。

AWS SDK for Javaのバージョン 1.x から 2.x に移行する手順に従います。

その後、DynamoDB Enhanced Client API の使用を開始するための手順に従います。

DynamoDB 用の Java クライアント側の暗号化ライブラリを使用するようにテーブルを設定する前に、アノテーション付きデータクラスを使用して `TableSchema` を生成し、拡張クライアントを作成する必要があります。

ステップ 1: 暗号化された項目の読み取りと書き込みの準備をする

Database Encryption SDK AWS クライアントが暗号化された項目を読み書きできるように準備するには、次の手順を実行します。次の変更をデプロイした後も、クライアントは引き続きプレーンテキスト項目の読み取りと書き込みを行います。テーブルに書き込まれる新しい項目の暗号化や署名は行いませんが、暗号化された項目が表示されるとすぐに復号できます。これらの変更により、クライアントが新しい項目の暗号化を開始するための準備が整います。次のステップに進む前に、次の変更を各リーダーにデプロイする必要があります。

1. 属性アクションを定義する

   アノテーション付きデータクラスを更新して、どの属性値を暗号化して署名するか、どの属性値を署名のみにするか、どの属性値を無視するかを定義する属性アクションを含めます。

   DynamoDB 拡張クライアント注釈の詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの SimpleClass.java を参照してください。

   デフォルトでは、プライマリキー属性は署名されてはいるが、暗号化されておらず (SIGN_ONLY)、他のすべての属性は暗号化されて署名されています (ENCRYPT_AND_SIGN)。例外を指定するには、DynamoDB 用の Java クライアント側の暗号化ライブラリで定義されている暗号化アノテーションを使用します。例えば、特定の属性を署名のみにしたい場合は、@DynamoDbEncryptionSignOnly アノテーションを使用します。特定の属性に署名して暗号化コンテキストに含める場合は、@DynamoDbEncryptionSignAndIncludeInEncryptionContext注釈を使用します。特定の属性が署名も暗号化もされないようにしたい場合 (DO_NOTHING) は、@DynamoDbEncryptionDoNothing アノテーションを使用します。

   > **Note**
   >
   > SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を指定する場合、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。の定義に使用される注釈の例についてはSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、「SimpleClass4.java」を参照してください。

   アノテーションの例については、「アノテーション付きデータクラスを使用する」を参照してください。

2. 署名から除外する属性を定義する

   次の例では、すべての DO_NOTHING 属性が個別のプレフィックス「:」を共有し、そのプレフィックスを使用して、許可される署名なし属性を定義すると想定しています。クライアントは、「:」というプレフィックスが付いた属性名は署名から除外されると想定します。詳細については、「Allowed unsigned attributes」を参照してください。

   ```
   final String unsignedAttrPrefix = ":";
   ```

3. キーリングを作成します。

   次の例では AWS KMS キーリングを作成します。 AWS KMS キーリングは、対称暗号化または非対称 RSA AWS KMS keys を使用してデータキーを生成、暗号化、復号します。

   この例では、CreateMrkMultiKeyring を使用して、対称暗号化 KMS キーで AWS KMS キーリングを作成します。CreateAwsKmsMrkMultiKeyring メソッドにより、キーリングは、単一リージョンのキーとマルチリージョンのキーの両方を確実に正しく処理します。

```
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 テーブルの暗号化設定を定義する

   次の例では、この DynamoDB テーブルの暗号化設定を表す tableConfigs マップを定義します。

   この例では、DynamoDB テーブル名を論理テーブル名として指定します。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。詳細については、「AWS Database Encryption SDK for DynamoDB の暗号化設定」を参照してください。

   プレーンテキストのオーバーライドとして FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT を指定する必要があります。このポリシーは、プレーンテキスト項目の読み取りと書き込みを継続し、暗号化された項目を読み取り、クライアントが暗号化された項目を書き込むための準備を整えます。

```
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** の作成

   次の例では、ステップ 3 の tableConfigs を使用して DynamoDbEncryptionInterceptor を作成します。

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

ステップ 2: 暗号化および署名された項目を書き込む

DynamoDbEncryptionInterceptor 設定内のプレーンテキストポリシーを更新して、クライアントが暗号化および署名された項目を書き込むことを許可します。次の変更をデプロイすると、クライアントはステップ 1 で設定した属性アクションに基づいて新しい項目を暗号化して署名します。クライアントは、プレーンテキストの項目と暗号化および署名された項目を読み取ることができるようになります。

ステップ 3 に進む前に、テーブル内の既存のすべてのプレーンテキスト項目を暗号化して署名する必要があります。既存のプレーンテキスト項目を迅速に暗号化するために実行できる単一のメトリクスやクエリはありません。システムにとって最も合理的なプロセスを使用してください。例えば、定義した属性アクションと暗号化設定を使用して、時間をかけてテーブルをスキャンし、項目を書き換える非同期プロセスを使用できます。テーブル内のプレーンテキスト項目を識別するには、 AWS Database Encryption SDK が暗号化および署名されたときに項目に追加する aws_dbe_headおよび aws_dbe_foot 属性を含まないすべての項目をスキャンすることをお勧めします。

次の の例では、ステップ 1 のテーブル暗号化設定を更新します。プレーンテキストのオーバーライドを FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT を使用して更新する必要があります。このポリシーはプレーンテキスト項目を引き続き読み取りますが、暗号化された項目の読み取りと書き込みも行います。更新された DynamoDbEncryptionInterceptorを使用して新しい を作成しますtableConfigs。

```
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);
```

ステップ 3: 暗号化および署名された項目のみを読み取る

すべての項目を暗号化して署名した後、DynamoDbEncryptionInterceptor 設定内のプレーンテキストオーバーライドを更新して、暗号化および署名された項目の読み取りと書き込みのみをクライアントに許可します。次の変更をデプロイすると、クライアントはステップ 1 で設定した属性アクションに基づいて新しい項目を暗号化して署名します。クライアントは、暗号化および署名された項目のみを読み取ることができます。

次の の例では、ステップ 2 のテーブル暗号化設定を更新します。FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT を使用してプレーンテキストオーバーライドを更新することも、設定からプレーンテキストポリシーを削除することもできます。クライアントは、デフォルトでは、暗号化および署名された項目の読み取りと書き込みのみを行います。更新された DynamoDbEncryptionInterceptorを使用して新しい を作成しますtableConfigs。

```
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);
```

DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x に移行する

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x は、2.x コードベースを大幅に書き直したものです。これには、新しい構造化データ形式、マルチテナンシーのサポートの改善、シームレスなスキーマの変更、検索可能な暗号化のサポートなど、多くの更新が含まれています。このトピックでは、コードをバージョン 3.x に移行する方法について説明します。

バージョン 1.x から 2.x への移行

バージョン 3.x に移行する前に、バージョン 2.x に移行してください。バージョン 2.x では、最新プロバイダーの符号が `MostRecentProvider` から `CachingMostRecentProvider` に変更されました。現在、符号が `MostRecentProvider` である DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 1.x を使用している場合は、コード内の符号名を `CachingMostRecentProvider` に更新する必要があります。詳細については、「最新のプロバイダーに更新する」を参照してください。

バージョン 2.x から 3.x への移行

次の手順では、DynamoDB 用の Java クライアント側の暗号化ライブラリのコードをバージョン 2.x からバージョン 3.x に移行する方法について説明します。

ステップ 1. 新しい形式で項目を読み取る準備をする

Database Encryption SDK AWS クライアントが新しい形式の項目を読み取る準備をするには、次の手順を実行します。次の変更をデプロイした後、クライアントは引き続きバージョン 2.x と同じように動作します。クライアントは引き続きバージョン 2.x 形式で項目の読み取りと書き込みを行いますが、これらの変更により、クライアントが新しい形式で項目を読み取る準備が整います。

をバージョン 2.x AWS SDK for Java に更新する

> DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x では、DynamoDB Enhanced Client が必要です。DynamoDB Enhanced Client は、以前のバージョンで使用されていた DynamoDBMapper を置き換えます。拡張クライアントを使用するには、 AWS SDK for Java 2.xを使用する必要があります。

AWS SDK for Javaのバージョン 1.x から 2.x に移行する手順に従います。

必要な AWS SDK for Java 2.x モジュールの詳細については、「」を参照してください前提条件。

従来のバージョンによって暗号化された項目を読み取るようにクライアントを設定する

次の手順では、以下のコード例で示されているステップの概要を説明します。

1. キーリングを作成します。

   キーリングと暗号マテリアルマネージャーは、DynamoDB 用の Java クライアント側の暗号化ライブラリの以前のバージョンで使用されていた暗号マテリアルプロバイダーを置き換えます。

   > **Important**
   >
   > キーリングの作成時に指定するラッピングキーは、バージョン 2.x で暗号マテリアルプロバイダーで使用したのと同じラッピングキーである必要があります。

2. アノテーション付きクラスに基づいてテーブルスキーマを作成します。

   このステップでは、新しい形式で項目の書き込みを開始するときに使用される属性アクションを定義します。

   新しい DynamoDB Enhanced Client の使用に関するガイダンスについては、「AWS SDK for Java デベロッパーガイド」の「TableSchema を生成する」を参照してください。

   次の例では、新しい属性アクションのアノテーションを使用して、アノテーション付きクラスをバージョン 2.x から更新したことを前提としています。属性アクションにアノテーションを付ける方法のガイダンスについては、「アノテーション付きデータクラスを使用する」を参照してください。

   > **Note**
   >
   > SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を指定する場合、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。の定義に使用される注釈の例についてはSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、「SimpleClass4.java」を参照してください。

3. どの属性を署名から除外するかを定義します。
4. バージョン 2.x のモデル化されたクラスで設定された属性アクションの明示的なマッピングを設定します。

   このステップでは、古い形式で項目を書き込むために使用される属性アクションを定義します。
5. DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 2.x で使用した `DynamoDBEncryptor` を設定します。
6. 従来の動作を設定します。
7. `DynamoDbEncryptionInterceptor` を作成します。
8. 新しい AWS SDK DynamoDB クライアントを作成します。
9. `DynamoDBEnhancedClient` を作成し、モデル化されたクラスを含むテーブルを作成します。

   DynamoDB Enhanced Client の詳細については、「拡張クライアントを作成する」を参照してください。

```
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);
    }
}
```

ステップ 2. 新しい形式で項目を書き込む

ステップ 1 からすべてのリーダーに変更をデプロイしたら、次のステップを実行して、新しい形式で項目を書き込むように AWS Database Encryption SDK クライアントを設定します。次の変更をデプロイした後、クライアントは引き続き古い形式で項目を読み取り、新しい形式で項目の書き込みと読み取りを開始します。

次の手順では、以下のコード例で示されているステップの概要を説明します。

1. ステップ 1 と同様に、キーリング、テーブルスキーマ、従来の属性アクション、`allowedUnsignedAttributes`、および `DynamoDBEncryptor` の設定を続行します。
2. 新しい形式を使用して新しい項目のみを書き込むように、従来の動作を更新します。
3. `DynamoDbEncryptionInterceptor` を作成する
4. 新しい AWS SDK DynamoDB クライアントを作成します。
5. `DynamoDBEnhancedClient` を作成し、モデル化されたクラスを含むテーブルを作成します。

   DynamoDB Enhanced Client の詳細については、「拡張クライアントを作成する」を参照してください。

```
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);
    }
}
```

ステップ 2 の変更をデプロイした後、ステップ 3 に進む前に、テーブル内のすべての古い項目を新しい形式で再暗号化する必要があります。既存の項目を迅速に暗号化するために実行できる単一のメトリクスやクエリはありません。システムにとって最も合理的なプロセスを使用してください。例えば、定義した新しい属性アクションと暗号化設定を使用して、時間をかけてテーブルをスキャンし、項目を書き換える非同期プロセスを使用できます。

ステップ 3. 新しい形式でのみ項目を読み書きする

テーブル内のすべての項目を新しい形式で再暗号化した後、設定から従来の動作を削除できます。新しい形式でのみ項目を読み書きするようにクライアントを設定するには、次のステップを実行します。

次の手順では、以下のコード例で示されているステップの概要を説明します。

1. ステップ 1 と同様に、キーリング、テーブルスキーマ、allowedUnsignedAttributes の設定を続行します。従来の属性アクションと DynamoDBEncryptor を設定から削除します。
2. DynamoDbEncryptionInterceptor を作成します。
3. 新しい AWS SDK DynamoDB クライアントを作成します。
4. DynamoDBEnhancedClient を作成し、モデル化されたクラスを含むテーブルを作成します。

   DynamoDB Enhanced Client の詳細については、「拡張クライアントを作成する」を参照してください。

```
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

このトピックでは、DynamoDB 用の .NET クライアント側の暗号化ライブラリのバージョン 3.x をインストールして使用する方法について説明します。 AWS Database Encryption SDK for DynamoDB を使用したプログラミングの詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの .NET の例を参照してください。

DynamoDB 用の .NET クライアント側の暗号化ライブラリは、C# やその他の .NET プログラミング言語でアプリケーションを記述している開発者を対象としています。Windows、macOS、Linux でサポートされています。

AWS Database Encryption SDK for DynamoDB のすべてのプログラミング言語実装は相互運用可能です。ただし、 SDK for .NET では、リストデータ型またはマップデータ型の空の値はサポートされていません。つまり、DynamoDB 用の Java クライアント側の暗号化ライブラリを使用して、リストまたはマップデータ型の空の値を含む項目を書き込む場合、DynamoDB 用の .NET クライアント側の暗号化ライブラリを使用してその項目を復号化して読み取ることはできません。

トピック

- DynamoDB 用の .NET クライアント側の暗号化ライブラリのインストール
- .NET を使用したデバッグ
- DynamoDB 用の .NET クライアント側の暗号化ライブラリの使用
- .NET の例

- AWS Database Encryption SDK for DynamoDB を使用するように既存の DynamoDB テーブルを設定する

DynamoDB 用の .NET クライアント側の暗号化ライブラリのインストール

DynamoDB 用の .NET クライアント側の暗号化ライブラリは、NuGet の AWS.Cryptography.DbEncryptionSDK.DynamoDb パッケージとして利用できます。ライブラリのインストールと構築の詳細については、aws-database-encryption-sdk-dynamodb リポジトリの .NET README.md ファイルを参照してください。DynamoDB 用の .NET クライアント側の暗号化ライブラリには、 AWS Key Management Service (AWS KMS) キーを使用していない場合 SDK for .NET でも が必要です。は NuGet SDK for .NET パッケージと共にインストールされます。

DynamoDB 用の .NET クライアント側の暗号化ライブラリのバージョン 3.x は、.NET 6.0 および .NET Framework net48 以降をサポートしています。

.NET を使用したデバッグ

DynamoDB 用の .NET クライアント側の暗号化ライブラリはログを生成しません。DynamoDB 用の .NET クライアント側の暗号化ライブラリの例外は例外メッセージを生成しますが、スタックトレースは生成されません。

デバッグしやすいように、 SDK for .NETへのログ記録を必ず有効にしてください。からのログとエラーメッセージ SDK for .NET は、 で発生するエラーを、DynamoDB 用の .NET クライアント側の暗号化ライブラリのエラー SDK for .NET と区別するのに役立ちます。 SDK for .NET ログ記録については、「 AWS SDK for .NET デベロッパーガイド」のAWSLogging」を参照してください。(このトピックを確認するには、[.NET Framework コンテンツを開く] セクションを展開してください)。

DynamoDB 用の .NET クライアント側の暗号化ライブラリの使用

このトピックでは、DynamoDB 用の .NET クライアント側の暗号化ライブラリのバージョン 3.x の関数とヘルパークラスの一部について説明します。

DynamoDB 用の .NET クライアント側の暗号化ライブラリを使用したプログラミングの詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの .NET の例を参照してください。

トピック

- 項目エンクリプタ
- AWS Database Encryption SDK for DynamoDB の属性アクション

- AWS Database Encryption SDK for DynamoDB の暗号化設定
- AWS Database Encryption SDK を使用した項目の更新

項目エンクリプタ

その中核となる AWS Database Encryption SDK for DynamoDB は項目エンクリプタです。DynamoDB 用の .NET クライアント側の暗号化ライブラリのバージョン 3.x を使用して、DynamoDB テーブル項目を次の方法で暗号化、署名、検証、復号できます。

DynamoDB API 用の低レベル AWS データベース暗号化 SDK

テーブル暗号化設定を使用して、DynamoDB `PutItem`リクエストでクライアント側で項目を自動的に暗号化して署名する DynamoDB クライアントを構築できます。このクライアントを直接使用するか、ドキュメントモデルまたはオブジェクト永続性モデルを構築できます。

検索可能な暗号化を使用するには、低レベルの AWS Database Encryption SDK for DynamoDB API を使用する必要があります。

下位レベルの **`DynamoDbItemEncryptor`**

下位レベルの `DynamoDbItemEncryptor` は、DynamoDB を呼び出すことなく、テーブル項目を直接暗号化して署名するか、または復号して検証します。DynamoDB の `PutItem` または `GetItem` リクエストは実行しません。例えば、下位レベルの `DynamoDbItemEncryptor` を使用して、既に取得した DynamoDB 項目を直接復号して検証できます。低レベルの を使用する場合は`DynamoDbItemEncryptor`、 が DynamoDB との通信 SDK for .NET に提供する低レベルのプログラミングモデルを使用することをお勧めします。

下位レベルの `DynamoDbItemEncryptor` は、検索可能な暗号化をサポートしていません。

AWS Database Encryption SDK for DynamoDB の属性アクション

属性アクションは、暗号化および署名される属性値、署名のみされる属性値、署名および暗号化コンテキストに含まれる属性値、および無視される属性値を決定します。

.NET クライアントで属性アクションを指定するには、オブジェクトモデルを使用して属性アクションを手動で定義します。名前と値のペアが属性名と指定されたアクションを表す`Dictionary`オブジェクトを作成して、属性アクションを指定します。

属性を暗号化して署名するように `ENCRYPT_AND_SIGN` を指定します。属性に署名するが暗号化はしないように `SIGN_ONLY` を指定します。`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` を指

定して属性に署名し、暗号化コンテキストに含めます。属性に署名することなく、その属性を暗号化することはできません。属性を無視するように DO_NOTHING を指定します。

パーティション属性とソート属性は、 SIGN_ONLY または のいずれかである必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。属性を として定義する場合SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 属性アクションを定義した後、どの属性を署名から除外するかを定義する必要があります。将来、新しい署名なし属性を簡単に追加できるように、署名なし属性を識別するための個別のプレフィックス (「:」など) を選択することをお勧めします。DynamoDB スキーマと属性アクションを定義するときに DO_NOTHING とマークされたすべての属性の属性名にこのプレフィックスを含めます。

次のオブジェクトモデルはENCRYPT_AND_SIGN、.NET クライアントで SIGN_ONLY、SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、、および DO_NOTHING 属性アクションを指定する方法を示しています。この例では、プレフィックス:「」を使用してDO_NOTHING属性を識別します。

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 暗号化アクションを使用するには、AWS Database Encryption SDK のバージョン 3.3 以降を使用する必要があります。データモデルを更新して を含める前に、すべてのリーダーに新しいバージョンをデプロイしますSIGN_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 DynamoDB の暗号化設定

AWS Database Encryption SDK を使用する場合は、DynamoDB テーブルの暗号化設定を明示的に定義する必要があります。暗号化設定に必要な値は、属性アクションを手動で定義したか、またはアノテーション付きデータクラスを使用して定義したかによって異なります。

次のスニペットは、低レベルの AWS Database Encryption SDK for DynamoDB API と、個別のプレフィックスで定義された許可された署名なし属性を使用して、DynamoDB テーブル暗号化設定を定義します。

```
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);
```

論理テーブル名

DynamoDB テーブルの論理テーブル名。

論理テーブル名は、DynamoDB の復元オペレーションを簡素化するために、テーブルに格納されているすべてのデータに暗号的にバインドされます。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。常に同じ論理テーブル名を指定する必要があります。復号を成功させるには、論理テーブル名が、暗号化の際に指定された名前と一致する必要があります。DynamoDB テーブルをバックアップから復

元した後に DynamoDB テーブル名が変更された場合でも、論理テーブル名を使用することで、復号オペレーションで引き続きテーブルが確実に認識されます。

許可された署名なし属性

属性アクションで DO_NOTHING とマークされた属性。

許可された署名なし属性は、どの属性が署名から除外されるかをクライアントに伝えます。クライアントは、他のすべての属性が署名に含まれていると想定します。その後、レコードを復号する際に、クライアントは、ユーザーが指定する、許可された署名なし属性の中からどの属性を検証する必要があり、どの属性を無視する必要があるかを決定します。許可された署名なし属性から属性を削除することはできません。

すべての DO_NOTHING 属性をリストする配列を作成することで、許可された署名なし属性を明示的に定義できます。また、DO_NOTHING 属性に名前を付ける際に個別のプレフィックスを指定し、そのプレフィックスを使用してどの属性が署名されていないかをクライアントに伝えることもできます。将来新しい DO_NOTHING 属性を追加するプロセスが簡素化されるため、個別のプレフィックスを指定することを強くお勧めします。詳細については、「データモデルの更新」を参照してください。

すべての DO_NOTHING 属性のためにプレフィックスを指定しない場合は、クライアントが復号時に署名されていないことを想定するすべての属性を明示的にリストする allowedUnsignedAttributes 配列を設定できます。どうしても必要な場合にのみ、許可された署名なし属性を明示的に定義する必要があります。

検索設定 (オプション)

SearchConfig はビーコンのバージョンを定義します。

検索可能な暗号化または署名付きビーコンを使用するには、SearchConfig を指定する必要があります。

アルゴリズムスイート (オプション)

algorithmSuiteId は、 AWS Database Encryption SDK が使用するアルゴリズムスイートを定義します。

代替アルゴリズムスイートを明示的に指定しない限り、 AWS Database Encryption SDK はデフォルトのアルゴリズムスイートを使用します。デフォルトのアルゴリズムスイートは、キーの導出、デジタル署名、およびキーコミットメントを備えた AES-GCM アルゴリズムを使用します。デフォルトのアルゴリズムスイートはほとんどのアプリケーションに適している可能性がありますが、代替アルゴリズムスイートを選択できます。例えば、一部の信頼モデルは、デジタル署名を含まないアルゴリズムスイートによって満たされます。 AWS Database Encryption

SDK がサポートするアルゴリズムスイートの詳細については、「」を参照してくださいAWS Database Encryption SDK でサポートされているアルゴリズムスイート。

ECDSA デジタル署名を使用しない AES-GCM アルゴリズムスイートを選択するには、テーブル暗号化設定に次のスニペットを含めます。

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

AWS Database Encryption SDK を使用した項目の更新

AWS Database Encryption SDK は、暗号化または署名された属性を含む項目に対して ddb:UpdateItem をサポートしていません。暗号化または署名された属性を更新するには、ddb:PutItem を使用する必要があります。PutItem リクエストで既存の項目と同じプライマリキーを指定すると、新しい項目が既存の項目に完全に置き換わります。CLOBBER を使用して、項目を更新した後、保存する際にすべての属性をクリアして置き換えることもできます。

.NET の例

次の例は、DynamoDB 用の .NET クライアント側の暗号化ライブラリを使用して、アプリケーション内のテーブル項目を保護する方法を示しています。その他の例を見つける (および独自の例を提供する) には、GitHub の aws-database-encryption-sdk-dynamodb リポジトリにある .NET の例を参照してください。

次の例は、入力されていない新しい Amazon DynamoDB テーブルで DynamoDB 用の .NET クライアント側の暗号化ライブラリを設定する方法を示しています。既存の Amazon DynamoDB テーブルをクライアント側の暗号化のために設定する場合は、「既存のテーブルにバージョン 3.x を追加する」を参照してください。

トピック

- 低レベルの AWS Database Encryption SDK for DynamoDB API の使用
- 下位レベルの使用 DynamoDbItemEncryptor

低レベルの AWS Database Encryption SDK for DynamoDB API の使用

次の例は、 AWS KMS キーリングで低レベルの AWS Database Encryption SDK for DynamoDB API を使用して、DynamoDB PutItemリクエストでクライアント側で項目を自動的に暗号化して署名する方法を示しています。

サポートされている任意のキーリングを使用できますが、可能な限りいずれかの AWS KMS キーリングを使用することをお勧めします。

完全なコードサンプルを参照: BasicPutGetExample.cs

ステップ 1: AWS KMS キーリングを作成する

次の例では、 CreateAwsKmsMrkMultiKeyringを使用して対称暗号化 KMS AWS KMS キーを持つ キーリングを作成します。CreateAwsKmsMrkMultiKeyring メソッドにより、キーリングは、単一リージョンのキーとマルチリージョンのキーの両方を確実に正しく処理します。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

ステップ 2: 属性アクションを設定する

次の例では、テーブル項目のサンプル属性アクションを表す attributeActionsOnEncrypt ディクショナリを定義します。

> **Note**
>
> 次の例では、属性を として定義していませんSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。SIGN_AND_INCLUDE_IN_ENCRYPTION_C 属性を指定する場合、パーティション属性とソート属性も である必要がありますSIGN_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
};
```

ステップ 3: 署名から除外する属性を定義する

次の例では、すべての DO_NOTHING 属性が個別のプレフィックス「:」を共有し、そのプレフィックスを使用して、許可される署名なし属性を定義すると想定しています。クライアントは、「:」というプレフィックスが付いた属性名は署名から除外されると想定します。詳細については、「Allowed unsigned attributes」を参照してください。

```
const String unsignAttrPrefix = ":";
```

ステップ 4: DynamoDB テーブルの暗号化設定を定義する

次の例では、この DynamoDB テーブルの暗号化設定を表す tableConfigs マップを定義します。

この例では、DynamoDB テーブル名を論理テーブル名として指定します。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。詳細については、「AWS Database Encryption SDK for DynamoDB の暗号化設定」を参照してください。

> **Note**
>
> 検索可能な暗号化または署名付きビーコンを使用するには、暗号化設定に SearchConfig も含める必要があります。

```
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);
```

ステップ 5: 新しい AWS SDK DynamoDB クライアントを作成する

次の例では、ステップ 4 TableEncryptionConfigsの を使用して新しい AWS SDK DynamoDB クライアントを作成します。

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

ステップ 6: DynamoDB テーブル項目を暗号化して署名する

次の例では、サンプルテーブル項目を表すitemディクショナリを定義し、その項目を DynamoDB テーブルに配置します。項目は、DynamoDB に送信される前に、クライアント側で暗号化および署名されます。

```
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);
```

下位レベルの使用 **DynamoDbItemEncryptor**

次の例は、下位レベルの DynamoDbItemEncryptor を AWS KMS キーリングとともに使用して、テーブル項目を直接暗号化して署名する方法を示しています。DynamoDbItemEncryptor は項目を DynamoDB テーブルに配置しません。

DynamoDB 拡張クライアントではサポートされている任意のキーリングを使用できますが、可能な限りいずれかの AWS KMS キーリングを使用することをお勧めします。

> **Note**
>
> 下位レベルの DynamoDbItemEncryptor は、検索可能な暗号化をサポートしていません。検索可能な暗号化を使用するには、低レベルの AWS Database Encryption SDK for DynamoDB API を使用します。

完全なコードサンプル「ItemEncryptDecryptExample.cs」を参照してください。

ステップ 1: AWS KMS キーリングを作成する

次の例では、 CreateAwsKmsMrkMultiKeyringを使用して対称暗号化 KMS AWS KMS キーを持つ キーリングを作成します。CreateAwsKmsMrkMultiKeyring メソッドにより、キーリングは、単一リージョンのキーとマルチリージョンのキーの両方を確実に正しく処理します。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

ステップ 2: 属性アクションを設定する

次の例では、テーブル項目のサンプル属性アクションを表す attributeActionsOnEncrypt ディクショナリを定義します。

> **Note**
>
> 次の例では、属性を として定義していませんSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。SIGN_AND_INCLUDE_IN_ENCRYPTION_C 属性を指定する場合、パーティション属性とソート属性も である必要がありますSIGN_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
```

```
};
```

ステップ 3: 署名から除外する属性を定義する

次の例では、すべての DO_NOTHING 属性が個別のプレフィックス「:」を共有し、そのプレフィックスを使用して、許可される署名なし属性を定義すると想定しています。クライアントは、「:」というプレフィックスが付いた属性名は署名から除外されると想定します。詳細については、「Allowed unsigned attributes」を参照してください。

```
String unsignAttrPrefix = ":";
```

ステップ 4: **DynamoDbItemEncryptor** 設定を定義する

次の例では、DynamoDbItemEncryptor の設定を定義します。

この例では、DynamoDB テーブル名を論理テーブル名として指定します。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。詳細については、「AWS Database Encryption SDK for DynamoDB の暗号化設定」を参照してください。

```
var config = new DynamoDbItemEncryptorConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
```

ステップ 5: **DynamoDbItemEncryptor** を作成する

次の例では、ステップ 4 の config を使用して新しい DynamoDbItemEncryptor を作成します。

```
var itemEncryptor = new DynamoDbItemEncryptor(config);
```

ステップ 6: テーブル項目を直接暗号化して署名する

次の例では、DynamoDbItemEncryptor を使用して項目を直接暗号化し、署名します。DynamoDbItemEncryptor は項目を DynamoDB テーブルに配置しません。

```
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;
```

AWS Database Encryption SDK for DynamoDB を使用するように既存の DynamoDB テーブルを設定する

DynamoDB 用の .NET クライアント側の暗号化ライブラリのバージョン 3.x では、クライアント側の暗号化用に既存の Amazon DynamoDB テーブルを設定できます。このトピックでは、データが入力されている既存の DynamoDB テーブルにバージョン 3.x を追加するために必要な 3 つのステップについてのガイダンスを提供します。

ステップ 1: 暗号化された項目の読み取りと書き込みの準備をする

Database Encryption SDK AWS クライアントが暗号化された項目を読み書きできるように準備するには、次の手順を実行します。次の変更をデプロイした後も、クライアントは引き続きプレーンテキスト項目の読み取りと書き込みを行います。テーブルに書き込まれる新しい項目の暗号化や署名は行いませんが、暗号化された項目が表示されるとすぐに復号できます。これらの変更により、クライアントが新しい項目の暗号化を開始するための準備が整います。次のステップに進む前に、次の変更を各リーダーにデプロイする必要があります。

1. 属性アクションを定義する

   オブジェクトモデルを作成して、暗号化および署名される属性値、署名のみされる属性値、および無視される属性値を定義します。

   デフォルトでは、プライマリキー属性は署名されてはいるが、暗号化されておらず (SIGN_ONLY)、他のすべての属性は暗号化されて署名されています (ENCRYPT_AND_SIGN)。

   属性を暗号化して署名するように ENCRYPT_AND_SIGN を指定します。属性に署名するが暗号化はしないように SIGN_ONLY を指定します。署名と属

性SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTを指定し、暗号化コンテキストに含めます。属性に署名することなく、その属性を暗号化することはできません。属性を無視するようにDO_NOTHING を指定します。詳細については、「AWS Database Encryption SDK for DynamoDB の属性アクション」を参照してください。

> **Note**
>
> SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性を指定する場合、パーティション属性とソート属性も である必要がありますSIGN_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. 署名から除外する属性を定義する

次の例では、すべての DO_NOTHING 属性が個別のプレフィックス「:」を共有し、そのプレフィックスを使用して、許可される署名なし属性を定義すると想定しています。クライアントは、「:」というプレフィックスが付いた属性名は署名から除外されると想定します。詳細については、「Allowed unsigned attributes」を参照してください。

```
const String unsignAttrPrefix = ":";
```

3. キーリングを作成します。

次の例では AWS KMS キーリングを作成します。 AWS KMS キーリングは、対称暗号化または非対称 RSA AWS KMS keys を使用してデータキーを生成、暗号化、復号します。

この例では、CreateMrkMultiKeyring を使用して、対称暗号化 KMS キーで AWS KMS キーリングを作成します。CreateAwsKmsMrkMultiKeyring メソッドにより、キーリングは、単一リージョンのキーとマルチリージョンのキーの両方を確実に正しく処理します。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. DynamoDB テーブルの暗号化設定を定義する

次の例では、この DynamoDB テーブルの暗号化設定を表す `tableConfigs` マップを定義します。

この例では、DynamoDB テーブル名を論理テーブル名として指定します。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。

プレーンテキストのオーバーライドとして `FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` を指定する必要があります。このポリシーは、プレーンテキスト項目の読み取りと書き込みを継続し、暗号化された項目を読み取り、クライアントが暗号化された項目を書き込むための準備を整えます。

テーブル暗号化設定に含まれる値の詳細については、「」を参照してくださいAWS Database Encryption SDK for DynamoDB の暗号化設定。

```
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. 新しい AWS SDK DynamoDB クライアントを作成する

次の の例では、ステップ 4 `TableEncryptionConfigs`の を使用して新しい AWS SDK DynamoDB クライアントを作成します。

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

ステップ 2: 暗号化および署名された項目を書き込む

テーブル暗号化設定のプレーンテキストポリシーを更新して、クライアントが暗号化および署名された項目を書き込むことを許可します。次の変更をデプロイすると、クライアントはステップ 1 で設定した属性アクションに基づいて新しい項目を暗号化して署名します。クライアントは、プレーンテキストの項目と暗号化および署名された項目を読み取ることができるようになります。

ステップ 3 に進む前に、テーブル内の既存のすべてのプレーンテキスト項目を暗号化して署名する必要があります。既存のプレーンテキスト項目を迅速に暗号化するために実行できる単一のメトリクスやクエリはありません。システムにとって最も合理的なプロセスを使用してください。例えば、定義した属性アクションと暗号化設定を使用して、時間をかけてテーブルをスキャンし、項目を書き換える非同期プロセスを使用できます。テーブル内のプレーンテキスト項目を識別するには、 AWS Database Encryption SDK が暗号化および署名されたときに項目に追加する aws_dbe_headおよび aws_dbe_foot 属性を含まないすべての項目をスキャンすることをお勧めします。

次の の例では、ステップ 1 のテーブル暗号化設定を更新します。プレーンテキストのオーバーライドを FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT を使用して更新する必要があります。このポリシーはプレーンテキスト項目を引き続き読み取りますが、暗号化された項目の読み取りと書き込みも行います。更新された を使用して新しい AWS SDK DynamoDB クライアントを作成します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);
```

ステップ 3: 暗号化および署名された項目のみを読み取る

すべての項目を暗号化して署名したら、テーブル暗号化設定のプレーンテキストオーバーライドを更新して、クライアントが暗号化および署名された項目の読み取りと書き込みのみを許可します。次の変更をデプロイすると、クライアントはステップ 1 で設定した属性アクションに基づいて新しい項

目を暗号化して署名します。クライアントは、暗号化および署名された項目のみを読み取ることができます。

次の の例では、ステップ 2 のテーブル暗号化設定を更新します。FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT を使用してプレーンテキストオーバーライドを更新することも、設定からプレーンテキストポリシーを削除することもできます。クライアントは、デフォルトでは、暗号化および署名された項目の読み取りと書き込みのみを行います。更新された を使用して新しい AWS SDK DynamoDB クライアントを作成します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

このトピックでは、DynamoDB 用の Rust クライアント側の暗号化ライブラリのバージョン 1.x をインストールして使用する方法について説明します。 AWS Database Encryption SDK for DynamoDB を使用したプログラミングの詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの Rust の例を参照してください。

AWS Database Encryption SDK for DynamoDB のすべてのプログラミング言語実装は相互運用可能です。

トピック

- 前提条件
- インストール
- DynamoDB 用の Rust クライアント側の暗号化ライブラリの使用

前提条件

DynamoDB 用の Rust クライアント側の暗号化ライブラリをインストールする前に、次の前提条件があることを確認してください。

Rust と Cargo をインストールする

rustup を使用して Rust の現在の安定リリースをインストールします。

rustup のダウンロードとインストールの詳細については、「The Cargo Book」の「インストール手順」を参照してください。

インストール

DynamoDB 用の Rust クライアント側の暗号化ライブラリは、Crates.io の aws-db-esdk クレートとして利用できます。ライブラリのインストールと構築の詳細については、aws-database-encryption-sdk-dynamodb GitHub リポジトリの README.md ファイルを参照してください。

手動

DynamoDB 用の Rust クライアント側の暗号化ライブラリをインストールするには、aws-database-encryption-sdk-dynamodb GitHub リポジトリのクローンを作成するか、ダウンロードします。

最新バージョンをインストールするには

プロジェクトディレクトリで次の Cargo コマンドを実行します。

```
cargo add aws-db-esdk
```

または、次の行を Cargo.toml に追加します。

```
aws-db-esdk = "<version>"
```

DynamoDB 用の Rust クライアント側の暗号化ライブラリの使用

このトピックでは、DynamoDB 用の Rust クライアント側の暗号化ライブラリのバージョン 1.x の関数とヘルパークラスの一部について説明します。

DynamoDB 用の Rust クライアント側の暗号化ライブラリを使用したプログラミングの詳細については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの Rust の例を参照してください。

トピック

- 項目エンクリプタ
- AWS Database Encryption SDK for DynamoDB の属性アクション
- AWS Database Encryption SDK for DynamoDB の暗号化設定
- AWS Database Encryption SDK を使用した項目の更新

項目エンクリプタ

その中核となる AWS Database Encryption SDK for DynamoDB は項目エンクリプタです。DynamoDB 用の Rust クライアント側の暗号化ライブラリのバージョン 1.x を使用して、DynamoDB テーブル項目を次の方法で暗号化、署名、検証、復号できます。

DynamoDB API 用の低レベル AWS データベース暗号化 SDK

テーブル暗号化設定を使用して、DynamoDB `PutItem`リクエストでクライアント側で項目を自動的に暗号化して署名する DynamoDB クライアントを構築できます。

検索可能な暗号化を使用するには、低レベルの AWS Database Encryption SDK for DynamoDB API を使用する必要があります。

DynamoDB API 用の低レベルの AWS Database Encryption SDK の使用方法を示す例については、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの basic_get_put_example.rs を参照してください。

下位レベルの **`DynamoDbItemEncryptor`**

下位レベルの `DynamoDbItemEncryptor` は、DynamoDB を呼び出すことなく、テーブル項目を直接暗号化して署名するか、または復号して検証します。DynamoDB の `PutItem` または `GetItem` リクエストは実行しません。例えば、下位レベルの `DynamoDbItemEncryptor` を使用して、既に取得した DynamoDB 項目を直接復号して検証できます。

下位レベルの `DynamoDbItemEncryptor` は、検索可能な暗号化をサポートしていません。

下位レベルの の使用方法を示す例については`DynamoDbItemEncryptor`、GitHub の aws-database-encryption-sdk-dynamodb リポジトリの item_encrypt_decrypt.rs を参照してください。

AWS Database Encryption SDK for DynamoDB の属性アクション

属性アクションは、暗号化および署名される属性値、署名のみされる属性値、署名および暗号化コンテキストに含まれる属性値、および無視される属性値を決定します。

Rust クライアントで属性アクションを指定するには、オブジェクトモデルを使用して属性アクションを手動で定義します。名前と値のペアが属性名と指定されたアクションを表すHashMapオブジェクトを作成して、属性アクションを指定します。

属性を暗号化して署名するように ENCRYPT_AND_SIGN を指定します。属性に署名するが暗号化はしないように SIGN_ONLY を指定します。属性に署名SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTして暗号化コンテキストに含めるには、 を指定します。属性に署名することなく、その属性を暗号化することはできません。属性を無視するように DO_NOTHING を指定します。

パーティション属性とソート属性は、 SIGN_ONLY または のいずれかである必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。属性を として定義する場合SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、パーティション属性とソート属性も である必要がありますSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 属性アクションを定義した後、どの属性を署名から除外するかを定義する必要があります。将来、新しい署名なし属性を簡単に追加できるように、署名なし属性を識別するための個別のプレフィックス (「:」など) を選択することをお勧めします。DynamoDB スキーマと属性アクションを定義するときに DO_NOTHING とマークされたすべての属性の属性名にこのプレフィックスを含めます。

次のオブジェクトモデルはENCRYPT_AND_SIGN、Rust クライアントで SIGN_ONLY、SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、、および DO_NOTHING 属性アクションを指定する方法を示しています。この例では、プレフィックス:「」を使用してDO_NOTHING属性を識別します。

```
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 DynamoDB の暗号化設定

AWS Database Encryption SDK を使用する場合は、DynamoDB テーブルの暗号化設定を明示的に定義する必要があります。暗号化設定に必要な値は、属性アクションを手動で定義したか、またはアノテーション付きデータクラスを使用して定義したかによって異なります。

次の入二ペットは、低レベルの AWS Database Encryption SDK for DynamoDB API と、個別のプレフィックスで定義された許可された署名なし属性を使用して、DynamoDB テーブル暗号化設定を定義します。

```
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()?;
```

論理テーブル名

DynamoDB テーブルの論理テーブル名。

論理テーブル名は、DynamoDB の復元オペレーションを簡素化するために、テーブルに格納されているすべてのデータに暗号的にバインドされます。最初に暗号化設定を定義する際に、DynamoDB テーブル名を論理テーブル名として指定することを強くお勧めします。常に同じ論理テーブル名を指定する必要があります。復号を成功させるには、論理テーブル名が、暗号化の際に指定された名前と一致する必要があります。DynamoDB テーブルをバックアップから復元した後に DynamoDB テーブル名が変更された場合でも、論理テーブル名を使用することで、復号オペレーションで引き続きテーブルが確実に認識されます。

許可された署名なし属性

属性アクションで DO_NOTHING とマークされた属性。

許可された署名なし属性は、どの属性が署名から除外されるかをクライアントに伝えます。クライアントは、他のすべての属性が署名に含まれていると想定します。その後、レコードを復号する際に、クライアントは、ユーザーが指定する、許可された署名なし属性の中からどの属性を検証する必要があり、どの属性を無視する必要があるかを決定します。許可された署名なし属性から属性を削除することはできません。

すべての DO_NOTHING 属性をリストする配列を作成することで、許可された署名なし属性を明示的に定義できます。また、DO_NOTHING 属性に名前を付ける際に個別のプレフィックスを指定し、そのプレフィックスを使用してどの属性が署名されていないかをクライアントに伝えることもできます。将来新しい DO_NOTHING 属性を追加するプロセスが簡素化されるため、個別のプレフィックスを指定することを強くお勧めします。詳細については、「データモデルの更新」を参照してください。

すべての DO_NOTHING 属性のためにプレフィックスを指定しない場合は、クライアントが復号時に署名されていないことを想定するすべての属性を明示的にリストする allowedUnsignedAttributes 配列を設定できます。どうしても必要な場合にのみ、許可された署名なし属性を明示的に定義する必要があります。

検索設定 (オプション)

SearchConfig はビーコンのバージョンを定義します。

検索可能な暗号化または署名付きビーコンを使用するには、SearchConfig を指定する必要があります。

アルゴリズムスイート (オプション)

algorithmSuiteId は、 AWS Database Encryption SDK が使用するアルゴリズムスイートを定義します。

代替アルゴリズムスイートを明示的に指定しない限り、 AWS Database Encryption SDK はデフォルトのアルゴリズムスイートを使用します。デフォルトのアルゴリズムスイートは、キーの導出、デジタル署名、およびキーコミットメントを備えた AES-GCM アルゴリズムを使用します。デフォルトのアルゴリズムスイートはほとんどのアプリケーションに適している可能性がありますが、代替アルゴリズムスイートを選択できます。例えば、一部の信頼モデルは、デジタル署名を含まないアルゴリズムスイートによって満たされます。 AWS Database Encryption

SDK がサポートするアルゴリズムスイートの詳細については、「」を参照してくださいAWS Database Encryption SDK でサポートされているアルゴリズムスイート。

ECDSA デジタル署名のない AES-GCM アルゴリズムスイートを選択するには、テーブル暗号化設定に次のスニペットを含めます。

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

AWS Database Encryption SDK を使用した項目の更新

AWS Database Encryption SDK は、暗号化または署名された属性を含む項目に対して ddb:UpdateItem をサポートしていません。暗号化または署名された属性を更新するには、ddb:PutItem を使用する必要があります。PutItem リクエストで既存の項目と同じプライマリキーを指定すると、新しい項目が既存の項目に完全に置き換わります。

レガシー DynamoDB 暗号化クライアント

2023 年 6 月 9 日、クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。 AWS Database Encryption SDK は、引き続きレガシー DynamoDB 暗号化クライアントバージョンをサポートします。名前の変更によって変更されたクライアント側の暗号化ライブラリのさまざまな部分の詳細については、「Amazon DynamoDB Encryption Client の名前の変更」を参照してください。

DynamoDB 用の Java クライアント側の暗号化ライブラリの最新バージョンに移行するには、「バージョン 3.x に移行する」を参照してください。

トピック

- AWS Database Encryption SDK for DynamoDB バージョンのサポート
- DynamoDB 暗号化クライアントの仕組み
- Amazon DynamoDB Encryption Client の概念
- 暗号マテリアルプロバイダー
- Amazon DynamoDB Encryption Client で利用可能なプログラミング言語
- データモデルの変更
- DynamoDB 暗号化クライアントアプリケーションの問題のトラブルシューティング

AWS Database Encryption SDK for DynamoDB バージョンのサポート

「レガシー」の章のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x～2.x および DynamoDB Encryption Client for Python のバージョン 1.x～3.x に関する情報が記載されています。

次の表には、Amazon DynamoDB でクライアント側の暗号化をサポートする言語とバージョンがリストされています。

プログラミング言語	バージョン	SDK メジャーバージョンのライフサイクルフェーズ
Java	バージョン 1.x	サポート終了フェーズ、2022 年 7 月発効
Java	バージョン 2.x	一般提供 (GA)
Java	バージョン 3.x	一般提供 (GA)
Python	バージョン 1.x	サポート終了フェーズ、2022 年 7 月発効
Python	バージョン 2.x	サポート終了フェーズ、2022 年 7 月発効
Python	バージョン 3.x	一般提供 (GA)

DynamoDB 暗号化クライアントの仕組み

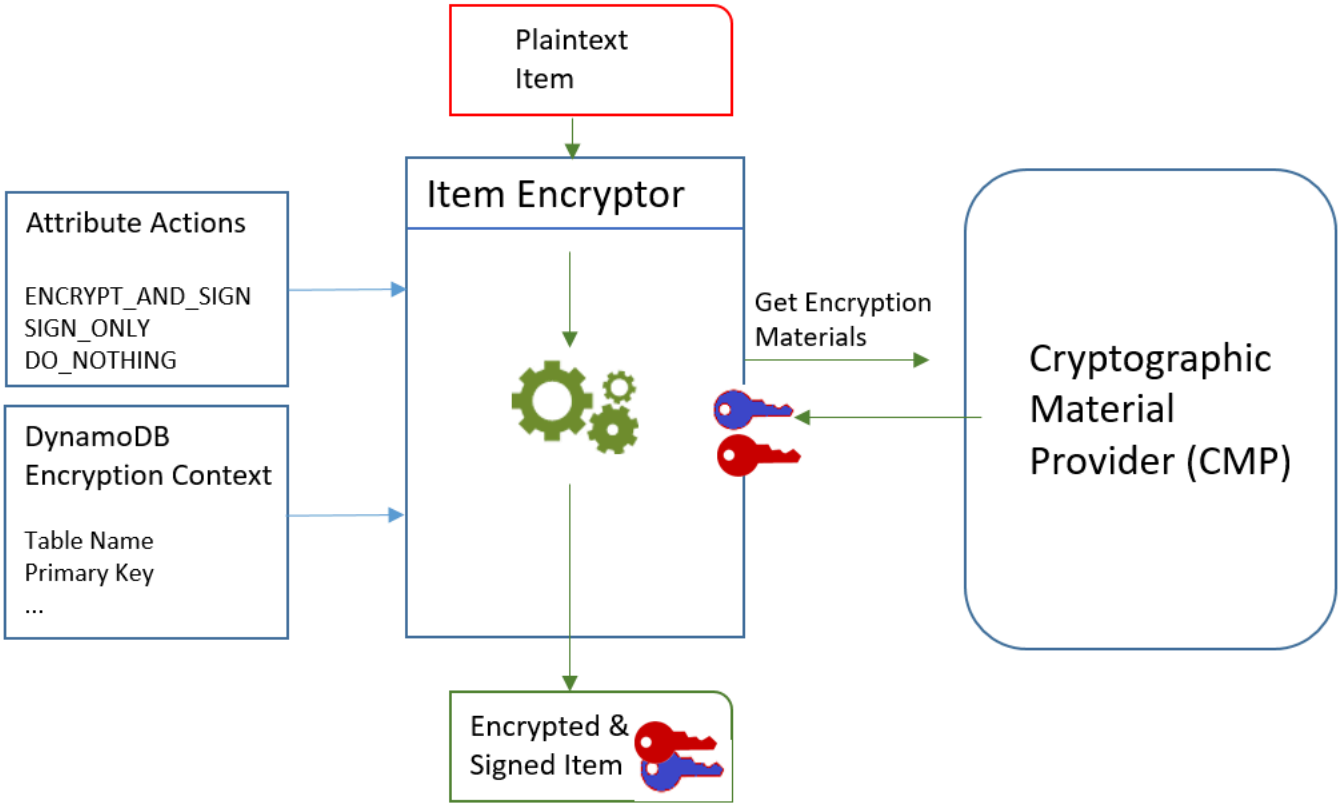
> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x～2.x および DynamoDB Encryption Client for Python のバージョン 1.x～3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

DynamoDB 暗号化クライアントは、DynamoDB に保存されているデータを保護するように特別に設計されています。ライブラリには、拡張が可能でまた変更なしで使用できる安全な実装が含まれています。また、ほとんどの要素は抽象要素で表されるため、互換性のあるカスタムコンポーネントを作成して使用できます。

テーブル項目の暗号化と署名

DynamoDB 暗号化クライアントの中核には、テーブル項目を暗号化、署名、検証、復号する項目エンクリプタがあります。テーブル項目に関する情報と、暗号化して署名する項目に関する指示が取り込まれます。選択して設定した暗号化マテリアルプロバイダーから、暗号化マテリアルとその使用方法に関する指示が取得されます。

次の図は、このプロセスの高レベルのビューを示しています。



テーブル項目を暗号化して署名するには、DynamoDB 暗号化クライアントに次のものが必要です。

- テーブルについての情報。お客様が提供する DynamoDB 暗号化コンテキストからテーブルに関する情報を取得します。一部のヘルパーは、DynamoDB から必要な情報を取得し、DynamoDB 暗号化コンテキストを作成します。

> **Note**
>
> DynamoDB 暗号化クライアントの DynamoDB 暗号化コンテキストは、 AWS Key Management Service (AWS KMS) および の暗号化コンテキストとは関係ありません AWS Encryption SDK。 DynamoDB

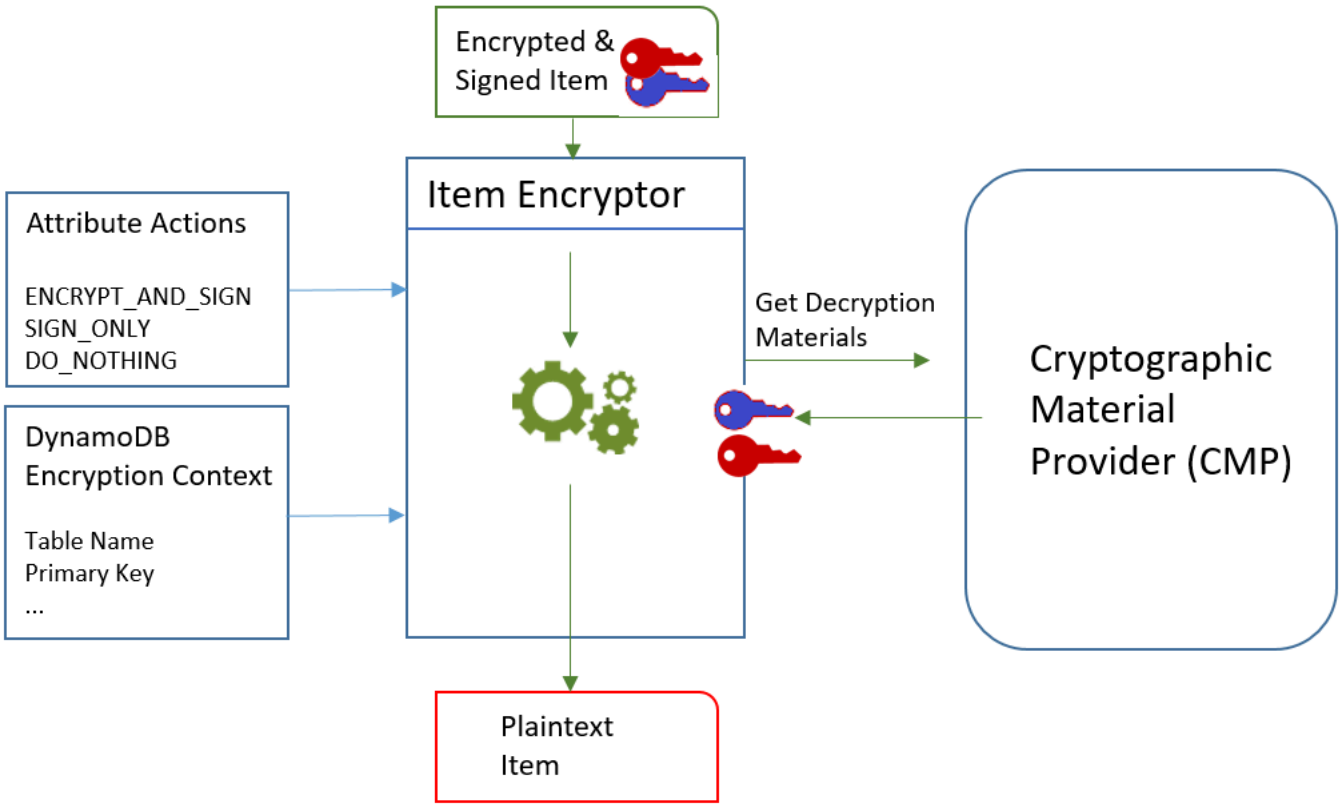
- 暗号化して署名する属性。この情報は、指定した属性アクションから取得されます。
- 暗号化および署名キーを含む、暗号化マテリアル。これらは、お客様が選択して設定する暗号化マテリアルプロバイダー (CMP) から取得されます。
- 項目の暗号化と署名の手順。CMP は、暗号化および署名アルゴリズムを含む、暗号化マテリアルを使用するための指示を実際のマテリアル説明に追加します。

項目エンクリプタは、これらの要素のすべてを使用して項目を暗号化して署名します。項目エンクリプタは、暗号化と署名の指示 (実際のマテリアル説明) を含むマテリアル説明属性と、その署名を含む属性を項目に追加します。項目エンクリプタと直接やり取りすることができます。また、項目エンクリプタとやり取りするヘルパー機能を使用して、安全なデフォルトの動作を実装することもできます。

結果は、暗号化された署名済みデータを含む DynamoDB 項目です。

テーブル項目の検証と復号

これらのコンポーネントは、次の図に示すように、項目を検証および復号するために一緒に機能します。



項目を検証し、復号するためには、DynamoDB 暗号化クライアントには、次のように、同じコンポーネント、同じ設定のコンポーネント、または項目を復号するために特に設計されたコンポーネントが必要です。

- DynamoDB 暗号化コンテキストからのテーブルに関する情報。
- 検証および復号する属性。これらは属性アクションから取得されます。
- 選択し、設定した暗号化マテリアルプロバイダー (CMP) からの検証キーおよび復号キーを含む復号マテリアル。

  暗号化された項目には、暗号化に使用された CMP のレコードは含まれません。同じ CMP、同じ設定の CMP、または項目を復号するように設計された CMP 指定する必要があります。
- 暗号化アルゴリズムと署名アルゴリズムを含む、項目の暗号化と項目の署名に関する情報。クライアントは、項目のマテリアル説明属性からこれらを取得します。

項目エンクリプタは、これらの要素のすべてを使用して項目の検証と復号を行います。また、マテリアル記述と署名属性も削除されます。結果はプレーンテキスト DynamoDB 項目です。

Amazon DynamoDB Encryption Client の概念

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x～2.x および DynamoDB Encryption Client for Python のバージョン 1.x～3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

このトピックでは、Amazon DynamoDB Encryption Client で使用されている概念と用語について説明します。

DynamoDB 暗号化クライアントのコンポーネントがやり取りする方法については、DynamoDB 暗号化クライアントの仕組み を参照してください。

トピック

- 暗号化マテリアルプロバイダー (CMP)
- 項目エンクリプタ
- 属性アクション
- マテリアル記述
- DynamoDB 暗号化コンテキスト
- プロバイダーストア

暗号化マテリアルプロバイダー (CMP)

DynamoDB 暗号化クライアントの実装時に、最初のタスクの 1 つとして、暗号化マテリアルプロバイダー (CMP) (暗号化マテリアルプロバイダーとも呼ばれる) の選択があります。残りの実装の多くは、この選択によって決まります。

暗号化マテリアルプロバイダー (CMP) は項目エンクリプタが、テーブル項目を暗号化し署名するのに使用する暗号化マテリアルを収集、アセンブルし、返します。CMP は、使用する暗号化アルゴリズムと、暗号化キーと署名キーを生成して保護する方法を決定します。

CMP は項目エンクリプタとやり取りします。項目エンクリプタは、暗号化または復号マテリアルを CMP に要求し、CMP はそれを項目エンクリプタに返します。次に、項目エンクリプタは、暗号化マテリアルを使用して、項目の暗号化、署名、検証、および復号を行います。

CMP は、クライアントの設定時に指定します。互換性のあるカスタム CMP を作成するか、ライブラリ内の多くの CMP のいずれかを使用できます。ほとんどの CMP は、複数のプログラミング言語で使用できます。

項目エンクリプタ

項目エンクリプタは、DynamoDB 暗号化クライアントの暗号化オペレーションを実行する低レベルのコンポーネントです。項目エンクリプタは、暗号化マテリアルプロバイダー (CMP) に暗号化マテリアルをリクエストし、CMP より返るマテリアルを使用して、テーブル項目を暗号化して署名するか、検証して復号します。

項目エンクリプタと直接やり取りするか、ライブラリにあるヘルパーを使用することができます。例えば、Java 用 DynamoDB 暗号化クライアントには、`DynamoDBMapper` で使用できる `AttributeEncryptor` ヘルパークラスが含まれています。`DynamoDBEncryptor` 項目エンクリプタとは直接やり取りしません。Python ライブラリには、項目エンクリプタとやり取りする、`EncryptedTable`、`EncryptedClient`、および `EncryptedResource` ヘルパークラスが含まれています。

属性アクション

属性アクションは、項目の各属性に対して実行するアクションを項目エンクリプタに指示します。

属性アクションの値は、次のいずれかの値になります。

- 暗号化と署名 –属性値を暗号化します。項目の署名に属性 (名前と値) を含めます。
- 署名のみ - 項目署名に属性を含めます。
- 何もしない - 属性に対して暗号化と署名のいずれも行いません。

機密データを保存できるすべての属性は、暗号化と署名を使用します。プライマリキー属性 (パーティションキーとソートキー) は、署名のみを使用します。マテリアル説明属性および署名属性は、署名も暗号化もされていません。これらの属性の属性アクションを指定する必要はありません。

属性アクションを慎重に選択します。不確かな場合は、暗号化と署名を使用します。DynamoDB 暗号化クライアントを使用してテーブル項目を保護した後は、署名検証エラーのリスクを冒すことな

く、属性のアクションを変更することはできません。詳細については、「データモデルの変更」を参照してください。

> **Warning**
>
> プライマリキー属性を暗号化しないでください。DynamoDB でテーブル全体のスキャンを実行せずに項目を見つけられるように、プレーンテキストの状態を維持する必要があります。

DynamoDB 暗号化コンテキストがプライマリキー属性を識別する場合、それらを暗号化しようとするとクライアントはエラーをスローします。

属性アクションの指定に使用する手法は、プログラミング言語ごとに異なります。また、使用するヘルパークラスに固有の場合もあります。

詳細については、使用しているプログラミング言語のドキュメントを参照してください。

- Python
- Java

マテリアル記述

暗号化されたテーブル項目のマテリアル説明は、暗号化アルゴリズムなどの情報で構成されます。この情報は、テーブル項目が暗号化および署名される仕組みに関するものです。暗号化マテリアルプロバイダー (CMP) は、暗号化し、署名するための暗号化マテリアルをアセンブルするときに、マテリアル説明を記録します。後で、項目を検証および復号するために暗号化されたマテリアルをアセンブルする必要がある場合は、そのマテリアル記述をガイドとして使用します。

DynamoDB 暗号化クライアントでは、マテリアル記述は 3 つの関連する要素について参照します。

リクエストされたマテリアル説明

暗号化マテリアルプロバイダー (CMP) によっては、暗号化アルゴリズムなどの高度なオプションを指定できます。選択肢を示すために、テーブル項目を暗号化するリクエストの DynamoDB 暗号化コンテキストのマテリアル説明プロパティに名前と値のペアを追加します。この要素は、リクエストされたマテリアル説明と呼ばれます。リクエストされたマテリアル記述の有効値は、選択した CMP によって定義されます。

> **Note**
>
> マテリアル記述は安全なデフォルト値を上書きできるため、やむを得ない理由がない限り、リクエストされたマテリアル記述を省略することをお奨めします。

実際のマテリアル記述

暗号化マテリアルプロバイダー (CMP) が返すマテリアル説明は、実際のマテリアル説明と呼ばれます。CMP が暗号化マテリアルを構築したときに使用した実際の値について説明します。また、通常、リクエストされたマテリアル記述で構成され、ある場合は追加と変更を含みます。

マテリアル記述属性

クライアントは、実際のマテリアル説明を暗号化項目のマテリアル説明属性に保存します。このマテリアル記述属性名は、`amzn-ddb-map-desc` で、その値は実際のマテリアル記述です。クライアントは、マテリアル記述属性の値を使用して、項目の検証および復号を行います。

DynamoDB 暗号化コンテキスト

DynamoDB 暗号化コンテキストは、テーブルと項目に関する情報を暗号化マテリアルプロバイダー (CMP) に提供します。高度な実装では、DynamoDB 暗号化コンテキストに、リクエストされたマテリアルの説明を含めることができます。

テーブル項目を暗号化すると、DynamoDB 暗号化コンテキストが暗号化された属性値に暗号でバインドされます。復号時に、DynamoDB 暗号化コンテキストが暗号化に使用された DynamoDB 暗号化コンテキストに対して大文字と小文字を区別して完全に一致しない場合、復号オペレーションは失敗します。項目エンクリプタと直接やり取りする場合は、暗号化メソッドまたは復号メソッドを呼び出すときに DynamoDB 暗号化コンテキストを提供する必要があります。ほとんどのヘルパーは、DynamoDB 暗号化コンテキストを作成します。

> **Note**
>
> DynamoDB 暗号化クライアントの DynamoDB 暗号化コンテキストは、 AWS Key Management Service (AWS KMS) および の暗号化コンテキストとは関係ありません AWS Encryption SDK。 DynamoDB

DynamoDB 暗号化のコンテキストによって次のフィールドを含めることができます。すべてのフィールドと値はオプションです。

- テーブル名
- パーティションキー名
- ソートキー名
- 属性名と値のペア
- リクエストされたマテリアル説明

プロバイダーストア

プロバイダーストアは、暗号化マテリアルプロバイダー (CMP) を返すコンポーネントです。プロバイダーストアは、CMP を作成するか、別のプロバイダーストアなどの別のソースから CMP を取得できます。プロバイダーストアは、作成した CMP のバージョンを、保存されたそれぞれの CMP がリクエスタのマテリアル名とバージョン番号によって識別される永続的ストレージに保存します。

DynamoDB 暗号化クライアントの最新プロバイダーはプロバイダーストアから CMP を取得しますが、プロバイダーストアを使用して任意のコンポーネントに CMP を提供できます。各最新のプロバイダーは 1 つのプロバイダーストアに関連付けられていますが、プロバイダーストアは複数のホスト間で多くのリクエスタに CMP を提供できます。

プロバイダーストアは、オンデマンドで新しいバージョンの CMP を作成し、新しいバージョンと既存のバージョンを返します。また、指定されたマテリアル名の最新バージョン番号も返されます。これにより、リクエスタは、プロバイダーストアからリクエストできる新しいバージョンの CMP がリリースされるタイミングを把握することができます。

DynamoDB 暗号化クライアントには MetaStore が含まれています。これは、DynamoDB に保管され、内部 DynamoDB 暗号化クライアントを使用して暗号化されたキーを使用してラップされた CMP を作成するプロバイダーストアです。

詳細はこちら:

- プロバイダーストア: Java、Python
- MetaStore: Java、Python

暗号マテリアルプロバイダー

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x～2.x および DynamoDB Encryption Client for Python のバージョン 1.x～3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

DynamoDB 暗号化クライアントを使用する場合に最も重要となる決定事項の 1 つは、暗号化マテリアルプロバイダー (CMP) の選択です。CMP は、暗号化マテリアルをアセンブルして、項目エンクリプタに返します。また、暗号化キーと署名キーの生成方法、新しいキーマテリアルが項目ごとに生成されるか、または再利用されるか、使用する暗号化アルゴリズムおよび署名アルゴリズムも指定されます。

DynamoDB 暗号化クライアントライブラリに含まれている実装から CMP を選択するか、互換性のあるカスタム CMP を構築できます。また、CMP の選択も、使用するプログラミング言語によって異なります。

このトピックでは、一般的な CMP について説明するとともに、アプリケーションに最適な CMP を選択するのに役立ついくつかのアドバイスを提供します。

Direct KMS マテリアルプロバイダー

Direct KMS マテリアルプロバイダーは、AWS KMS key によってテーブル項目を保護しているため、AWS Key Management Service (AWS KMS) は必ず暗号化されます。アプリケーションで、暗号化マテリアルを生成または管理する必要はありません。を使用して項目ごとに一意の暗号化キーと署名キー AWS KMS key を生成するため、このプロバイダーは項目を暗号化または復号する AWS KMS たびに を呼び出します。

を使用し AWS KMS 、トランザクションごとに 1 回の AWS KMS 呼び出しがアプリケーションで実用的である場合、このプロバイダーが適しています。

詳細については、「Direct KMS マテリアルプロバイダー」を参照してください。

ラップされたマテリアルプロバイダー (ラップされた CMP)

ラップされたマテリアルプロバイダー (ラップされた CMP) では、DynamoDB 暗号化クライアントの外部で、ラッピングおよび署名キーを生成および管理することができます。

ラップされた CMP は、項目ごとに一意の暗号化キーを生成します。次に、生成したラップキー (またはアンラップキー) および署名キーを使用します。したがって、ラップキーおよび署名キーの生成方法と、それらが各項目に一意か、または再利用されたものかを判断します。ラップされた CMP は、 を使用せず、暗号化マテリアルを安全に管理 AWS KMS できるアプリケーション用の Direct KMS プロバイダーの安全な代替手段です。

詳細については、「ラップされたマテリアルプロバイダー」を参照してください。

最新プロバイダー

最新プロバイダーは、プロバイダーストアで機能するように設計された暗号化マテリアルプロバイダー (CMP) です。プロバイダーストアから CMP を取得し、CMP から返る暗号化マテリアルを取得します。最新プロバイダーでは通常、各 CMP を使用して暗号化マテリアルの複数の要求を満たしますが、プロバイダーストアの機能を使用して、マテリアルの再利用範囲を制御したり、CMP の回転頻度を判断したりできるほか、最新プロバイダーを変更せずに使用される CMP のタイプを変更することもできます。

最新プロバイダーは互換性のあるプロバイダーストアで使用できます。DynamoDB 暗号化クライアントには、ラップされた CMP を返すプロバイダーストアである MetaStore が含まれています。

最新プロバイダーは、その暗号ソースへの呼び出しを最小限に抑える必要のあるアプリケーションや、セキュリティ要件に違反せずに一部の暗号化マテリアルを再利用できるアプリケーションに適しています。たとえば、項目を暗号化または復号する AWS KMS たびに を呼び出すことなく、 AWS KMS key AWS Key Management Service (AWS KMS) の で暗号化マテリアルを保護できます。

詳細については、「最新プロバイダー」を参照してください。

静的マテリアルプロバイダー

静的マテリアルプロバイダーは、検証や概念実証のデモンストレーション、および従来の互換性を目的として設計されています。項目ごとに一意の暗号化マテリアルが生成されることはありません。指定した暗号化キーと署名キーが返ります。これらのキーは、テーブル項目の暗号化、復号、および署名に直接使用されます。

> **Note**
>
> Java ライブラリ内の非対称静的プロバイダーは静的プロバイダーではありません。これは、ラップされた CMP の代替コンストラクタを指定するだけです。本稼働環境での使用は安全ですが、できるだけラップされた CMP を直接使用する必要があります。

トピック

Direct KMS マテリアルプロバイダー

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

Direct KMS マテリアルプロバイダー (Direct KMS プロバイダー) は、AWS KMS key によってテーブル項目を保護しているため、AWS Key Management Service (AWS KMS) は必ず暗号化されます。この暗号化マテリアルプロバイダーより、テーブル項目ごとに一意の暗号化キーと署名キーが返ります。そのためには、項目を暗号化または復号する AWS KMS たびに を呼び出します。

DynamoDB 項目を高頻度かつ大規模に処理している場合、1 AWS KMS requests-per-second数の制限を超えると、処理の遅延が発生する可能性があります。制限を超過する必要がある場合は、AWS サポート センターでケースを作成してください。また、最新プロバイダーなど、キーの再利用が制限された暗号化マテリアルプロバイダーの使用を検討することもできます。

Direct KMS プロバイダーを使用するには、発信者に、 AWS アカウントで GenerateDataKey および Decrypt オペレーションを呼び出すための 、少なくとも 1 つの AWS KMS key、および アクセス許可が必要です AWS KMS key。 AWS KMS key は対称暗号化キーである必要があります。DynamoDB 暗号化クライアントは非対称暗号化をサポートしていません。DynamoDB グローバルテーブルを使用している場合、AWS KMS マルチリージョンキーを指定することもできます。詳細については、「使用方法」を参照してください。

> **Note**
>
> Direct KMS プロバイダーを使用すると、プライマリキー属性の名前と値は、関連する AWS KMS オペレーションのAWS KMS 暗号化コンテキストと AWS CloudTrail ログにプレーンテキストで表示されます。ただし、DynamoDB 暗号化クライアントが、暗号化された属性値をプレーンテキストで公開することはありません。

Direct KMS プロバイダーは、DynamoDB 暗号化クライアントがサポートしている複数の暗号化マテリアルプロバイダー (CMP) の 1 つです。他の CMP の詳細については、「暗号マテリアルプロバイダー」を参照してください。

サンプルコードについては、以下を参照してください。

- Java: AwsKmsEncryptedItem
- Python: aws-kms-encrypted-table、aws-kms-encrypted-item

トピック

- 使用方法
- 仕組み

使用方法

Direct KMS プロバイダーを作成するには、キー ID パラメータを使用して、アカウントに対称暗号化 KMS キーを指定します。キー ID パラメータの値は、キー ID、キー ARN、エイリアス名、または AWS KMS keyのエイリアス ARN にすることができます。キー ID の詳細については、AWS Key Management Service デベロッパーガイドの「キー識別子」を参照してください。

Direct KMS プロバイダーでは、対称暗号化 KMS キーが必要です。非対称 KMS キーを使用することはできません。ただし、マルチリージョン KMS キー、インポートされたキーマテリアルを含む KMS キー、またはカスタムキーストア内の KMS キーを使用できます。KMS キーに kms:GenerateDataKey アクセス許可と kms:Decrypt アクセス許可がある必要があります。そのため、 マネージドまたは AWS 所有の KMS キーではなく、カスタマー AWS マネージドキーを使用する必要があります。

DynamoDB Encryption Client for Python は、キー ID パラメータ値にリージョンが含まれている場合、そのリージョン AWS KMS から呼び出すリージョンを決定します。それ以外の場合は、 AWS KMS クライアントでリージョンを指定するか、 で設定したリージョンを使用します AWS SDK for

Python (Boto3)。Python でのリージョンの選択については、 AWS SDK for Python (Boto3) API リファレンスの「設定」を参照してください。

Java 用 DynamoDB 暗号化クライアントは、指定したクライアントにリージョンが含まれている場合、クライアントのリージョン AWS KMS AWS KMS から呼び出すリージョンを決定します。リージョンが含まれていない場合、 AWS SDK for Javaで設定されたリージョンが使用されます。でのリージョンの選択については AWS SDK for Java、「 AWS SDK for Java デベロッパーガイド」のAWS リージョン 「選択」を参照してください。

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

次の例では、キー ARN を使用して AWS KMS keyを指定しています。キー識別子に が含まれていない場合 AWS リージョン、DynamoDB 暗号化クライアントは、設定された Botocore セッションがある場合、または Boto のデフォルトからリージョンを取得します。

```
# 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 グローバルテーブルを使用している場合は、 AWS KMS マルチリージョンキーでデータを暗号化することをお勧めします。マルチリージョンキーは異なる AWS KMS keys にあり AWS リージョン 、同じキー ID とキーマテリアルを持つため、同じ意味で使用できます。詳細については、AWS Key Management Service デベロッパーガイドの「マルチリージョンキーを使用する」を参照してください。

> **Note**
>
> グローバルテーブルのバージョン 2017.11.29 を使用している場合は、予約されたレプリケーションフィールドが暗号化または署名されないように属性アクションを設定する必要が

あります。詳細については、「古いバージョンのグローバルテーブルの問題」を参照してください。

DynamoDB 暗号化クライアントでマルチリージョンキーを使用するには、マルチリージョンキーを作成し、アプリケーションを実行するリージョンにレプリケートします。次に、DynamoDB 暗号化クライアントが AWS KMSを呼び出すリージョンでマルチリージョンキーを使用するように Direct KMS プロバイダーを設定します。

次の例では、マルチリージョンキーを使用して、米国東部 (バージニア北部) (us-east-1) リージョンのデータを暗号化し、米国西部 (オレゴン) (us-west-2) リージョンのデータを復号するように DynamoDB 暗号化クライアントを設定します。

Java

この例では、DynamoDB 暗号化クライアントは、 AWS KMS クライアントの リージョン AWS KMS から を呼び出すための リージョンを取得します。keyArn 値は、同じリージョンのマルチリージョンキーを識別します。

```
// 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

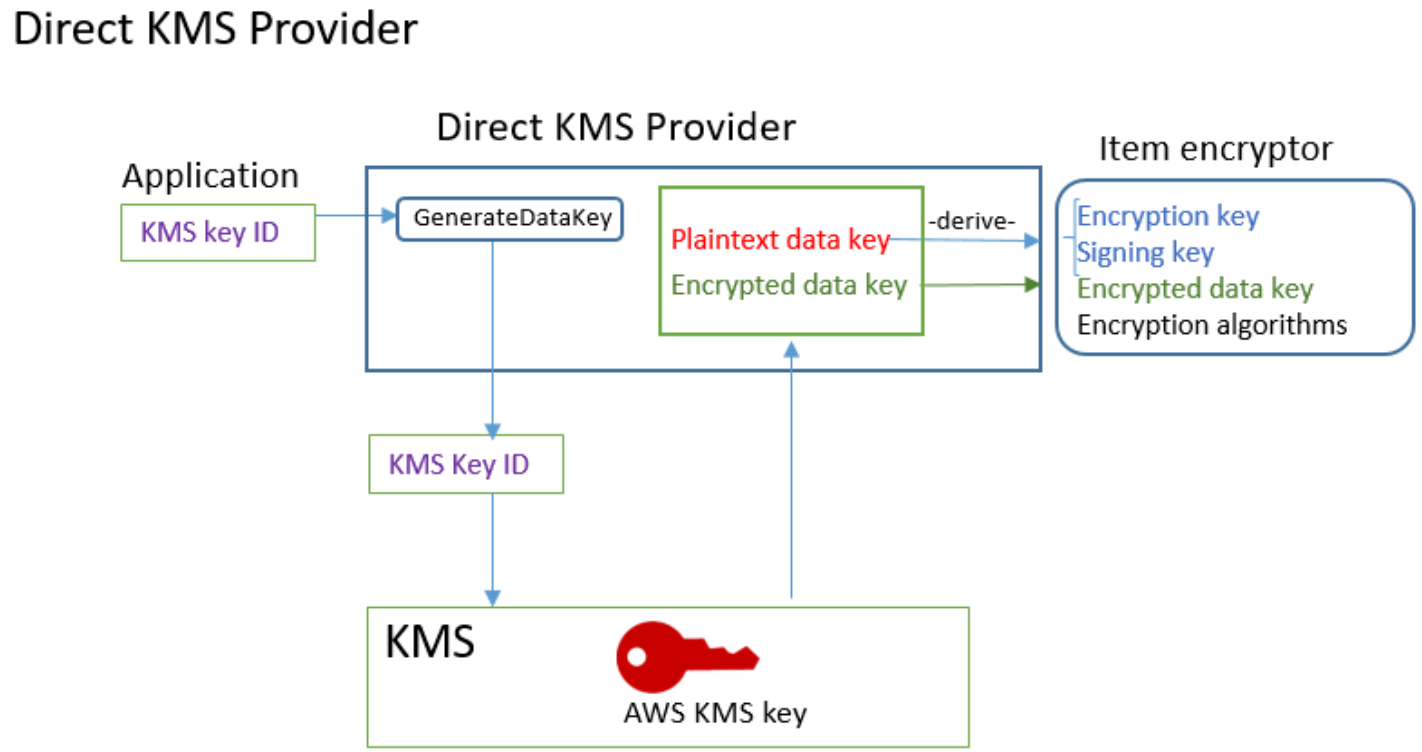
この例では、DynamoDB 暗号化クライアントは、キー ARN のリージョン AWS KMS から を呼び出すためのリージョンを取得します。

```
# 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)
```

仕組み

以下の図に示されているように、Direct KMS プロバイダーは、指定した AWS KMS key で保護されている暗号化キーおよび署名キーを返します。



- 暗号化マテリアルを生成するために、Direct KMS プロバイダーは、指定した を使用して項目ごとに一意のデータキーを生成する AWS KMS ように に求め AWS KMS key ます。これにより、データキーのプレーンテキストコピーから項目の暗号化キーと署名キーが導出され、暗号化データキーと一緒に返ります。このデータキーは、項目のマテリアル記述属性に保存されます。

  項目エンクリプタでは、この暗号化キーおよび署名キーを使用します。また、メモリから可能な限り早くそれらを削除します。導出されたデータキーの暗号化されたコピーのみ、暗号化された項目に保存されます。

- 復号マテリアルを生成するために、Direct KMS プロバイダーは暗号化されたデータキーを復号 AWS KMS するように に求めます。これにより、プレーンテキストデータキーより検証キーおよび署名キーが導出され、項目エンクリプタに返されます。

  項目エンクリプタは項目を検証し、検証が成功すると、暗号化された値が復号されます。次に、可能な限り早く、メモリよりキーが削除されます。

暗号化マテリアルを取得する

このセクションでは、項目エンクリプタより暗号化マテリアルのリクエストを受け取るときの Direct KMS プロバイダーの入力、出力、処理の詳細について説明します。

入力 (アプリケーションから)

- のキー ID AWS KMS key。

入力 (項目エンクリプタから)

- DynamoDB 暗号化コンテキスト

出力 (項目エンクリプタへ)

- 暗号化キー (プレーンテキスト)
- 署名キー
- 実際のマテリアル説明で、これらの値は、クライアントより項目に追加されるマテリアル説明属性に保存されます。
  - amzn-ddb-env-key: によって暗号化された Base64-encodedされたデータキー AWS KMS key
  - amzn-ddb-env-alg: 暗号化アルゴリズム。デフォルトは AES/256
  - amzn-ddb-sig-alg: 署名アルゴリズム。デフォルトは HmacSHA256/256
  - amzn-ddb-wrap-alg: kms

Processing

1. Direct KMS プロバイダー AWS KMS は、指定された を使用して項目の一意のデータキー AWS KMS key を生成するリクエストを送信します。 https://docs.aws.amazon.com/kms/latest/APIReference/API_GenerateDataKey.htmlこのオペレーションによって、プレーンテキストキーと、 AWS KMS keyで暗号化されたコピーが返ります。これは、初期のキーマテリアルと呼ばれます。

   このリクエストの AWS KMS 暗号化テキストには、次のプレーンテキスト形式の値が含まれています。これらのシークレットではない値は、暗号化されたオブジェクトに暗号的にバインドされているため、復号時には同じ暗号化コンテキストが必要です。これらの値を使用して、AWS CloudTrail 口グ AWS KMS で への呼び出しを識別できます。

   - amzn-ddb-env-alg - 暗号化アルゴリズム。デフォルトは AES/256
   - amzn-ddb-sig-alg - 署名アルゴリズム。デフォルトは HmacSHA256/256
   - (オプション) aws-kms-table – *#####*
   - (オプション) *##########* – *##########* (バイナリ値は Base64 エンコード形式)

- (オプション) ###### – ###### (バイナリ値は Base64 エンコード形式)

  Direct KMS プロバイダーは、項目の DynamoDB AWS KMS 暗号化コンテキストから暗号化コンテキストの値を取得します。 DynamoDB DynamoDB 暗号化コンテキストにテーブル名などの値が含まれていない場合、その名前と値のペアは AWS KMS 暗号化コンテキストから省略されます。

2. Direct KMS プロバイダーは、対称暗号化キーおよび署名キーをデータキーから導出します。デフォルトでは、セキュアハッシュアルゴリズム (SHA) 256 および RFC5869 HMAC ベースのキー導出関数を使用して、256 ビット AES 対称暗号化キーおよび 256 ビット HMAC-SHA-256 署名キーを導出します。
3. Direct KMS プロバイダーは、項目エンクリプタに出力を返します。
4. 項目エンクリプタは、暗号化キーを使用して、指定された属性を暗号化し、署名キーを使用して署名します。この際、実際のマテリアル記述で指定されたアルゴリズムを使用します。可能な限り早く、メモリよりプレーンテキストキーが削除されます。

復号マテリアルを取得する

このセクションでは、項目エンクリプタより復号マテリアルのリクエストを受け取るときの Direct KMS プロバイダーの入力、出力、処理の詳細について説明します。

入力 (アプリケーションから)

- のキー ID AWS KMS key。

  キー ID の値は、キー ID、キー ARN、エイリアス名、または AWS KMS keyのエイリアス ARN にすることができます。キー ID に含まれていない値 (リージョンなど) はすべて、AWS 名前付きプロファイルで入手できる必要があります。キー ARN により、 AWS KMS で必要なすべての値が提供されます。

入力 (項目エンクリプタから)

- DynamoDB 暗号化コンテキストのコピー (マテリアル説明属性の内容を含む)。

出力 (項目エンクリプタへ)

- 暗号化キー (プレーンテキスト)
- 署名キー

Processing

1. Direct KMS プロバイダーは、暗号化された項目のマテリアル記述属性から暗号化されたデータキーを取得します。
2. は、指定された AWS KMS key を使用して暗号化されたデータキーを復号 AWS KMS するよう に要求します。オペレーションでプレーンテキストのキーが返ります。

   このリクエストでは、データキーの生成および暗号化に使用したのと同じ AWS KMS 暗号化コンテキストを使用する必要があります。

   - aws-kms-table – *#####*
   - *##########* – *##########* (バイナリ値は Base64 エンコード形式)
   - (オプション) *######* – *######* (バイナリ値は Base64 エンコード形式)
   - amzn-ddb-env-alg - 暗号化アルゴリズム。デフォルトは AES/256
   - amzn-ddb-sig-alg - 署名アルゴリズム。デフォルトは HmacSHA256/256
3. Direct KMS プロバイダーでは、セキュアハッシュアルゴリズム (SHA) 256 および RFC5869 HMAC ベースのキー導出関数を使用して、データキーから 256 ビット AES 対称暗号化キーおよび 256 ビット HMAC-SHA-256 署名キーを導出します。
4. Direct KMS プロバイダーは、項目エンクリプタに出力を返します。
5. 項目エンクリプタは、署名キーを使用して項目を検証します。成功すると、暗号化された属性値は対称暗号化キーを使用して復号されます。これらのオペレーションでは、実際のマテリアル記述で指定された暗号化アルゴリズムおよび署名アルゴリズムが使用されます。項目エンクリプタによって、可能な限り早く、メモリよりプレーンテキストキーが削除されます。

ラップされたマテリアルプロバイダー

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

ラップされたマテリアルプロバイダー (ラップされた CMP) では、DynamoDB 暗号化クライアントを使用して任意のソースからラッピングおよび署名キーを使用できます。ラップされた CMP はどの AWS サービスにも依存しません。ただし、クライアントの外部にあるラップキーと署名キーを生成して管理する必要があります。これには、項目を検証および復号するための正しいキーを提供することが含まれます。

ラップされた CMP は、項目ごとに固有の項目暗号化キーを生成します。項目暗号化キーを指定したラップキーでラップし、ラップされた項目暗号化キーを項目のマテリアル説明属性に保存します。ラップキーと署名キーを指定するため、ラップキーと署名キーの生成方法と、それらが各項目に固有のものか再利用されたものかを判断します。

ラップされた CMP は、安全な実装であり、暗号化マテリアルを管理できるアプリケーションに適しています。

ラップされた CMP は、DynamoDB 暗号化クライアントがサポートしている複数の暗号化マテリアルプロバイダー (CMP) の 1 つです。他の CMP の詳細については、「暗号マテリアルプロバイダー」を参照してください。

サンプルコードについては、以下を参照してください。

- Java: AsymmetricEncryptedItem
- Python: wrapped-rsa-encrypted-table、wrapped-symmetric-encrypted-table

トピック

- 使用方法
- 仕組み

使用方法

ラップされた CMP を作成するには、ラップキー (暗号化に必要)、ラップ解除キー (復号に必要)、および署名キーを指定します。項目を暗号化および復号するときには、キーを指定する必要があります。

ラップキー、ラップ解除キー、および署名キーは、対称キーまたは非対称キーペアにすることができます。

Java

```
// This example uses asymmetric wrapping and signing key pairs
```

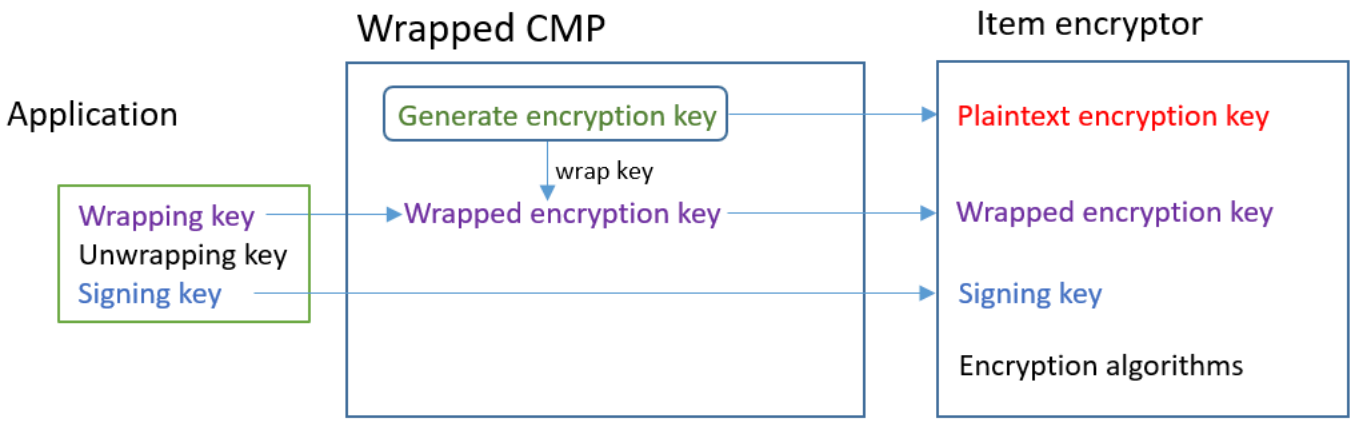
```
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

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
)
```

仕組み

ラップされた CMP は、すべての項目に新しい項目暗号化キーを生成します。次の図に示すように、ラップキー、ラップ解除キー、および署名キーを使用します。



暗号化マテリアルを取得する

このセクションでは、暗号化マテリアルのリクエストを受け取る際のラップされたマテリアルプロバイダー (ラップされた CMP) の入力、出力、処理の詳細について説明します。

入力 (アプリケーションから)

- ラップされたキー: Advanced Encryption Standard (AES) 対称キー、または RSA パブリックキー。属性値が暗号化されている場合は必須です。それ以外の場合はオプションであり、無視されます。
- ラップ解除キー: オプションで無視されます。
- 署名キー

入力 (項目エンクリプタから)

- DynamoDB 暗号化コンテキスト

出力 (項目エンクリプタへ):

- プレーンテキスト項目暗号化キー
- 署名キー (変更されません)
- 実際のマテリアル説明: これらの値は、クライアントが項目に追加するマテリアル説明属性に保存されます。
  - `amzn-ddb-env-key`: Base64 でエンコードされたラップされた項目暗号化キー
  - `amzn-ddb-env-alg`: 項目を暗号化するために使用される暗号化アルゴリズム。デフォルトは AES-256-CBC です。
  - `amzn-ddb-wrap-alg`: ラップされた CMP が項目暗号化キーをラップするために使用したラップアルゴリズム。ラッピングキーが AES キーの場合、RFC 3394 で定義されているように、キーは埋め込みなしの `AES-Keywrap` を使用してラップされます。ラップキーが RSA キーの場合、キーは MGF1 パディング付き RSA OAEP を使用して暗号化されます。

Processing

項目を暗号化する際は、ラップキーと署名キーで渡します。ラップ解除キーは、オプションで無視されます。

1. ラップされた CMP は、テーブル項目に固有の対称項目暗号化キーを生成します。
2. 項目暗号化キーをラップするために指定したラップキーを使用します。次に、可能な限り早く、メモリより削除されます。

3. これは、プレーンテキスト項目暗号化キー、指定した署名キー、実際のマテリアル説明 (ラップされた項目暗号化キー、暗号化およびラップアルゴリズムを含む) を返します。
4. 項目エンクリプタは、プレーンテキスト暗号化キーを使用して項目を暗号化します。項目に署名するために指定した署名キーを使用します。次に、可能な限り早く、メモリよりプレーンテキストキーが削除されます。ラップされた暗号化キー (amzn-ddb-env-key) を含む、実際のマテリアル記述のフィールドを項目のマテリアル記述属性にコピーします。

復号マテリアルを取得する

このセクションでは、復号マテリアルのリクエストを受け取る際のラップされたマテリアルプロバイダー (ラップされた CMP) の入力、出力、処理の詳細について説明します。

入力 (アプリケーションから)

- ラップキー: オプションで無視されます。
- ラップ解除キー: 同じ Advanced Encryption Standard (AES) 対称キーまたは RSA 暗号化に使用された RSA パブリックキーに対応するプライベートキー。属性値が暗号化されている場合は必須です。それ以外の場合はオプションであり、無視されます。
- 署名キー

入力 (項目エンクリプタから)

- DynamoDB 暗号化コンテキストのコピー (マテリアル説明属性の内容を含む)。

出力 (項目エンクリプタへ)

- プレーンテキスト項目暗号化キー
- 署名キー (変更されません)

Processing

項目を復号する際は、ラップ解除キーと署名キーで渡します。ラップキーは、オプションで無視されます。

1. ラップされた CMP は、項目のマテリアル記述属性からラップされた項目暗号化キーを取得します。
2. 項目暗号化キーをラップ解除するためにラップ解除キーとアルゴリズムを使用します。

3. それは、項目エンクリプタにプレーンテキスト項目暗号化キー、署名キー、および暗号化および署名アルゴリズムを返します。
4. 項目エンクリプタは、署名キーを使用して項目を検証します。成功すると、項目暗号化キーを使用して項目を復号します。次に、可能な限り早く、メモリよりプレーンテキストキーが削除されます。

最新プロバイダー

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x～2.x および DynamoDB Encryption Client for Python のバージョン 1.x～3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

最新プロバイダーは、プロバイダーストアで機能するように設計された暗号化マテリアルプロバイダー (CMP) です。プロバイダーストアから CMP を取得し、CMP から返る暗号化マテリアルを取得します。これは、通常、各 CMP を使用して複数の暗号化マテリアルをリクエストします。ただし、プロバイダーストアの機能を使用して、マテリアルが再利用される範囲を制御し、CMP のローテーション頻度を決定し、最新プロバイダーを変更せずに使用する CMP のタイプを変更することもできます。

> **Note**
>
> 最新プロバイダーの `MostRecentProvider` 記号に関連付けられたコードは、プロセスの有効期間の間、暗号化マテリアルをメモリに保存する場合があります。これにより、呼び出し元は、使用する権限がなくなったキーを使用できるようになる可能性があります。`MostRecentProvider` 記号は、DynamoDB 暗号化クライアントのサポートされている古いバージョンでは廃止されており、バージョン 2.0.0 から除去されています。これは、`CachingMostRecentProvider` 記号に置き換えられています。詳細については、「最新プロバイダーの更新」を参照してください。

最新プロバイダーは、プロバイダーストアとその暗号ソースへの呼び出しを最小限に抑える必要のあるアプリケーションや、セキュリティ要件に違反せずに一部の暗号化マテリアルを再利用できるアプ

リケーションに適しています。たとえば、項目を暗号化または復号する AWS KMS たびに を呼び出すことなく、 AWS KMS key AWS Key Management Service (AWS KMS) の で暗号化マテリアルを保護できます。

選択したプロバイダーストアによって、最新プロバイダーが使用する CMP のタイプと、新しい CMP を取得する頻度が決まります。設計したカスタムプロバイダーストアを含む、最新プロバイダーと互換性のある任意のプロバイダーストアを使用できます。

DynamoDB 暗号化クライアントには、ラップされたマテリアルプロバイダー (ラップされた CMP) を作成して返す MetaStore が含まれています。MetaStore は、生成したラップされた CMP の複数のバージョンを内部の DynamoDB テーブルに保存し、DynamoDB 暗号化クライアントの内部インスタンスによるクライアント側の暗号化でそれらを保護します。

任意のタイプの内部 CMP を使用するように MetaStore を設定して、 によって保護された暗号化マテリアルを生成する Direct KMS プロバイダー AWS KMS key、指定したラッピングキーと署名キーを使用するラップされた CMP、または設計した互換性のあるカスタム CMP など、テーブル内のマテリアルを保護できます。

サンプルコードについては、以下を参照してください。

- Java: MostRecentEncryptedItem
- Python: most_recent_provider_encrypted_table

トピック

- 使用方法
- 仕組み
- 最新プロバイダーの更新

使用方法

最新プロバイダーを作成するには、プロバイダーストアを作成して構成した後、プロバイダーストアを使用する最新プロバイダーを作成する必要があります。

次の例は、MetaStore を使用し、Direct KMS プロバイダーから暗号化マテリアルを含む内部 DynamoDB テーブルのバージョンを保護する最新プロバイダーを作成する方法を示しています。以下の例では、`CachingMostRecentProvider` 記号を使用します。

それぞれの最新プロバイダーには、MetaStore テーブル内の CMP、有効期限 (TTL) 設定、およびキャッシュが保持できるエントリの数を決定するキャッシュサイズ設定を特定する名前が付けられ

ています。これらの例では、キャッシュサイズを 1000 エントリに設定し、TTL を 60 秒に設定します。

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
)
```

仕組み

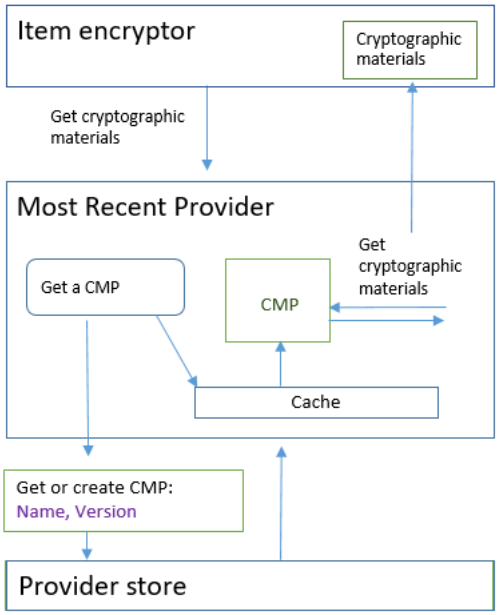
最新プロバイダーがプロバイダーストアから CMP を取得します。次に、CMP を使用して、暗号化マテリアルを生成し、それを項目エンクリプタに返します。

最新プロバイダーについて

最新プロバイダーは、プロバイダーストアから暗号化マテリアルプロバイダー (CMP) を取得します。次に、CMP を使用して、それが返す暗号化マテリアルを生成します。各最新プロバイダーは 1 つのプロバイダーストアに関連付けられていますが、プロバイダーストアは複数のホスト間で複数のプロバイダーに CMP を提供できます。

最新プロバイダーは、任意のプロバイダーストアから互換性のある CMP を使用できます。暗号化または復号マテリアルを CMP に要求し、項目エンクリプタに出力を返します。暗号化オペレーションは実行されません。

最新プロバイダーは、そのプロバイダーストアから CMP を要求するために、使用する既存の CMP のマテリアル名とバージョンを提供します。暗号化マテリアルでは、最新プロバイダーは常に最大 (「最新の」) バージョンをリクエストします。復号マテリアルの場合、次の図に示すように、暗号化マテリアルの作成に使用された CMP のバージョンをリクエストします。



最新プロバイダーは、プロバイダーストアが返す CMP のバージョンをメモリ内のローカル最小使用 (LRU) キャッシュに保存します。キャッシュにより、最新プロバイダーは、すべての項目のプロバイダーストアを呼び出さずに必要な CMP を取得できます。必要に応じてキャッシュをクリアすることができます。

最新プロバイダーは、アプリケーションの特性に基づいて調整できる設定可能な有効期限 (TTL) 値を使用します。

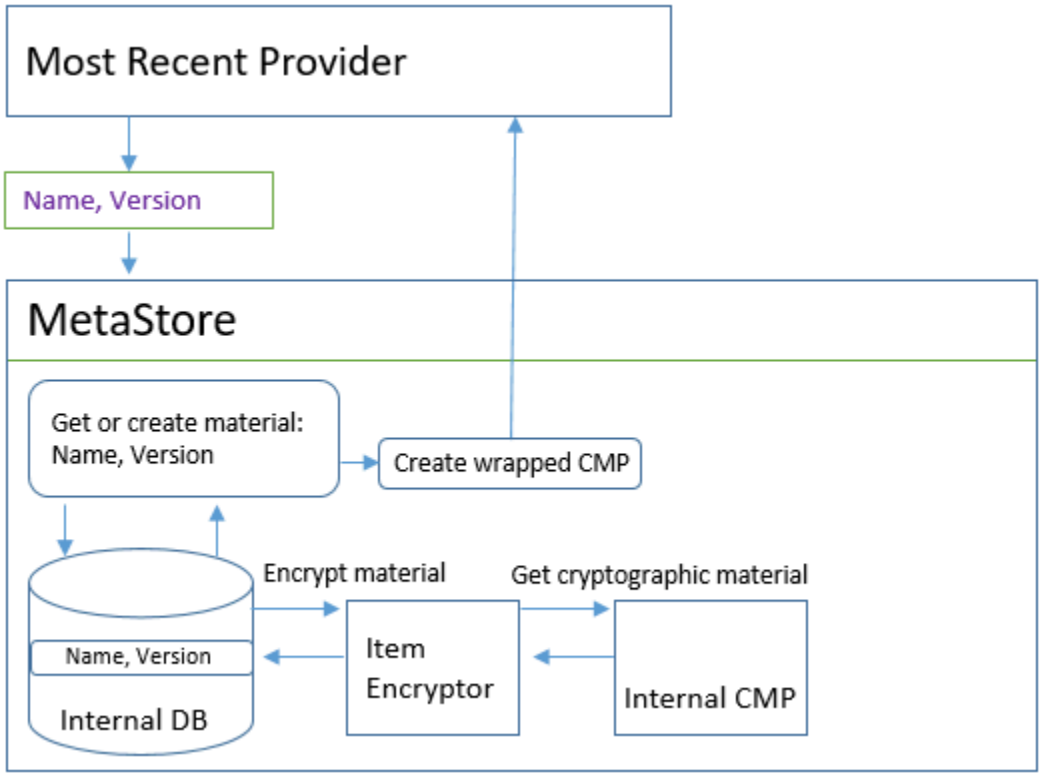
MetaStore について

互換性のあるカスタムプロバイダーストアなどの任意のプロバイダーストアで最新プロバイダーを使用できます。DynamoDB 暗号化クライアントには、設定してカスタマイズできる安全な実装である MetaStore が含まれています。

MetaStore は、CMP で必要なラッピングキー、ラップ解除キー、および署名キーで構成された、ラップされた CMP を作成して返すプロバイダーストアです。ラップされた CMP は常にすべて

の項目に対して一意の項目暗号化キーを生成するため、MetaStore は最新プロバイダーにとって安全なオプションです。項目暗号化キーと署名キーを保護するラップキーのみが再利用されます。

次の図は、MetaStore のコンポーネントと、最新プロバイダーとのやり取りの方法を示しています。



MetaStore はラップされた CMP を生成し、内部 DynamoDB テーブルに (暗号化された形式で) 保存します。パーティションキーは、最新プロバイダーマテリアルの名前であり、ソートキーはそのバージョン番号です。テーブル内のマテリアルは、項目エンクリプタや内部暗号化マテリアルプロバイダー (CMP) など、内部 DynamoDB 暗号化クライアントによって保護されています。

MetaStore では、Direct KMS プロバイダー、提供する暗号化マテリアルを使用したラップされた CMP、または互換性のあるカスタム CMP など、あらゆるタイプの内部 CMP を使うことができます。MetaStore の内部 CMP が Direct KMS プロバイダーの場合、再利用可能なラッピングおよび署名キーは、AWS Key Management Service (AWS KMS) 内の AWS KMS key で保護されます。MetaStore は、内部テーブルに新しい CMP バージョンを追加するか、内部テーブルから CMP バージョンを取得する AWS KMS たびに を呼び出します。

有効期限 (TTL) の値を設定する

作成した最新プロバイダーごとに有効期限 (TTL) の値を設定できます。一般に、アプリケーションで実用的な最も低い TTL 値を使用します。

TTL 値の使用は、最新プロバイダーの CachingMostRecentProvider 記号で変更されます。

> **Note**
>
> 最新プロバイダーの MostRecentProvider 記号は、DynamoDB 暗号化クライアントのサポートされている古いバージョンでは廃止されており、バージョン 2.0.0 から除去されています。これは、CachingMostRecentProvider 記号に置き換えられています。可能な限り早急にコードを更新することをお勧めします。詳細については、「最新プロバイダーの更新」を参照してください。

CachingMostRecentProvider

CachingMostRecentProvider は、以下の 2 つの異なる方法で TTL 値を使用します。

- TTL により、最新プロバイダーがプロバイダーストアで新しいバージョンの CMP をチェックする頻度を決定します。新しいバージョンが利用可能な場合、最新プロバイダーはその CMP を置き換え、暗号化マテリアルを更新します。それ以外の場合、現在の CMP と暗号化マテリアルを引き続き使用します。
- TTL により、キャッシュ内の CMP を使用できる期間を決定します。キャッシュされた CMP を暗号化に使用する前に、最新プロバイダーはキャッシュ内の時間を評価します。CMP キャッシュ時間が TTL を超えると、CMP はキャッシュから削除され、最新プロバイダーはプロバイダストアから新しい最新バージョン CMP を取得します。

MostRecentProvider

MostRecentProvider では、TTL により、最新プロバイダーがプロバイダーストアで新しいバージョンの CMP をチェックする頻度が決定されます。新しいバージョンが利用可能な場合、最新プロバイダーはその CMP を置き換え、暗号化マテリアルを更新します。それ以外の場合、現在の CMP と暗号化マテリアルを引き続き使用します。

TTL では、新しい CMP バージョンが作成される頻度は決定されません。新しい CMP バージョンを作成するには、暗号化マテリアルをローテーションします。

理想的な TTL 値は、アプリケーションとそのレイテンシー、および可用性の目標によって異なります。TTL を低くすると、暗号化マテリアルがメモリに格納される時間が短縮され、セキュリティプロファイルが向上します。また、TTL が低いほど、重要な情報がより頻繁に更新されます。例えば、内部 CMP が Direct KMS プロバイダーである場合、呼び出し元が AWS KMS keyを使用する権限をまだ持っているかを、より頻繁に検証します。

ただし、TTL が低すぎると、プロバイダーストアへの頻繁な呼び出しによってコストが増加し、プロバイダーストアがアプリケーションや、サービスアカウントを共有する他のアプリケーションからのリクエストをスロットリングする可能性があります。また、暗号化マテリアルをローテーションする速度で TTL を調整することでメリットが得られる場合があります。

テスト中に、お使いのアプリケーションと、セキュリティおよびパフォーマンス標準に適した設定が見つかるまで、さまざまなワークロードで TTL とキャッシュサイズを変更します。

暗号化マテリアルの回転

最新プロバイダーで暗号化マテリアルが必要な場合、最新プロバイダーは必ず、認識している最新バージョンの CMP を使用します。新しいバージョンをチェックする頻度は、最新プロバイダーを構成するときに設定した有効期限 (TTL) 値によって決定されます。

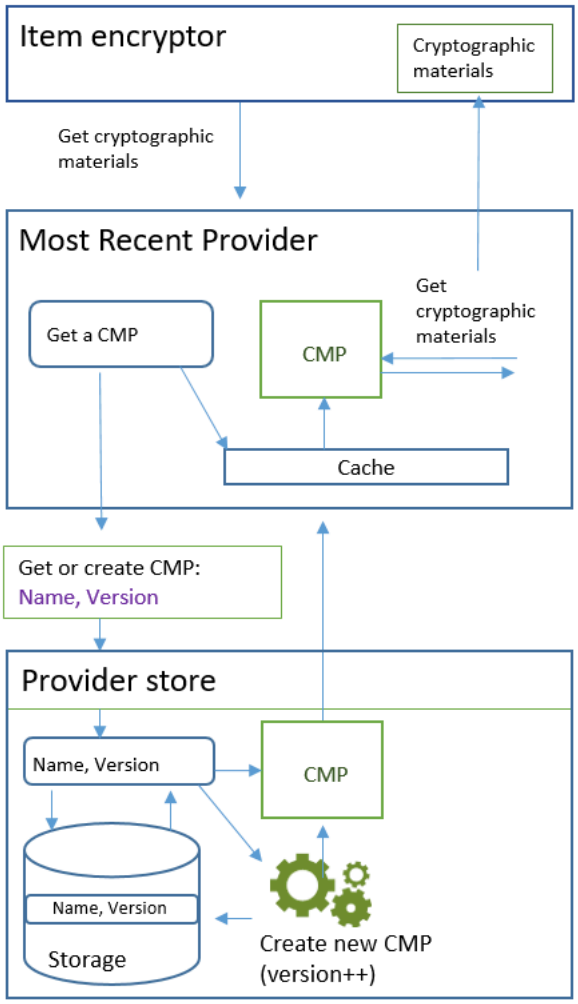
TTL が期限切れになると、最新プロバイダーはプロバイダーストアで新しいバージョンの CMP をチェックします。新しいバージョンを使用できる場合、最新プロバイダーはそれを取得し、キャッシュ内の CMP を置き換えます。プロバイダーストアに新しいバージョンがあることが検出されるまで、この CMP とその暗号化マテリアルが使用されます。

最新プロバイダーの新しいバージョンの CMP を作成するようにプロバイダーストアに指示するには、プロバイダーストアの新規プロバイダーの作成オペレーション作を、最新プロバイダーのマテリアル名で呼び出します。プロバイダーストアは新しい CMP を作成し、暗号化されたコピーをより大きなバージョン番号で内部ストレージに保存します。(CMP を返しますが、破棄することもできます。) その結果、次に最新プロバイダーがプロバイダーストアにその CMP の最大バージョン番号を問い合わせるときに、最新プロバイダーは新しいより大きなバージョン番号を取得し、それをストアに対する後続のリクエストで使用して、CMP の新しいバージョンが作成されたかどうかを確認します。

時間、処理された項目または属性の数、またはアプリケーションに合ったその他のメトリクスに基づいて、新しいプロバイダー作成コールをスケジュールできます。

暗号化マテリアルを取得する

最新プロバイダーは、この図に示す次のプロセスを使用して、項目エンクリプタに返す暗号化マテリアルを取得します。出力は、プロバイダーストアが返す CMP のタイプによって異なります。最新プロバイダーは、DynamoDB 暗号化クライアントに含まれる MetaStore などの互換性のある任意のプロバイダーストアを使用できます。



CachingMostRecentProvider記号を使用して最新プロバイダーを作成するときに、プロバイダーストア、最新プロバイダーの名前、および有効期限 (TTL) 値を指定します。オプションで、キャッシュ内に存在できる暗号化マテリアルの最大数を決定するキャッシュサイズを指定することもできます。

項目エンクリプタが最新プロバイダーに暗号化マテリアルを要求すると、最新プロバイダーは、そのCMP の最新バージョンのキャッシュの検索を開始します。

- キャッシュ内で最新バージョンの CMP を検出し、CMP が TTL 値を超過していない場合、最新プロバイダーは CMP を使用して暗号化マテリアルを生成します。次に、暗号化マテリアルを項目エンクリプタに返します。このオペレーションでは、プロバイダーストアへの呼び出しは必要ありません。
- CMP の最新バージョンがキャッシュ内に存在しない場合、またはキャッシュ内に存在していても TTL 値を超過している場合、最新プロバイダーはそのプロバイダーストアから CMP をリクエスト

します。リクエストには、最新プロバイダーのマテリアル名と、既知の最大のバージョン番号が含まれています。

1. プロバイダーストアは、永続的ストレージから CMP を返します。プロバイダーストアが MetaStore の場合、最新プロバイダーのマテリアル名をパーティションキーとして使用し、バージョン番号をソートキーとして使用して、内部の DynamoDB テーブルから暗号化済みのラップされた CMP を取得します。MetaStore は、内部項目エンクリプタと内部 CMP を使用して、ラップされた CMP を復号します。次に、プレーンテキスト CMP を最新プロバイダーに返します。内部 CMP が Direct KMS Provider の場合、このステップには AWS Key Management Service (AWS KMS) コールが含まれます。
2. CMP は、`amzn-ddb-meta-id`フィールドを実際のマテリアル説明に追加します。その値は、内部テーブルの CMP のマテリアル名とバージョンです。プロバイダーストアは CMP を最新プロバイダーに返します。
3. 最新プロバイダーは CMP をメモリにキャッシュします。
4. 最新プロバイダーは CMP を使用して暗号化マテリアルを生成します。次に、暗号化マテリアルを項目エンクリプタに返します。

復号マテリアルを取得する

項目エンクリプタが最新プロバイダーに復号マテリアルを要求すると、最新プロバイダーは以下のプロセスを使用して、それらを取得し返します。

1. 最新プロバイダーは、項目を暗号化するために使用された暗号化マテリアルのバージョン番号をプロバイダーストアに問い合わせます。項目のマテリアル説明属性から実際のマテリアル説明を渡します。
2. プロバイダーストアは、実際のマテリアル説明の `amzn-ddb-meta-id` フィールドから暗号化 CMP バージョン番号を取得し、最新プロバイダーに返します。
3. 最新プロバイダーは、項目の暗号化と署名に使用された CMP のバージョンをキャッシュ内で検索します。

- CMP の一致するバージョンがキャッシュにあり、かつ、CMP が有効期限 (TTL) 値を超過していないことがわかった場合、最新プロバイダーは CMP を使用して復号マテリアルを生成します。次に、復号マテリアルを項目エンクリプタに返します。このオペレーションでは、プロバイダーストアまたは他の CMP への呼び出しは必要ありません。
- CMP の一致するバージョンがキャッシュ内に存在しない場合、またはキャッシュされた AWS KMS key が TTL 値を超過している場合、最新プロバイダーはそのプロバイダーストアから CMP

をリクエストします。リクエストには、マテリアル名および暗号化 CMP バージョン番号が送信されます。

1. プロバイダーストアは、最新プロバイダー名をパーティションキーとして使用し、バージョン番号をソートキーとして使用して、CMP の永続的ストレージを検索します。
   - 名前とバージョン番号が永続的ストレージにない場合、プロバイダーストアは例外をスローします。CMP を生成するためにプロバイダーストアを使用した場合、意図的に削除されていない限り、CMP は永続的ストレージに保存する必要があります。
   - 一致する名前とバージョン番号を持つ CMP がプロバイダーストアの永続的ストレージにある場合、プロバイダーストアは指定された CMP を最新プロバイダーに返します。

     プロバイダーストアが MetaStore の場合、その DynamoDB テーブルから暗号化済みの CMP を取得します。次に、内部 CMP の暗号化マテリアルを使用して、CMP を最新プロバイダーに返す前に暗号化された CMP を復号します。内部 CMP が Direct KMS Provider の場合、このステップには AWS Key Management Service (AWS KMS) コールが含まれます。
2. 最新プロバイダーは CMP をメモリにキャッシュします。
3. 最新プロバイダーは CMP を使用して復号マテリアルを生成します。次に、復号マテリアルを項目エンクリプタに返します。

最新プロバイダーの更新

最新プロバイダーの記号が `MostRecentProvider` から `CachingMostRecentProvider` に変更されています。

> **Note**
>
> 最新プロバイダーを表す `MostRecentProvider` 記号は、両方の言語実装で、Java 用 DynamoDB 暗号化クライアントバージョン 1.15、および Python 用 DynamoDB 暗号化クライアントバージョン 1.3 では廃止され、DynamoDB 暗号化クライアントバージョン 2.0.0 では除去されています。代わりに、`CachingMostRecentProvider` を使用してください。

`CachingMostRecentProvider` では、以下の変更が実装されます。

- `CachingMostRecentProvider` は、メモリ内の時間が、設定された有効期限 (TTL) 値を超過すると、メモリから暗号化マテリアルを定期的に除去します。

MostRecentProvider は、プロセスの有効期間の間、メモリに暗号化マテリアルを保存する場合があります。その結果、最新プロバイダーは認可の変更を認識しない可能性があります。暗号化キーを使用するための呼び出し元のアクセス許可が取り消された後に、暗号化キーを使用する場合があります。

この新しいバージョンにアップデートできない場合、キャッシュで clear() メソッドを定期的に呼び出すことで同様の効果を得られます。このメソッドは、キャッシュの内容を手動でフラッシュし、最新プロバイダーが新しい CMP と新しい暗号化マテリアルを要求するように求めます。

- また、CachingMostRecentProvider にキャッシュサイズの設定を含めて、キャッシュに対するより詳細な制御を行うこともできます。

CachingMostRecentProvider を更新するには、コード内の記号名を変更する必要があります。その他の点ではすべて、CachingMostRecentProvider には MostRecentProvider との完全な下位互換性があります。テーブル項目を再暗号化する必要はありません。

ただし、CachingMostRecentProvider による、基盤となる主要インフラストラクチャへの呼び出しが増えます。有効期限 (TTL) の各間隔で、プロバイダーストアが少なくとも 1 回呼び出されます。多数のアクティブな CMP を持つアプリケーション (頻繁なローテーションによる)、または大規模なフリートを持つアプリケーションは、この変更の影響を受ける可能性が最も高くなります。

更新したコードをリリースする前に、徹底的にテストして、より頻繁な呼び出しによってアプリケーションが損なわれたり、 AWS Key Management Service (AWS KMS) や Amazon DynamoDB など、プロバイダーが依存するサービスによってスロットリングが発生したりしないことを確認します。パフォーマンスの問題を軽減するために、確認したパフォーマンス特性に基づいて、CachingMostRecentProvider のキャッシュサイズや有効期限 (TTL) を調整してください。ガイダンスについては、「有効期限 (TTL) の値を設定する」を参照してください。

静的マテリアルプロバイダー

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

静的マテリアルプロバイダー (静的 CMP) は、テスト、概念実証デモ、および従来の互換性を目的とした、非常にシンプルな暗号化マテリアルプロバイダー (CMP) です。

静的 CMP を使用してテーブル項目を暗号化するには、Advanced Encryption Standard (AES) 対称暗号化キーと署名キーまたはキーペアを指定します。暗号化された項目を復号するために同じキーを指定する必要があります。静的 CMP は暗号化オペレーションを実行しません。代わりに、項目エンクリプタに指定した暗号化キーをそのまま渡します。項目エンクリプタは、暗号化キーの直下の項目を暗号化します。次に、署名キーを直接使用して署名します。

静的 CMP は一意の暗号化マテリアルを生成しないため、処理するすべてのテーブル項目は同じ暗号化キーで暗号化され、同じ署名キーで署名されます。同じキーを使用して多数の項目の属性値を暗号化するか、同じキーまたはキーペアを使用してすべての項目に署名すると、キーの暗号化の制限を超える危険性があります。

> **Note**
>
> Java ライブラリ内の非対称静的プロバイダーは静的プロバイダーではありません。これは、ラップされた CMP の代替コンストラクタを指定するだけです。本稼働環境での使用は安全ですが、できるだけラップされた CMP を直接使用する必要があります。

静的 CMP は、DynamoDB 暗号化クライアントがサポートしている複数の暗号化マテリアルプロバイダー (CMP) の 1 つです。他の CMP の詳細については、「暗号マテリアルプロバイダー」を参照してください。

サンプルコードについては、以下を参照してください。

- Java: SymmetricEncryptedItem

トピック

- 使用方法
- 仕組み

使用方法

静的なプロバイダーを作成するには、暗号化キーやキーペアおよび署名キーやキーペアを指定します。テーブル項目を暗号化および復号するには、キーマテリアルを指定する必要があります。

Java

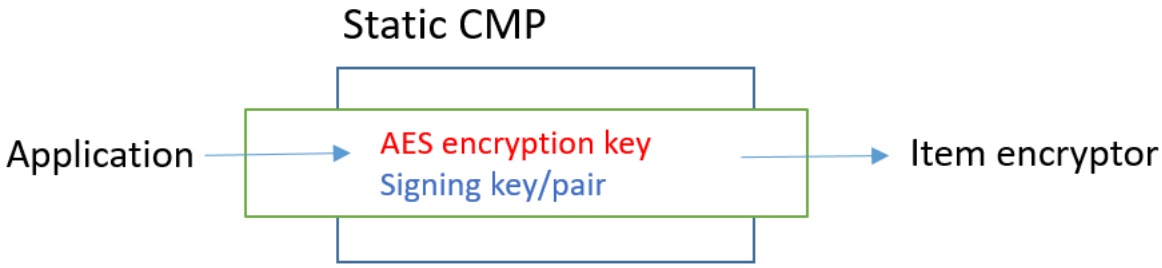
```
// To encrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Signing key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);

// 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
)
```

仕組み

静的プロバイダーは、指定した暗号化キーと署名キーを項目エンクリプタに渡します。ここで、これらのアイテムは、テーブル項目の暗号化と署名に直接使用されます。各項目に異なるキーを指定しない限り、すべての項目で同じキーが使用されます。



暗号化マテリアルを取得する

このセクションでは、暗号化マテリアルのリクエストを受け取る際の静的マテリアルプロバイダー (静的 CMP) の入力、出力、処理の詳細について説明します。

入力 (アプリケーションから)

- 暗号化キー - これは、Advanced Encryption Standard (AES) キーなどの対称キーである必要があります。
- 署名キー - これは、対称キーまたは非対称キーペアです。

入力 (項目エンクリプタから)

- DynamoDB 暗号化コンテキスト

出力 (項目エンクリプタへ)

- 入力として渡される暗号化キー。
- 入力として渡される署名キー。
- 実際のマテリアル説明: リクエストされたマテリアル説明。存在する場合は、変更されません。

復号マテリアルを取得する

このセクションでは、復号マテリアルのリクエストを受け取る際の静的マテリアルプロバイダー (静的 CMP) の入力、出力、処理の詳細について説明します。

暗号化マテリアルの取得と、復号マテリアルの取得のための異なるメソッドが含まれていますが、動作は同じです。

入力 (アプリケーションから)

- 暗号化キー - これは、Advanced Encryption Standard (AES) キーなどの対称キーである必要があります。
- 署名キー - これは、対称キーまたは非対称キーペアです。

入力 (項目エンクリプタから)

- DynamoDB 暗号化コンテキスト (使用しません)

出力 (項目エンクリプタへ)

- 入力として渡される暗号化キー。
- 入力として渡される署名キー。

Amazon DynamoDB Encryption Client で利用可能なプログラミング言語

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

Amazon DynamoDB Encryption Client では、以下のプログラミング言語を使用できます。言語固有のライブラリはさまざまですが、結果として得られる実装は相互運用ができます。たとえば、Java クライアントで項目を暗号化 (および署名) し、Python クライアントで項目を復号することができます。

詳細については、該当するトピックを参照してください。

トピック

- Amazon DynamoDB Encryption Client for Java
- Python 用 DynamoDB 暗号化クライアント

Amazon DynamoDB Encryption Client for Java

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x～2.x および DynamoDB Encryption Client for Python のバージョン 1.x～3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

このトピックでは、Amazon DynamoDB Encryption Client for Java をインストールして使用する方法について説明します。DynamoDB 暗号化クライアントを使用したプログラミングの詳細については、Java の例、GitHub の aws-dynamodb-encryption-java リポジトリにある例、および DynamoDB 暗号化クライアント用の Javadoc を参照してください。

> **Note**
>
> DynamoDB Encryption Client for Java のバージョン 1.x.x は、2022 年 7 月にサポート終了フェーズに入ります。可能な限り早急に新しいバージョンにアップグレードしてください。

トピック

前提条件

Amazon DynamoDB Encryption Client for Java をインストールする前に、以下の前提条件が満たされていることを確認してください。

Java 開発環境

Java 8 以降が必要になります。Oracle のウェブサイトで Java SE のダウンロードに移動し、Java SE Development Kit (JDK) をダウンロードして、インストールします。

Oracle JDK を使用する場合は、Java Cryptography Extension (JCE) 無制限強度の管轄ポリシーファイルをダウンロードして、インストールする必要があります。

AWS SDK for Java

DynamoDB 暗号化クライアントには、アプリケーションが DynamoDB とやり取りしない場合 AWS SDK for Java でも、 の DynamoDB モジュールが必要です。SDK 全体またはこのモジュールだけをインストールできます。Maven を使用している場合は、`aws-java-sdk-dynamodb` を `pom.xml` ファイルに追加します。

のインストールと設定の詳細については AWS SDK for Java、「」を参照してくださいAWS SDK for Java。

インストール

Amazon DynamoDB Encryption Client for Java は、以下の方法でインストールできます。

手動

Amazon DynamoDB Encryption Client for Java をインストールするには、aws-dynamodb-encryption-java GitHub リポジトリをクローンまたはダウンロードしてください。

Apache Maven の使用

Amazon DynamoDB Encryption Client for Java は、以下の依存定義を使用して、Apache Maven を介して利用できます。

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

SDK をインストールしたら、このガイドと GitHub の DynamoDB 暗号化クライアント Javadoc のサンプルコードを確認して開始します。

Java 用 Amazon DynamoDB 暗号化クライアントの使用方法

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x およ

び DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

このトピックでは、Java での Amazon DynamoDB 暗号化クライアントの機能の一部について説明します。他のプログラミング言語には実装されていない機能も含まれます。

DynamoDB 暗号化クライアントを使用したプログラミングの詳細については、Java の例、GitHub の `aws-dynamodb-encryption-java repository` にある例、および DynamoDB 暗号化クライアント用の Javadoc を参照してください。

トピック

- 項目エンクリプタ: AttributeEncryptor および DynamoDBEncryptor
- 保存動作の設定
- Java の属性アクション
- テーブル名の上書き

項目エンクリプタ: AttributeEncryptor および DynamoDBEncryptor

Java の DynamoDB 暗号化クライアントには、下位レベルの DynamoDBEncryptor および AttributeEncryptor という 2 つの項目エンクリプタがあります。

`AttributeEncryptor` は、DynamoDB 暗号化クライアントの で DynamoDBMapper DynamoDB `Encryptor` を使用するのに役立つヘルパークラスです。 AWS SDK for Java DynamoDB `DynamoDBMapper` で `AttributeEncryptor` を使用すると、項目の保存時に項目が透過的に暗号化および署名されます。また、項目のロード時に項目が透過的に検証および復号されます。

保存動作の設定

`AttributeEncryptor` および `DynamoDBMapper` を使用して、署名のみが行われた属性または暗号化および署名された属性を持つテーブル項目を追加またはレプリケートできます。これらのタスクでは、次の例に示すように、PUT 保存動作を使用するよう設定することをお勧めします。そのように設定しない場合、データを復号できないことがあります。

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
```

```
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

テーブルの項目でモデル化された属性のみを更新するデフォルトの保存動作を使用する場合、モデル化されていない属性は署名に含まれず、テーブルの書き込みによって変更されません。その結果、モデル化されていない属性が含まれていないため、その後のすべての属性の読み取りでは、署名は検証されません。

また、CLOBBER 保存動作を使用することもできます。この動作は、オプティミスティックロックを無効にしてテーブルの項目を上書きするという点を除いて、PUT 保存動作と同じです。

署名エラーを防ぐために、`AttributeEncryptor` が CLOBBER または PUT の保存動作で設定されていない `DynamoDBMapper` とともに使用される場合、DynamoDB Encryption Client はランタイム例外をスローします。

サンプル内で使用されているこのコードを確認するには、DynamoDBMapper の使用 と、GitHub の `aws-dynamodb-encryption-java` リポジトリにある AwsKmsEncryptedObject.java の例を参照してください。

Java の属性アクション

属性アクションでは、暗号化されて署名された属性値、署名のみされた属性値、無視される属性値を指定します。属性アクションの指定に使用するメソッドは、`DynamoDBMapper` および `AttributeEncryptor`、または下位レベルの DynamoDBEncryptor の使用有無によって異なります。

> **Important**
>
> 属性アクションを使用してテーブル項目を暗号化した後、データモデルから属性を追加または削除すると、署名の検証エラーが発生し、データの復号ができなくなることがあります。詳細な説明については、「データモデルの変更」を参照してください。

DynamoDBMapper の属性アクション

`DynamoDBMapper` および `AttributeEncryptor` を使用する場合は、注釈を使用して属性アクションを指定します。DynamoDB 暗号化クライアントは標準の DynamoDB 属性の注釈を使用して、属性を保護する方法を決定する属性のタイプを定義します。デフォルトでは、プライマリキーを除く属性がすべて暗号化されます。これらの属性は署名されますが、暗号化はされません。

> **Note**
>
> @DynamoDBVersionAttribute 注釈を使用して属性値を暗号化できます。ただし、署名することはできます (署名する必要があります)。それ以外の場合、その値を使用する条件によって、意図しない結果をもたらす場合があります。

```
// 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")
```

例外を指定するには、Java 用 Amazon DynamoDB 暗号化クライアントに定義されている暗号化注釈を使用します。クラスレベルで指定した場合は、クラスのデフォルト値になります。

```
// Sign only
@DoNotEncrypt

// Do nothing; not encrypted or signed
@DoNotTouch
```

たとえば、これらの注釈で署名するが、PublicationYear 属性を暗号化しない場合は、ISBN 属性値を暗号化または署名しないでください。

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

DynamoDBEncryptor の属性アクション

DynamoDBEncryptor を使用する際に属性アクションを直接指定するには、名前と値のペアで属性名と指定されたアクションを表している HashMap オブジェクトを作成します。

属性アクションの有効な値は、列挙型の EncryptionFlags で定義されています。ENCRYPT と SIGN を一緒に使用したり、SIGN を単独で使用したりできます。また、両方除外することもできます。ただし、ENCRYPT を単独で使用すると、DynamoDB 暗号化クライアントはエラーをスローします。未署名の属性を暗号化することはできません。

```
ENCRYPT
SIGN
```

> **Warning**
>
> プライマリキー属性を暗号化しないでください。DynamoDB でテーブル全体のスキャンを実行せずに項目を見つけられるように、プレーンテキストの状態を維持する必要があります。

暗号化コンテキストでプライマリキーを指定し、いずれかのプライマリキー属性の属性アクションで ENCRYPT を指定した場合、DynamoDB 暗号化クライアントは例外をスローします。

たとえば、次の Java コードは、record 項目内のすべての属性を暗号化および署名する actions HashMap を作成します。例外は、署名されているが暗号化されていないパーティションキー属性とソートキー属性、および署名または暗号化されていない test 属性です。

```
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;
  }
```

```
}
```

その後、DynamoDBEncryptor の encryptRecord メソッドを呼び出すときに、attributeFlags パラメータの値としてマップを指定します。たとえば、この encryptRecord の呼び出しでは、actions マップが使用されます。

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

テーブル名の上書き

DynamoDB 暗号化クライアントでは、DynamoDB テーブルの名前は、暗号化メソッドおよび復号メソッドに渡される DynamoDB 暗号化コンテキストの要素です。テーブル項目を暗号化または署名すると、テーブル名を含む DynamoDB 暗号化コンテキストが暗号化テキストに暗号でバインドされます。復号メソッドに渡される DynamoDB 暗号化コンテキストが、暗号化メソッドに渡された DynamoDB 暗号化コンテキストと一致しない場合、復号オペレーションは失敗します。

テーブルをバックアップする場合や、ポイントインタイムリカバリを実行する場合など、テーブルの名前が変更されることがあります。これらの項目の署名を復号または検証する際、元のテーブル名を含む、項目の暗号化と署名に使用されたのと同じ DynamoDB 暗号化コンテキストを渡す必要があります。現在のテーブル名は必要ありません。

DynamoDBEncryptor を使用する場合、DynamoDB 暗号化コンテキストを手動で組み立てます。ただし、DynamoDBMapper を使用している場合は、AttributeEncryptor によって現在のテーブル名を含む DynamoDB 暗号化コンテキストが作成されます。異なるテーブル名で暗号化コンテキストを作成するよう AttributeEncryptor に指示するには、EncryptionContextOverrideOperator を使用します。

たとえば、次のコードは、暗号化マテリアルプロバイダー (CMP) と DynamoDBEncryptor のインスタンスを作成します。次に、DynamoDBEncryptor の setEncryptionContextOverrideOperator メソッドを呼び出します。これは、1 つのテーブル名を上書きする overrideEncryptionContextTableName 演算子を使用します。このように設定すると、AttributeEncryptor によって oldTableName の代わりに newTableName を含む DynamoDB 暗号化コンテキストが作成されます。完全な例については、EncryptionContextOverridesWithDynamoDBMapper.java を参照してください。

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

```
encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionCon
                oldTableName, newTableName));
```

項目を復号および検証する DynamoDBMapper の load メソッドを呼び出す際、元のテーブル名を指定します。

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTabl
                .build());
```

また、複数のテーブル名を上書きする `overrideEncryptionContextTableNameUsingMap` 演算子を使用することもできます。

テーブル名の上書き演算子は通常、データの復号と署名の検証に使用されます。ただし、それらの演算子を使用して、暗号化および署名時に DynamoDB 暗号化コンテキスト内のテーブル名を別の値に設定することができます。

`DynamoDBEncryptor` を使用している場合は、テーブル名の上書き演算子を使用しないでください。代わりに、元のテーブル名で暗号化コンテキストを作成し、復号メソッドに送信します。

Java 用 DynamoDB 暗号化クライアントのサンプルコード

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

以下の例では、Java 用 DynamoDB 暗号化クライアントを使用して、アプリケーションの DynamoDB テーブル項目を保護する方法について説明します。GitHub の aws-dynamodb-encryption-java リポジトリの examples ディレクトリに、その他の例 (および独自の使用に役立つ例) があります。

トピック

- DynamoDBEncryptor の使用

- DynamoDBMapper の使用

DynamoDBEncryptor の使用

この例は、下位レベルの DynamoDBEncryptor を Direct KMS プロバイダーで使用する方法を示しています。Direct KMS プロバイダーは、指定した AWS KMS key in AWS Key Management Service (AWS KMS) で暗号化マテリアルを生成して保護します。

互換性のある暗号化マテリアルプロバイダー (CMP) を DynamoDBEncryptor で使用できます。また、Direct KMS プロバイダーを DynamoDBMapper および AttributeEncryptor で使用できます。

完全なコードサンプルの参照: AwsKmsEncryptedItem.java

ステップ 1: Direct KMS プロバイダーを作成する

指定されたリージョンで AWS KMS クライアントのインスタンスを作成します。次に、クライアントインスタンスを使用して、任意の AWS KMS keyで Direct KMS プロバイダーのインスタンスを作成します。

この例では、Amazon リソースネーム (ARN) を使用して を識別しますが AWS KMS key、任意の有効なキー識別子を使用できます。

```
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);
```

ステップ 2: 項目を作成する

この例では、サンプルテーブル項目を表す record HashMap を定義します。

```
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"));
```

ステップ 3: DynamoDBEncryptor を作成する

Direct KMS プロバイダーを使用して DynamoDBEncryptor のインスタンスを作成します。

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

ステップ 4: DynamoDB 暗号化コンテキストを作成する

DynamoDB 暗号化コンテキストには、テーブル構造に関する情報と、暗号化および署名の方法が含まれます。DynamoDBMapper を使用する場合は、AttributeEncryptor で暗号化テキストが作成されます。

```
final String tableName = "testTable";

final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

ステップ 5: 属性アクションオブジェクトを作成する

属性アクションでは、暗号化されて署名された項目の属性値、署名のみされた項目の属性値、暗号化も署名もされていない項目の属性値を指定します。

Java で属性アクションを指定するには、属性名と EncryptionFlags 値のペアの HashMap を作成します。

たとえば、以下の Java コードでは、actions 項目のすべての属性を暗号化して署名する record HashMap を作成します。ただし、署名済みだが暗号化されていないパーティションキーおよびソートキー属性、暗号化されていない未署名の test 属性は除きます。

```
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;
  }
}
```

ステップ 6: 項目を暗号化および署名する

テーブル項目を暗号化して署名するには、`encryptRecord` のインスタンスで `DynamoDBEncryptor` メソッドを呼び出します。テーブル項目 (`record`)、属性アクション (`actions`)、暗号化テキスト (`encryptionContext`) を指定します。

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

ステップ 7: DynamoDB テーブルに項目を入力する

最後に、暗号化された署名済みの項目を DynamoDB テーブルに入力します。

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

DynamoDBMapper の使用

以下の例は、DynamoDB マッパーヘルパークラスを Direct KMS プロバイダーで使用する方法を示しています。Direct KMS プロバイダーは、指定した AWS Key Management Service (AWS KMS) の AWS KMS key で暗号化マテリアルを生成して、保護します。

互換性のある暗号化マテリアルプロバイダー (CMP) を `DynamoDBMapper` で使用できます。また、Direct KMS プロバイダーを下位レベルの `DynamoDBEncryptor` で使用できます。

完全なコードサンプルの参照: AwsKmsEncryptedObject.java

ステップ 1: Direct KMS プロバイダーを作成する

指定されたリージョンで AWS KMS クライアントのインスタンスを作成します。次に、クライアントインスタンスを使用して、任意の AWS KMS keyで Direct KMS プロバイダーのインスタンスを作成します。

この例では、Amazon リソースネーム (ARN) を使用して を識別しますが AWS KMS key、任意の有効なキー識別子を使用できます。

```
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);
```

ステップ 2: DynamoDB エンクリプタと DynamoDBMapper を作成する

前のステップで作成した Direct KMS プロバイダーを使用して、DynamoDB エンクリプタのインスタンスを作成します。DynamoDB マッパーを使用するには、下位レベルの DynamoDB エンクリプタをインスタンス化する必要があります。

次に、DynamoDB データベースのインスタンスとマッパー設定を作成し、それらを使用して DynamoDB マッパーのインスタンスを作成します。

> **Important**
>
> DynamoDBMapper を使用して、署名された (または暗号化されて署名された) 項目を追加または編集するときは、以下の例に示されているように、PUT のような保存動作を使用するように設定して、すべての属性が含まれるようにします。そのように設定しない場合、データを復号できないことがあります。

```
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));
```

ステップ 3: DynamoDB テーブルを定義する

次に、DynamoDB テーブルを定義します。注釈を使用して、属性アクションを指定します。この例では、DynamoDB テーブルとして ExampleTable を作成し、テーブル項目を表す DataPoJo クラスを作成します。

このサンプルテーブルでは、プライマリキーの属性は署名されますが、暗号化されません。これは、@DynamoDBHashKey という注釈が付いた partition_attribute に適用されます。また、@DynamoDBRangeKey という注釈が付いた sort_attribute に適用されます。

@DynamoDBAttribute という注釈が付いた属性 (some numbers など) は暗号化されて署名されます。例外は、DynamoDB 暗号化クライアントで定義された @DoNotEncrypt (署名のみ) または @DoNotTouch (暗号化も署名もなし) 暗号化注釈を使用する属性です。たとえば、leave me 属性には @DoNotTouch 注釈が付いているため、暗号化も署名もされません。

```
@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 +
"]";
  }
```

```
}
```

ステップ 4: テーブル項目を暗号化して保存する

これで、テーブル項目を作成し、DynamoDB マッパーを使用して項目を保存すると、項目はテーブルに追加される前に自動的に暗号化されて署名されます。

この例では、`record` というテーブル項目を定義しています。この項目がテーブルに保存される前に、`DataPoJo` クラスの注釈に基づいて、その属性は暗号化されて署名されます。この場合、`PartitionAttribute`、`SortAttribute`、`LeaveMe` を除くすべての属性が暗号化されて署名されます。`PartitionAttribute` と `SortAttributes` は署名のみされます。`LeaveMe` 属性は暗号化または署名されていません。

`record` 項目を暗号化して署名し、`ExampleTable` に追加するには、`DynamoDBMapper` クラスの `save` メソッドを呼び出します。DynamoDB マッパーは PUT 保存動作を使用するように設定されているため、項目は更新されず、代わりに同じプライマリキーを使用する項目に置き換えられます。これにより、確実に署名が一致するようになり、その項目をテーブルからの取得時に復号化できます。

```
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);
```

Python 用 DynamoDB 暗号化クライアント

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

このトピックでは、Python 用 DynamoDB 暗号化クライアントをインストールして使用する方法について説明します。このコードは、GitHub の aws-dynamodb-encryption-python リポジトリにあり、開始するのに役立つ完全でテスト済みのサンプルコードが含まれています。

> **Note**
>
> DynamoDB Encryption Client for Python のバージョン 1.x.x および 2.x.x は、2022 年 7 月にサポート終了フェーズに入ります。可能な限り早急に新しいバージョンにアップグレードしてください。

トピック

前提条件

Amazon DynamoDB Encryption Client for Python をインストールする前に、以下の前提条件が満たされていることを確認してください。

Python のサポートされているバージョン

Amazon DynamoDB Encryption Client for Python バージョン 3.3.0 以降では、Python 3.8 以降が必要です。Python をダウンロードするには、「Python のダウンロード」を参照してください。

Amazon DynamoDB Encryption Client for Python の以前のバージョンでは Python 2.7 および Python 3.4 以降がサポートされていますが、最新バージョンの DynamoDB 暗号化クライアントを使用することをお勧めします。

Python 用 pip インストールツール

Python 3.6 以降には pip が含まれていますが、アップグレードすることもできます。pip のアップグレードまたはインストールの詳細については、pip ドキュメント内のインストールを参照してください。

インストール

以下の例に示すように、pip を使用して Amazon DynamoDB Encryption Client for Python をインストールします。

最新バージョンをインストールするには

```
pip install dynamodb-encryption-sdk
```

pip を使用してパッケージをインストールおよびアップグレードする方法の詳細については、「パッケージのインストール」を参照してください。

DynamoDB 暗号化クライアントでは、すべてのプラットフォームで cryptography ライブラリが必要です。pip のすべてのバージョンでは、Windows に cryptography ライブラリがインストールされて構築されます。pip 8.1 以降では、Linux に cryptography がインストールされて構築されます。以前のバージョンの pip を使用していて、Linux 環境に暗号ライブラリを構築するために必要なツールがない場合は、それらをインストールする必要があります。詳細については、「Building cryptography on Linux」を参照してください。

DynamoDB 暗号化クライアントの最新開発バージョンは、GitHub の aws-dynamodb-encryption-python リポジトリから取得できます。

DynamoDB 暗号化クライアントをインストールしたら、このガイドの Python コードの例を見ながら開始します。

Python 用 Amazon DynamoDB 暗号化クライアントの使用方法

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

このトピックでは、Python 用 Amazon DynamoDB 暗号化クライアントの機能の一部について説明します。他のプログラミング言語には実装されていない機能も含まれます。これらの機能は、最も安全

な方法で DynamoDB 暗号化クライアントを簡単に使用できるように設計されています。通常とは異なるユースケースを除き、この方法を使用することをお勧めします。

DynamoDB 暗号化クライアントを使用したプログラミングの詳細については、このガイドの Python の例、GitHub の aws-dynamodb-encryption-python リポジトリにある例、および DynamoDB 暗号化クライアント用の Python ドキュメントを参照してください。

トピック

- クライアントのヘルパークラス
- TableInfo クラス
- Python の属性アクション

クライアントのヘルパークラス

Python 用 DynamoDB 暗号化クライアントには、DynamoDB の Boto 3 クラスをミラーリングする複数のクライアントヘルパークラスが含まれています。これらのヘルパークラスでは、次のように、暗号化の追加、既存の DynamoDB アプリケーションへの署名、一般的な問題の回避を簡単に行うことができます。

- 項目のプライマリキーを暗号化できないように、プライマリキーの上書きアクションを AttributeActions オブジェクトを追加するか、`AttributeActions` オブジェクトを使用してプライマリキーを暗号化するようにクライアントに明示的に指示している場合は例外をスローします。`AttributeActions` オブジェクトのデフォルトアクションが `DO_NOTHING` の場合、クライアントのヘルパークラスではプライマリキーのアクションが使用されます。それ以外の場合は、`SIGN_ONLY` を使用します。
- TableInfo オブジェクトを作成し、DynamoDB への呼び出しに基づいて DynamoDB 暗号化コンテキストにデータを入力します。これにより、DynamoDB 暗号化コンテキストの精度が確保され、クライアントはプライマリキーを識別できるようになります。
- DynamoDB テーブルが読み書きされるときにテーブル項目を透過的に暗号化および復号するメソッド (`put_item` や `get_item` など) をサポートしています。ただし、`update_item` メソッドはサポートされていません。

クライアントのヘルパークラスを使用します。低レベルの項目エンクリプタを使用して直接やり取りする必要はありません。項目エンクリプタで高度オプションを設定する必要がある場合を除き、これらのクラスを使用します。

クライアントのヘルパークラスには、以下のものが含まれます。

- EncryptedTable: 1 つのテーブルを同時に処理するために DynamoDB でテーブルリソースを使用するアプリケーション用。
- EncryptedResource: バッチ処理用に DynamoDB でサービスリソースクラスを使用するアプリケーション用。
- EncryptedClient: DynamoDB で低レベルクライアントを使用するアプリケーション用。

クライアントのヘルパークラスを使用するには、ターゲットテーブルの DynamoDB DescribeTable オペレーションを呼び出すアクセス許可が発信者に必要です。

TableInfo クラス

TableInfo クラスは、DynamoDB テーブルを表すヘルパークラスです。プライマリキーとセカンダリインデックスのフィールドを使用します。これにより、テーブルに関する正確なリアルタイム情報を簡単に取得できます。

クライアントのヘルパークラスを使用している場合は、`TableInfo` オブジェクトが作成、使用されます。それ以外の場合、オブジェクトを明示的に作成できます。例については、項目エンクリプタを使用するを参照してください。

`TableInfo` オブジェクトで `refresh_indexed_attributes` メソッドを呼び出すと、DynamoDB DescribeTable オペレーションを呼び出して、オブジェクトのプロパティ値が入力されます。テーブルのクエリは、ハードコーディングのインデックス名よりも信頼性がはるかに高まります。`TableInfo` クラスには、DynamoDB 暗号化コンテキストに必要な値を提供する `encryption_context_values` プロパティも含まれます。

`refresh_indexed_attributes` メソッドを使用するには、ターゲットテーブルの DynamoDB DescribeTable オペレーションを呼び出すアクセス許可が発信者に必要です。

Python の属性アクション

属性アクションは、項目の各属性に対して実行するアクションを項目エンクリプタに指示します。属性アクションを Python で指定するには、デフォルトアクションで `AttributeActions` オブジェクトと、特定の属性の例外を作成します。有効な値は、列挙型の `CryptoAction` で定義されています。

> **Important**
>
> 属性アクションを使用してテーブル項目を暗号化した後、データモデルから属性を追加または削除すると、署名の検証エラーが発生し、データの復号ができなくなることがあります。詳細な説明については、「データモデルの変更」を参照してください。

```
DO_NOTHING = 0
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

たとえば、この AttributeActions オブジェクトは、すべての属性のデフォルトとして ENCRYPT_AND_SIGN を確立し、ISBN 属性および PublicationYear 属性の例外を指定します。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

クライアントのヘルパークラスを使用している場合は、プライマリキー属性の属性アクションを指定する必要はありません。クライアントのヘルパークラスを使用して、プライマリキーを暗号化することはできません。

クライアントのヘルパークラスを使用しておらず、デフォルトアクションが ENCRYPT_AND_SIGN の場合は、プライマリキーのアクションを指定する必要があります。プライマリキーに推奨されているアクションは SIGN_ONLY です。簡単に行うには、set_index_keys メソッドを使用します。このメソッドでは、プライマリキーに SIGN_ONLY、デフォルトアクションの場合には DO_NOTHING が使用されます。

> **Warning**
>
> プライマリキー属性を暗号化しないでください。DynamoDB でテーブル全体のスキャンを実行せずに項目を見つけられるように、プレーンテキストの状態を維持する必要があります。

```
actions = AttributeActions(
```

```
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Python 用 DynamoDB 暗号化クライアントのサンプルコード

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x～2.x および DynamoDB Encryption Client for Python のバージョン 1.x～3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

以下の例では、Python 用 DynamoDB 暗号化クライアントを使用して、アプリケーションの DynamoDB データを保護する方法について説明します。GitHub の aws-dynamodb-encryption-python リポジトリの examples ディレクトリに、その他の例 (および独自の使用に役立つ例) があります。

トピック

- EncryptedTable クライアントヘルパークラスを使用する
- 項目エンクリプタを使用する

EncryptedTable クライアントヘルパークラスを使用する

以下の例は、`EncryptedTable` クライアントヘルパークラスで Direct KMS プロバイダーを使用する方法を示しています。この例では、次の 項目エンクリプタを使用する の例と同じ暗号化マテリアルプロバイダーを使用しています。ただし、低レベルの項目エンクリプタと直接やり取りするのではなく、`EncryptedTable` クラスを使用します。

これらの例を比較することで、クライアントのヘルパークラスが行う作業を確認できます。この処理では、DynamoDB 暗号化コンテキストを作成します。また、プライマリキー属性が常に署名されているが暗号化されていないことを確認します。暗号化コンテキストを作成し、プライマリキーを検出するには、クライアントのヘルパークラスで DynamoDB DescribeTable オペレーションを呼び出します。このコードを実行するには、このオペレーションを呼び出すアクセス許可が必要です。

完全なコードサンプルの参照: aws_kms_encrypted_table.py

ステップ 1: テーブルを作成する

開始するには、テーブル名を指定して、標準の DynamoDB テーブルのインスタンスを作成します。

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

ステップ 2: 暗号化マテリアルプロバイダーを作成する

選択した暗号化マテリアルプロバイダー (CMP) のインスタンスを作成します。

この例では、Direct KMS プロバイダーを使用していますが、互換性のある CMP を使用することもできます。Direct KMS プロバイダーを作成するには、AWS KMS key を指定します。この例では、 の Amazon リソースネーム (ARN) を使用しますが AWS KMS key、任意の有効なキー識別子を使用できます。

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

ステップ 3: 属性アクションオブジェクトを作成する

属性アクションは、項目の各属性に対して実行するアクションを項目エンクリプタに指示します。この例の `AttributeActions` オブジェクトは、無視される `test` 属性を除くすべての項目を暗号化し、署名します。

クライアントのヘルパークラスを使用する場合は、プライマリキー属性の属性アクションを指定しないでください。`EncryptedTable` クラスでは、プライマリキー属性に署名しますが、暗号化しません。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

ステップ 4: 暗号化されたテーブルを作成する

標準テーブル、Direct KMS プロバイダー、属性アクションを使用して、暗号化されたテーブルを作成します。このステップで設定を完了します。

```
encrypted_table = EncryptedTable(
```

```
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
)
```

ステップ 5: テーブルにプレーンテキスト項目を入力する

encrypted_table で put_item メソッドを呼び出すと、テーブル項目は透過的に暗号化されて署名された後、DynamoDB テーブルに追加されます。

まず、テーブル項目を定義します。

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

次に、テーブルにデータを入力します。

```
encrypted_table.put_item(Item=plaintext_item)
```

DynamoDB テーブルから暗号化形式で項目を取得するには、table オブジェクトに対して get_item メソッドを呼び出します。復号された項目を取得するには、get_item オブジェクトの encrypted_table メソッドを呼び出します。

項目エンクリプタを使用する

この例は、テーブル項目を暗号化するときに、項目エンクリプタと自動的にやり取りするクライアントヘルパークラスを使用する代わりに、DynamoDB 暗号化クライアントで項目エンクリプタと直接やり取りする方法を示しています。

この方法を使用するときは、DynamoDB 暗号化コンテキストと設定オブジェクト (CryptoConfig) を手動で作成します。また、1 つの呼び出しで項目を暗号化し、別の呼び出しで DynamoDB テーブルにその項目を入力します。これにより、put_item 呼び出しのカスタマイズや、DynamoDB 暗号化クライアントを使用した構造化データの暗号化および署名を行うことができます。このデータが DynamoDB に送信されることはありません。

この例では、Direct KMS プロバイダーを使用していますが、互換性のある CMP を使用することもできます。

完全なコードサンプルの参照: aws_kms_encrypted_item.py

ステップ 1: テーブルを作成する

開始するには、テーブル名を指定して、標準の DynamoDB テーブルリソースのインスタンスを作成します。

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

ステップ 2: 暗号化マテリアルプロバイダーを作成する

選択した暗号化マテリアルプロバイダー (CMP) のインスタンスを作成します。

この例では、Direct KMS プロバイダーを使用していますが、互換性のある CMP を使用することもできます。Direct KMS プロバイダーを作成するには、AWS KMS key を指定します。この例では、 の Amazon リソースネーム (ARN) を使用しますが AWS KMS key、任意の有効なキー識別子を使用できます。

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

ステップ 3: TableInfo ヘルパークラスを使用する

DynamoDB からテーブルに関する情報を取得するには、TableInfo ヘルパークラスのインスタンスを作成します。項目エンクリプタで直接操作する場合は、TableInfo インスタンスを作成してそのメソッドを呼び出す必要があります。クライアントのヘルパークラスを使用してこの操作を行うことができます。

TableInfo の refresh_indexed_attributes メソッドでは、DescribeTable DynamoDB オペレーションを使用して、テーブルに関するリアルタイムで正確な情報を取得します。この情報には、プライマリキーと、ローカルおよびグローバルセカンダリインデックスが含まれます。DescribeTable を呼び出すアクセス許可が発信者に必要です。

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

ステップ 4: DynamoDB 暗号化コンテキストを作成する

DynamoDB 暗号化コンテキストには、テーブル構造に関する情報と、暗号化および署名の方法が含まれます。この例では、項目エンクリプタとやり取りするため、DynamoDB 暗号化コンテキストを明示的に作成します。クライアントのヘルパークラスでは、DynamoDB 暗号化コンテキストが作成されます。

パーティションキーおよびソートキーを取得するには、TableInfo ヘルパークラスのプロパティを使用できます。

```
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)
)
```

ステップ 5: 属性アクションオブジェクトを作成する

属性アクションは、項目の各属性に対して実行するアクションを項目エンクリプタに指示します。この例の AttributeActions オブジェクトは、署名されているが暗号されていないプライマリキー属性と、無視される test 属性を除き、すべての項目を暗号化し、署名します。

項目エンクリプタを使用して直接やり取りし、デフォルトアクションが ENCRYPT_AND_SIGN の場合、プライマリキーの代替アクションを指定する必要があります。set_index_keys メソッドを使用できます。このメソッドでは、プライマリキーに SIGN_ONLY、デフォルトアクションの場合には DO_NOTHING が使用されます。

プライマリキーを指定するために、この例では、TableInfo オブジェクトのインデックスキーを使用します。これは、DynamoDB への呼び出しによって指定されます。この技術は、ハードコーディングのプライマリキー名より安全です。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

```
actions.set_index_keys(*table_info.protected_index_keys())
```

ステップ 6: 項目の設定を作成する

DynamoDB 暗号化クライアントを設定するには、テーブル項目の CryptoConfig 設定で先ほど作成したオブジェクトを使用します。クライアントのヘルパークラスでは、CryptoConfig が作成されます。

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
)
```

ステップ 7: 項目を暗号化する

このステップでは、項目を暗号化および署名しますが、DynamoDB テーブルには入力されません。

クライアントのヘルパークラスを使用するときに、項目は透過的に暗号化されて署名され、その後、ヘルパークラスの put_item メソッドを呼び出すときに、DynamoDB テーブルに追加されます。項目エンクリプタを直接使用する場合、暗号化および入力アクションは独立しています。

まず、プレーンテキスト項目を作成します。

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

次に、その項目を暗号化して署名します。encrypt_python_item メソッドでは、CryptoConfig 設定オブジェクトが必要です。

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

ステップ 8: テーブルに項目を入力する

このステップでは、暗号化された署名済みの項目を DynamoDB テーブルに入力します。

```
table.put_item(Item=encrypted_item)
```

暗号化された項目を表示するには、元の get_item オブジェクトの table メソッドを呼び出します。encrypted_table オブジェクトではありません。これにより、検証および復号せずに、DynamoDB テーブルより項目を取得することができます。

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

次の画像は、暗号化された署名済みのテーブル項目の例の一部を示します。

暗号化された属性値は、バイナリデータです。プライマリキー属性の名前および値 (partition_attribute および sort_attribute) と、test 属性は、プレーンテキスト形式のままです。また、この出力は、署名を含む属性 (*amzn-ddb-map-sig*) とマテリアル説明属性 (*amzn-ddb-map-desc*) を示します。

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\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'
}
```

データモデルの変更

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

項目を暗号化または復号するたびに、暗号化して署名する属性、署名する (ただし、暗号化はしない) 属性、および無視する属性を DynamoDB 暗号化クライアントに伝達する属性アクションを指定する必要があります。属性アクションは、暗号化された項目に保存されないため、DynamoDB 暗号化クライアントは属性アクションを自動的に処理しません。

> **Important**
>
> DynamoDB Encryption Client は、既存の暗号化されていない DynamoDB テーブルデータの暗号化をサポートしていません。

データモデルを変更するたびに、つまり、テーブル項目から属性を追加または削除すると、エラーが発生する危険性があります。指定した属性アクションが、項目のすべての属性で構成されていない場合、その項目は、意図した方法で暗号化および署名されない場合があります。さらに重要な点として、項目の復号時に指定する属性アクションが、項目の暗号化時に指定した属性アクションと異なる場合は、署名検証が失敗する場合があります。

たとえば、項目の暗号化に使用する属性アクションで、`test` 属性に署名するよう指示した場合、その項目の署名には `test` 属性が含まれます。項目の復号に使用する属性アクションが、`test` 属性で構成されていない場合、クライアントは `test` 属性を含まない署名の検証を試みるため、検証は失敗します。

これは、DynamoDB 暗号化クライアントがすべてのアプリケーションの項目に対して同じ署名を計算する必要があるため、複数のアプリケーションが同じ DynamoDB 項目の読み取りおよび書き込みを行う場合に特に問題になります。また、属性アクションの変更がすべてのホストに反映される必要があるため、分散アプリケーションでも問題になります。DynamoDB テーブルが 1 つのプロセスで 1 つのホストによってアクセスされる場合でも、ベストプラクティスプロセスを確立すると、プロジェクトが複雑になった場合にエラーを防ぐことができます。

テーブル項目を読み取ることができない署名検証エラーを回避するには、次のガイダンスを使用します。

- 属性の追加 — 新しい属性によって属性アクションが変更される場合は、項目に新しい属性を含める前に属性アクションの変更を完全にデプロイします。
- 属性の削除 - 項目で属性の使用を中止する場合は、属性アクションを変更しないでください。
- アクションの変更 - 属性アクション設定を使用してテーブル項目を暗号化した後は、テーブル内のすべての項目を再暗号化しなければ、デフォルトのアクションまたは既存の属性のアクションを安全に変更することはできません。

署名検証エラーは解決が非常に困難な場合があるため、最善の方法は、それらのエラーを回避することです。

トピック

属性の追加

テーブル項目に新しい属性を追加する場合、属性アクションの変更が必要になることがあります。署名検証エラーを回避するために、この変更を 2 ステージのプロセスで実装することをお勧めします。第 2 ステージを開始する前に、第 1 ステージが完了していることを確認します。

1. テーブルの読み取りまたは書き込みを行うすべてのアプリケーションで属性アクションを変更します。これらの変更をデプロイして、更新がすべての送信先ホストに反映されていることを確認します。
2. テーブル項目の新しい属性に値を書き込みます。

この 2 ステージのアプローチでは、すべてのアプリケーションおよびホストに同じ属性アクションが設定され、新しい属性が見つかる前に同じ署名が計算されます。これは、属性のアクションが何もしない (暗号化または署名しない) 場合でも重要です。その理由は、一部の暗号化では暗号化と署名がデフォルトであるためです。

次の例は、このプロセスの第 1 ステージのコードを示しています。新しい項目属性 `link` が追加されます。これには、別のテーブル項目へのリンクが保存されます。このリンクはプレーンテキストのままにする必要があるため、この例では署名のみアクションを割り当てます。この変更を完全にデプロイし、すべてのアプリケーションおよびホストに新しい属性アクションがあることを確認したら、テーブル項目で `link` 属性の使用を開始します。

Java DynamoDB Mapper

> `DynamoDB Mapper` と `AttributeEncryptor` を使用すると、デフォルトでは、プライマリキーを除く属性がすべて暗号化されます。これらの属性は署名されますが、暗号化はされません。署名のみアクションを指定するには、`@DoNotEncrypt` 注釈を使用します。
>
> この例では、新しい `link` 属性に `@DoNotEncrypt` 注釈を使用します。

```
@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

下位レベルの DynamoDB エンクリプタでは、属性ごとにアクションを設定する必要があります。この例では、switch 文を使用します。デフォルトは encryptAndSign で、パーティションキー、ソートキー、および新しい link 属性に例外が指定されています。この例では、リンク属性コードが使用前に完全にデプロイされていない場合、リンク属性の暗号化および署名を行うアプリケーションや、リンク属性の署名のみを行うアプリケーションがあります。

```
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

Python 用の DynamoDB 暗号化クライアントでは、すべての属性にデフォルトのアクションを指定してから、例外を指定できます。

Python クライアントのヘルパークラスを使用している場合は、プライマリキー属性の属性アクションを指定する必要はありません。クライアントのヘルパークラスを使用して、プライマリキーを暗号化することはできません。ただし、クライアントのヘルパークラスを使用していない場合は、パーティションキーとソートキーに SIGN_ONLY アクションを設定する必要があります。パーティションキーまたはソートキーを誤って暗号化した場合、完全なテーブルスキャンを行わないとデータを復元できません。

この例では、SIGN_ONLY アクションを取得する新しい link 属性の例外を指定します。

```
actions = AttributeActions(
```

```
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

属性の削除

DynamoDB 暗号化クライアントで暗号化された項目で属性が不要になった場合は、その属性の使用を停止できます。ただし、その属性のアクションを削除または変更しないでください。その削除または変更を行ってから、その属性を持つ項目が見つかった場合、その項目に対して計算された署名は元の署名と一致せず、署名の検証は失敗します。

コードから属性のすべてのトレースを削除したいと思うかもしれませんが、項目を削除するのではなく、項目が使用されなくなったというコメントを追加してください。完全なテーブルスキャンを実行して属性のすべてのインスタンスを削除しても、その属性を持つ暗号化された項目は、設定のどこかでキャッシュされるか、または処理中になる可能性があります。

DynamoDB 暗号化クライアントアプリケーションの問題のトラブルシューティング

> **Note**
>
> クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。次のトピックには、DynamoDB Encryption Client for Java のバージョン 1.x ~ 2.x および DynamoDB Encryption Client for Python のバージョン 1.x ~ 3.x に関する情報が記載されています。詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

このセクションでは、DynamoDB 暗号化クライアントを使用する際に直面する可能性のある問題を示すとともに、その問題の解決方法を提案します。

DynamoDB 暗号化クライアントに関するフィードバックを提供するには、aws-dynamodb-encryption-java または aws-dynamodb-encryption-python GitHub リポジトリに問題を提出します。

このドキュメントに関するフィードバックを提供するには、任意のページのフィードバックリンクを使用します。

トピック

アクセスが拒否されました

問題: アプリケーションから必要なリソースにアクセスできない。

提案: 必要なアクセス許可について説明します。アプリケーションが実行されているセキュリティコンテキストにこのアクセス許可を追加します。

詳細

DynamoDB 暗号化クライアントライブラリを使用するアプリケーションを実行するには、そのコンポーネントを使用するためのアクセス許可が呼び出し元に必要です。それ以外の場合、必要な要素への発信者のアクセスは拒否されます。

- DynamoDB 暗号化クライアントは、Amazon Web Services (AWS) アカウントを必要とせず、どの AWS サービスにも依存しません。ただし、アプリケーションが を使用する場合は AWS、アカウントを使用するアクセス許可を持つ AWS アカウントおよび ユーザーが必要です。 https://docs.aws.amazon.com/IAM/latest/UserGuide/getting-started_create-admin-group.html
- DynamoDB 暗号化クライアントには Amazon DynamoDB は必要ありません。ただし、クライアントを使用するアプリケーションで DynamoDB テーブルを作成する、テーブルに項目を入力する、またはテーブルから項目を取得する場合、呼び出し元には、 AWS アカウントで必要な DynamoDB オペレーションを使用するためのアクセス許可が必要です。詳細については、Amazon DynamoDB デベロッパーガイドのアクセスコントロールのトピックを参照してください。
- アプリケーションが、Python 用 DynamoDB 暗号化クライアントでクライアントヘルパークラスを使用する場合、呼び出し元には、DynamoDB DescribeTable オペレーションを呼び出すためのアクセス許可が必要です。
- DynamoDB 暗号化クライアントには AWS Key Management Service () は必要ありませんAWS KMS。ただし、アプリケーションが Direct KMS マテリアルプロバイダーを使用している場合、または が使用するプロバイダーストアで最新プロバイダーを使用している場合 AWS KMS、呼び出し元には、 AWS KMS GenerateDataKey および Decrypt オペレーションを使用するアクセス許可が必要です。

署名の検証失敗

問題: 署名検証に失敗したため、項目を復号できない。また、項目は、意図したように暗号化および署名されていない場合があります。

提案: 指定した属性アクションが、項目内のすべての属性で構成されていることを確認してください。項目を復号する場合は、項目の暗号化に使用するアクションと一致する属性アクションを指定します。

詳細

指定する属性アクションによって、DynamoDB 暗号化クライアントに、暗号化して署名する属性、署名する (ただし、暗号化はしない) 属性、および無視する属性が伝達されます。

指定した属性アクションが、項目のすべての属性で構成されていない場合、その項目は、意図した方法で暗号化および署名されない場合があります。項目の復号時に指定する属性アクションが、項目の暗号化時に指定した属性アクションと異なる場合は、署名検証が失敗する場合があります。これは、分散アプリケーション固有の問題で、新しい属性アクションがすべてのホストに反映されていない可能性があります。

署名の検証エラーは解決が困難です。それらのエラーを防ぐために、データモデルの変更時に追加の対策を講じてください。詳細については、「データモデルの変更」を参照してください。

古いバージョンのグローバルテーブルの問題

問題: 署名の検証が失敗するため、古いバージョンの Amazon DynamoDB グローバルテーブルの項目を復号できません。

推奨: 予約されたレプリケーションフィールドが暗号化または署名されないように属性アクションを設定します。

詳細

DynamoDB グローバルテーブルを使用して DynamoDB Encryption Client を使用できます。マルチリージョン KMS キーを持つグローバルテーブルを使用し、グローバルテーブルがレプリケートされるすべての AWS リージョン に KMS キーをレプリケートすることをお勧めします。

グローバルテーブルのバージョン 2019.11.21 以降、特別な設定を行うことなく、DynamoDB Encryption Client でグローバルテーブルを使用できるようになりました。ただし、グローバルテーブルのバージョン 2017.11.29 を使用する場合は、予約されたレプリケーションフィールドが暗号化または署名されていないことを確認する必要があります。

グローバルテーブルのバージョン 2017.11.29 を使用している場合は、次の属性の属性アクションを Java で DO_NOTHING または Python で @DoNotTouch に設定する必要があります。

- aws:rep:deleting
- aws:rep:updatetime
- aws:rep:updateregion

他のバージョンのグローバルテーブルを使用している場合は、アクションは必要ありません。

最新プロバイダーのパフォーマンスが悪い

問題: 特に DynamoDB 暗号化クライアントの新しいバージョンに更新すると、アプリケーションの応答性が低下します。

提案: 有効期限 (TTL) 値とキャッシュサイズを調整します。

詳細

最新のプロバイダーは、暗号化マテリアルの再利用を制限できるようにすることで、DynamoDB 暗号化クライアントを使用するアプリケーションのパフォーマンスを向上させるように設計されています。アプリケーションの最新プロバイダーを設定するときは、パフォーマンスの向上と、キャッシュと再利用によって生じるセキュリティ上の問題とのバランスを取る必要があります。

DynamoDB 暗号化クライアントの新しいバージョンでは、有効期限 (TTL) の値によって、キャッシュされた暗号化マテリアルプロバイダー (CMP) の使用期間が決定されます。TTL により、最新プロバイダーが新しいバージョンの CMP をチェックする頻度も決定されます。

TTL が長すぎると、アプリケーションがビジネスルールやセキュリティ基準に違反する可能性があります。TTL が短すぎると、プロバイダーストアへの頻繁な呼び出しによって、プロバイダーストアがアプリケーションや、サービスアカウントを共有する他のアプリケーションからのリクエストを抑制する可能性があります。この問題を解決するには、レイテンシーと可用性の目標を満たし、セキュリティ基準に準拠する値に TTL とキャッシュサイズを調整します。詳細については、有効期限 (TTL) の値を設定する を参照してください

Amazon DynamoDB Encryption Client の名前の変更

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

2023 年 6 月 9 日、クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。 AWS Database Encryption SDK は Amazon DynamoDB と互換性があります。従来の DynamoDB Encryption Client によって暗号化された項目を復号して読み取ることができます。従来の DynamoDB Encryption Client のバージョンの詳細については、「AWS Database Encryption SDK for DynamoDB バージョンのサポート」を参照してください。

AWS Database Encryption SDK は、DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x を提供します。これは DynamoDB Encryption Client for Java の大幅な書き換えです。これには、新しい構造化データ形式、マルチテナンシーのサポートの改善、シームレスなスキーマの変更、検索可能な暗号化のサポートなど、多くの更新が含まれています。

AWS Database Encryption SDK で導入された新機能の詳細については、以下のトピックを参照してください。

検索可能な暗号化

データベース全体を復号せずに、暗号化されたレコードを検索できるデータベースを設計できます。脅威モデルとクエリ要件に応じて、検索可能な暗号化を使用して、暗号化されたレコードに対して完全一致検索やよりカスタマイズされた複雑なクエリを実行できます。

キーリング

AWS Database Encryption SDK は、キーリングを使用してエンベロープ暗号化を実行します。キーリングは、レコードを保護するデータキーを生成、暗号化、復号します。 AWS Database Encryption SDK は、対称暗号化または非対称 RSA を使用してデータキーを保護する AWS KMS キーリングと、レコードを暗号化または復号 AWS KMS するたびに AWS KMS keysを呼び出すことなく、対称暗号化 KMS キーで暗号化マテリアルを保護する階層 AWS KMS キーリングをサポートしています。Raw AES キーリングおよび Raw RSA キーリングを使用して独自のキーマテリアルを指定することもできます。

シームレスなスキーマ変更

AWS Database Encryption SDK を設定するときは、暗号化および署名するフィールド、署名するフィールド (暗号化しない) 、無視するフィールドをクライアントに指示する暗号化アクションを提供します。 AWS Database Encryption SDK を使用してレコードを保護した後でも、データモデルを変更できます。暗号化されたフィールドの追加や削除などの暗号化アクションを単一のデプロイで更新できます。

クライアント側の暗号化のために既存の DynamoDB テーブルを設定する

DynamoDB Encryption Client のレガシーバージョンは、データが入力されていない新しいテーブルに実装されるように設計されています。Database AWS Encryption SDK for DynamoDB を使用すると、既存の Amazon DynamoDB テーブルを DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x に移行できます。

参照資料

クライアント側の暗号化ライブラリの名前が AWS Database Encryption SDK に変更されました。このデベロッパーガイドでは、引き続き DynamoDB Encryption Client に関する情報を提供します。

以下のトピックでは、 AWS Database Encryption SDK の技術的な詳細について説明します。

マテリアルの説明の形式

マテリアルの説明は、暗号化されたレコードのヘッダーとして機能します。 AWS Database Encryption SDK を使用してフィールドを暗号化して署名すると、エンクリプタは暗号化マテリアルをアセンブルするときにマテリアルの説明を記録し、エンクリプタがレコードに追加する新しいフィールド (aws_dbe_head) にマテリアルの説明を保存します。マテリアルの説明は、暗号化されたデータキーと、レコードがどのように暗号化および署名されたかに関する情報を含む、ポータブルな形式のデータ構造です。次の表には、マテリアルの説明を構成する値が記載されています。バイトは示されている順に追加されます。

値	長さ (バイト)
Version	1
Signatures Enabled	1
Record ID	32
Encrypt Legend	変数
Encryption Context Length	2
???	可変
Encrypted Data Key Count	1
Encrypted Data Keys	可変
Record Commitment	1

バージョン

aws_dbe_head フィールドの形式のバージョン。

署名が有効

このレコードで ECDSA デジタル署名が有効になっているかどうかをエンコードします。

バイト値	意味
0x01	ECDSA デジタル署名が有効 (デフォルト)
0x00	ECDSA デジタル署名が無効

レコード ID

レコードを識別するランダムに生成された 256 ビットの値。レコード ID:

- 暗号化されたレコードを一意に識別します。
- マテリアルの説明を暗号化されたレコードにバインドします。

暗号化の凡例

どの認証済みフィールドが暗号化されたのかを示すシリアル化された説明。[暗号化の凡例] は、復号メソッドがどのフィールドの復号を試行するかを決定するために使用されます。

バイト値	意味
0x65	ENCRYPT_AND_SIGN
0x73	SIGN_ONLY

[暗号化の凡例] は次のようにシリアル化されます。

1. 辞書順 (正規パスを表すバイトシーケンスの順番)。
2. 各フィールドに、上記で指定したバイト値の 1 つを順番に付加して、そのフィールドを暗号化するかどうかを示します。

暗号化コンテキストの長さ

暗号化コンテキストの長さ。これは 16 ビットの符号なし整数として解釈される 2 バイトの値です。最大長は 65,535 バイトです。

暗号化コンテキスト

任意のシークレットではない追加認証データを含む名前と値のペアのセット。

ECDSA デジタル署名を有効にすると、暗号化コンテキストにはキーと値のペア が含まれます`{"aws-crypto-footer-ecdsa-key": Qtxt}`。 `Qtxt`は、SEC 1 バージョン 2.0 に従ってQ圧縮された楕円曲線ポイントを表し、base64 でエンコードされます。

暗号化されたデータキーの数

暗号化されたデータキーの数。これは、暗号化されたデータキーの数を指定する 8 ビットの符号なし整数として解釈される 1 バイトの値です。各レコード内の暗号化されたデータキーの最大数は 255 です。

暗号化されたデータキー

暗号化されたデータキーのシーケンス。シーケンスの長さは暗号化されたデータキーの数とそれぞれの長さによって決まります。シーケンスには、少なくとも 1 つの暗号化されたデータキーが含まれています。

以下の表では、暗号化された各データキーを形成するフィールドについて説明します。バイトは示されている順に追加されます。

暗号化されたデータキーの構造

フィールド	長さ (バイト)
Key Provider ID Length	2
Key Provider ID	変数。 前の 2 バイト (キープロバイダー ID の長さ) で指定された値と同じです。
Key Provider Information Length	2
Key Provider Information	変数。 前の 2 バイト (キープロバイダー情報の長さ) で指定された値と同じです。
Encrypted Data Key Length	2
Encrypted Data Key	変数。 前の 2 バイト (暗号化されたデータキーの長さ) で指定された値と同じです。

キープロバイダー ID の長さ

キープロバイダー ID の長さ。これは、キープロバイダー ID を含むバイト数を指定する 16 ビットの符号なし整数として解釈される 2 バイトの値です。

キープロバイダー ID

キープロバイダー ID。これは、暗号化されたデータキーのプロバイダーを示すために使用され、拡張することを目的としています。

キープロバイダー情報の長さ

キープロバイダー情報の長さ。これは、キープロバイダー情報を含むバイト数を指定する 16 ビットの符号なし整数として解釈される 2 バイトの値です。

キープロバイダー情報

キープロバイダー情報 これはキープロバイダーによって決定されます。

AWS KMS キーリングを使用している場合、この値には の Amazon リソースネーム (ARN) が含まれます AWS KMS key。

暗号化されたデータキーの長さ

暗号化されたデータキーの長さ。これは、暗号化されたデータキーを含むバイト数を指定する 16 ビットの符号なし整数として解釈される 2 バイトの値です。

暗号化されたデータキー

暗号化されたデータキー これは、キープロバイダーによって暗号化されたデータキーです。

コミットメントを記録する

コミットキーを使用して、前述のすべてのマテリアル説明バイトで計算された個別の 256 ビットハッシュベースのメッセージ認証コード (HMAC) ハッシュ。

AWS KMS 階層キーリングの技術的な詳細

AWS KMS 階層キーリングは、一意のデータキーを使用して各フィールドを暗号化し、アクティブなブランチキーから導出した一意のラッピングキーを使用して各データキーを暗号化します。HMAC SHA-256 の擬似ランダム関数を使用したカウンターモードで鍵導出を使用して、次の入力で 32 バイトのラッピングキーを導出します。

- 16 バイトのランダムソルト

- アクティブなブランチキー
- キープロバイダー識別子「aws-kms-hierarchy」の UTF-8 でエンコードされた値

階層キーリングは、導出されたラッピングキーと、16 バイトの認証タグと次の入力を含む AES-GCM-256 を使用して、プレーンテキストデータキーのコピーを暗号化します。

- 導出されたラッピングキーは AES-GCM 暗号キーとして使用されます
- データキーは AES-GCM メッセージとして使用されます
- 12 バイトのランダム初期化ベクトル (IV) が AES-GCM IV として使用されます
- 次のシリアル化された値を含む追加認証データ (AAD)。

値	長さ (バイト)	次のように解釈されます
「aws-kms-hierarchy」	17	UTF-8 でエンコード済み
ブランチキーの識別子	変数	UTF-8 でエンコード済み
ブランチキーのバージョン	16	UTF-8 でエンコード済み
暗号化コンテキスト	変数	UTF-8 でエンコードされた key-value ペア

AWS Database Encryption SDK デベロッパーガイドのドキュメント履歴

以下の表は、このドキュメントの大きな変更点をまとめたものです。主要な変更に加えて、その内容の説明と例を改善し、ユーザーから寄せられるフィードバックにも応える目的で、このドキュメントは頻繁に更新されます。重要な変更についての通知を受け取るには、RSS フィードをサブスクライブします。

変更	説明	日付
新機能	AWS KMS ECDH キーリングと Raw ECDH キーリングのドキュメントを追加しました。	2024 年 6 月 17 日
一般提供 (GA)	DynamoDB 用の .NET クライアント側の暗号化ライブラリのサポートを紹介します。	2024 年 1 月 17 日
一般提供 (GA)	DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x の GA リリースのドキュメントを更新しました。 **Warning** デベロッパープレビューリリース中に作成されたブランチキーはサポートされなくなりました。	2023 年 7 月 24 日
DynamoDB Encryption Client のブランドの変更	クライアント側の暗号化ライブラリの名前が AWS	2023 年 6 月 9 日

	Database Encryption SDK に変更されました。	
プレビューリリース	DynamoDB 用の Java クライアント側の暗号化ライブラリのバージョン 3.x のドキュメントを追加および更新しました。これには、新しい構造化データ形式、マルチテナンシーサポートの改善、シームレスなスキーマ変更、および検索可能な暗号化サポートが含まれます。	2023 年 6 月 9 日
ドキュメントの変更	カスタマーマスターキー (CMK) という AWS Key Management Service 用語を AWS KMS keyおよび KMS キーに置き換えます。	2021 年 8 月 30 日
新機能	AWS Key Management Service (AWS KMS) マルチリージョンキーのサポートが追加されました。マルチリージョンキーは AWS KMS 異なる のキー AWS リージョン であり、キー ID とキーマテリアルが同じであるため、同じ意味で使用できます。	2021 年 6 月 8 日
新しい例	Java で DynamoDBMapper を使用する例を追加しました。	2018 年 9 月 6 日
Python サポート	Java に加えて Python のサポートを追加しました。	2018 年 5 月 2 日
初回リリース	このドキュメントの初回リリース。	2018 年 5 月 2 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。